Optimization of Trajectories

Kristin Cordwell
Erika Debenedictis
Brian Lott

April 04, 2007

Team 38
New Mexico Supercomputing Challenge

Table of Contents

Executive Summary

We have developed a process to find gravity assisted routes for spacecraft launched from Earth to distant objects in our solar system. These routes use the gravitational effects of planets and Lagrange points to affect the trajectories and are both fuel and time effecient. By analyzing characteristics common to the identified routes it is possible to predict new routes that may be more efficient.

In developing this program, we experimented with different approximation methods, including Simple Newtonian and Runge-Kutta Approximations, for calculating the movement of a spacecraft. Using a computer cluster, we were able to partition the calculations over multiple computers and document the effect on the program's performance.

We also investigated two conceptual models for finding routes for a spacecraft. The first involves setting up an array of spacecraft, observing their trajectories, and then refining the initial conditions, thus moving the spacecraft trajectories closer to the target. This method was effective in finding gravity assisted paths to get to both Mars and Pluto. In another model, we investigated the fuel-saving effects of utilizing the Lagrange points when planning the spacecraft's route. This method excelled in identifing longer duration paths where the spacecraft reaches its target with significantly lower fuel requirements.

Problem Statement

In order to find efficient and fast routes to other planets, it is necessary to use the gravity of planetary bodies to boost the spacecraft's speed and change the spacecraft's position and velocity vector. Simulating the spacecraft's path can reveal the optimal way to launch a spacecraft so that it uses little fuel and reaches its destination in a short amount of time. These simulations are especially important because of the expense of the probes and the scientific value of the data they collect.

By analyzing characteristics common to routes that use little fuel and take a reasonable amount of time, it should be possible to narrow down which routes will be most efficient. These characteristics include gravity slingshot effects of an interaction with a planet, the speed and direction of the spacecraft as it leaves earth, relative position to multiple planetary bodies, and relative position to Lagrange points.

The n-body problem of many gravitational fields affecting an object can be expressed, at a given time, geometrically by a three dimensional graph of the gravitional potential, and this can help one investigate the properties of various Lagrange points. Dynamic gravitational effects involving the movement of multiple planets can greatly change the path of an object, and we can use the theory of elastic collisions to predict how best to use a gravitational slingshot. By using three ways to change position and speed, namely Lagrange points, the slingshot effect, and burning fuel, spacecraft can be accurately and efficiently moved around the Solar System.

Gravitational Physics

Gravity! Gravity!
All matter has a force
That pulls things toward its core.
Gravity! Gravity!
Is what we call that force.

-Space Songs

Single Central Mass

An object of mass M produces a gravitational force on another mass m equal to $-\frac{GM\vec{r}}{r^3}$, where $\vec{r}$ is the vector from mass M to object m, and r is the distance, or the magnitude of $\vec{r}$. In terms of the unit vector $\hat{r}$ from M to m, the force equals $-\frac{GM\hat{r}}{r^2}$, which is proportional to the masses M and m, and inversely proportional to the distance between the two objects. G is the gravitational constant, approximately equal to $6.67 \cdot 10^{-11}\ Nm^2/kg^2$.

If m is much less than M ($m << M$), we can ignore the effect of m on M and solve for the motion of m in the gravitational field created by M. We apply Newton's second law to obtain the equation $m\vec{a} = -\frac{GM\vec{r}}{r^3}$, or, equivalently, $\frac{d^2\vec{r}}{dt^2} = -\frac{GM\vec{r}}{r^3}$.

This is normally solved in the reference frame where M is not moving. The solution is then automatically in only two dimensions. One usually uses polar coordinates and obtains the equations

$\frac{d^2r}{dt^2} - r(\frac{d\theta}{dt})^2 = -\frac{GM}{r^2}$ and
$\frac{1}{r} \cdot \frac{d}{dt}(r^2\frac{d\theta}{dt}) = 0.$

The second equation says that $r^2\frac{d\theta}{dt}$ is a constant, which we recognize as the specific angular momentum of m. Calling this value l, we obtain

$$\frac{d^2r}{dt^2} - \frac{l^2}{r^3} = -\frac{GM}{r^2}.$$

This is usually solved by a substitution trick, letting $r = \frac{1}{u}$, obtaining a simpler differential equation in u, solving it, and then substituting back. The general solution is then $r(\theta) = \frac{l^2/GM}{1 + e\cos(\theta - \theta_0)}$, where e is the eccentricity of the orbit. This solution describes curves that are conic sections; that is, parabolas, hyperbolas, or ellipses. If the eccentricity e is less than 1, the curve is an ellipse and we have the relation $\frac{l^2}{GM} = a(1 - e^2)$, where a is the semi-major axis of the ellipse.

Two Body Systems

When there are two attracting bodies we are still able to solve the system exactly. The bodies attract each other, but in a way that causes them to orbit about the center of mass of the two bodies. By a trick of switching the coordinate to be the vector between the two bodies, the two body problem reduces, mathematically, to the one (attracting) body problem. After this is solved, one just multiplies by $\frac{M_1}{M_1 + M_2}$ and $-\frac{M_2}{M_1 + M_2}$, respectively, to recover the orbital position vectors of the two bodies.

N Body Systems

There is no general, closed-form solution for more than two attracting bodies. One must use approximations and solve simultaneous equations numerically.

Elastic Collisions

Introduction

Elastic collisions between two objects occur when the total kinetic energy is conserved. For example, if a ball of modeling clay hits a hard floor, it is likely not to bounce much, if at all – this is an inelastic collision, as the energy of the ball is converted into internal friction and heat. If a superball (think "elastic") hits the floor, though, it will bounce back to almost the same initial height from which is was dropped – this collision is (almost) elastic. We can set up the equations for Conservation of Momentum and Conservation of Energy to find out how two objects behave during an elastic collision.

The reason that we spend time looking at elastic collisions is that most situations where a spacecraft approaches a planet or a large body are, in fact, elastic collisions. The mechanism for the collision does not need to be an impact and a bounce, it only needs to provide some means of interaction so that the bodies trajectories are altered, without loss of kinetic energy. Gravitational interactions provide just such a means; a spacecraft that flies near the sun or a planet does, indeed, have its trajectory altered, and, because of the nature of the gravitational fields, almost no energy is lost to internal energy (unless the spacecraft hits the planet!).

As we will see from the equations below, in an elastic collision it is possible for a small body to gain a boost in speed. This is the basis for what is called a slingshot maneuver.

One Dimensional Collisions

Collisions in one dimension occur on a line; the objects are assumed to be moving on this line before and after the collision.

Conservation of Momentum implies that

$$mv_1 + Mv_2 = m\hat{v_1} + M\hat{v_2} \tag{1}$$

Conservation of Energy implies that

$$\frac{1}{2}m{v_1}^2 + \frac{1}{2}M{v_2}^2 = \frac{1}{2}m{\hat{v_1}}^2 + \frac{1}{2}M{\hat{v_2}}^2 \tag{2}$$

From equation (2), we get

$$m(v_1 + \hat{v_1})(v_1 - \hat{v_1}) = M(\hat{v_2} + v_2)(\hat{v_2} - v_2) \tag{3}$$

From equation (1), we get

$$m(v_1 - \hat{v_1}) = M(\hat{v_2} - v_2) \tag{4}$$

Substituting equation (4) into equation (3) gives us

$$(v_1 + \hat{v_1}) = (\hat{v_2} + v_2) \tag{5}$$

Equations (5) and (4) then become

$mv_1 + m\hat{v_1} = mv_2 + m\hat{v_2}$
$mv_1 - m\hat{v_1} = M\hat{v_2} - Mv_2$

Solving this system, we obtain

$2mv_1 = (m - M)v_2 + (M + m)\hat{v_2}$

Or, solving for $\hat{v_2}$,

$$\hat{v_2} = \frac{2mv_1 + (M - m)v_2}{(M + m)}$$

Similarly, we can use equations (5) and (4) to solve for $\hat{v_1}$ as follows

$Mv_1 + M\hat{v_1} = Mv_2 + M\hat{v_2}$

$-mv_1 + m\hat{v_1} = -M\hat{v_2} + Mv_2$

$(M - m)v_1 + (M + m)\hat{v_1} = 2Mv_2$

which gives us

$$\hat{v_1} = \frac{2Mv_2 - (M-m)v_1}{(M+m)}.$$

One Mass Much Larger than the Other

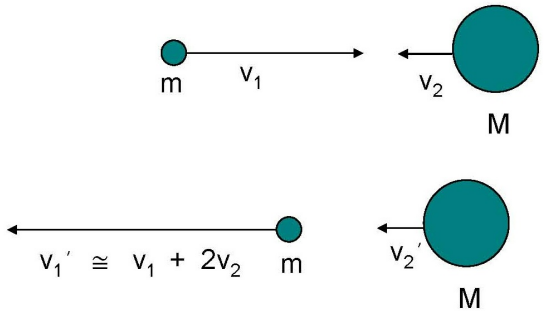
Assuming that $M >> m$,

$\hat{v_1} \approx 2v_2 - v_1$.

If v_1 and v_2 are in opposite directions, object m collides head-on with M and bounces backwards. In this case, it gains speed in the amount

$|\hat{v_1}| \approx v_1 + 2v_2$.

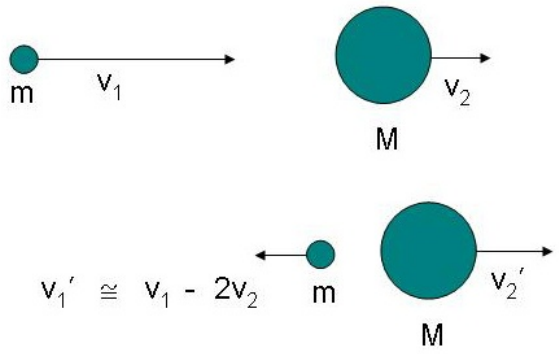
Figure 1: One Dimensional Collision - Gain in Speed



If v_1 and v_2 are in the same direction and object m is moving fast enough, it hits M from behind and bounces backwards, but with a reduced speed.

$|\hat{v_1}| \approx v_1 - 2v_2 \qquad (\text{if } v_1 > 2v_2)$

Figure 2: One Dimensional Collision - Loss of Speed



If m is not moving as fast, it hits M from behind and continues in the same direction, also with a reduced speed.

$|\hat{v_1}| \approx 2v_2 - v_1 \qquad (\text{if } v_1 > v_2 > \frac{1}{2}v_1)$

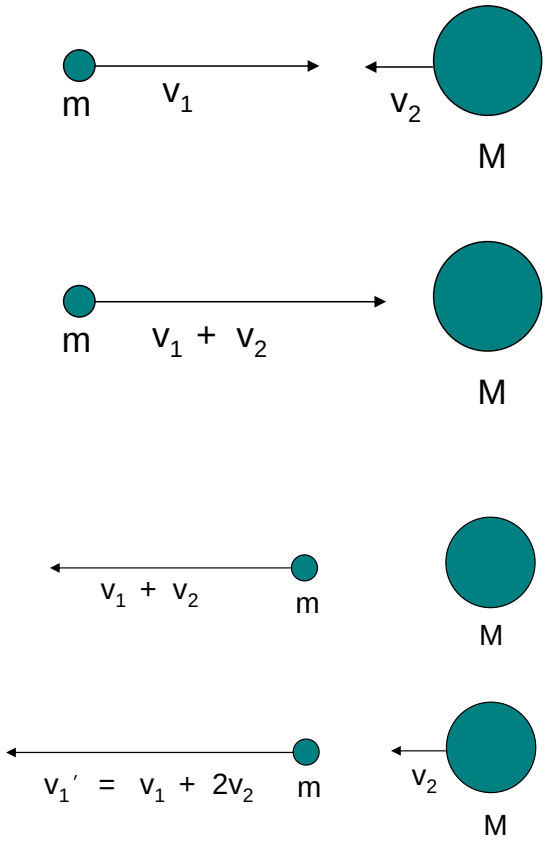
If we now consider only the case where $M >> m$, then there is an easier way to derive the final speed v_1. We first change to the reference frame where M is not moving, i.e., $v_2 = 0$, as shown.

Conservation of Energy in this frame requires that the final speed of m be the same as the initial speed (we are using $M >> m$ here).

As seen in the figure, we add back in v_2 to get our final result.

Two Dimensional Collisions

Elastic collisions in two dimensions again require conservation of momentum and conservation of energy. Since momentum is a vector quantity, this gives



us two equations, one for the x-component and one for the y-component. Together with the equation for energy, we get
$mv_1 = m\hat{v_1}cos(\theta_1) + M\hat{v_2}cos(\theta_2)$

$0 = -m\hat{v_1}sin(\theta_1) + M\hat{v_2}sin(\theta_2)$

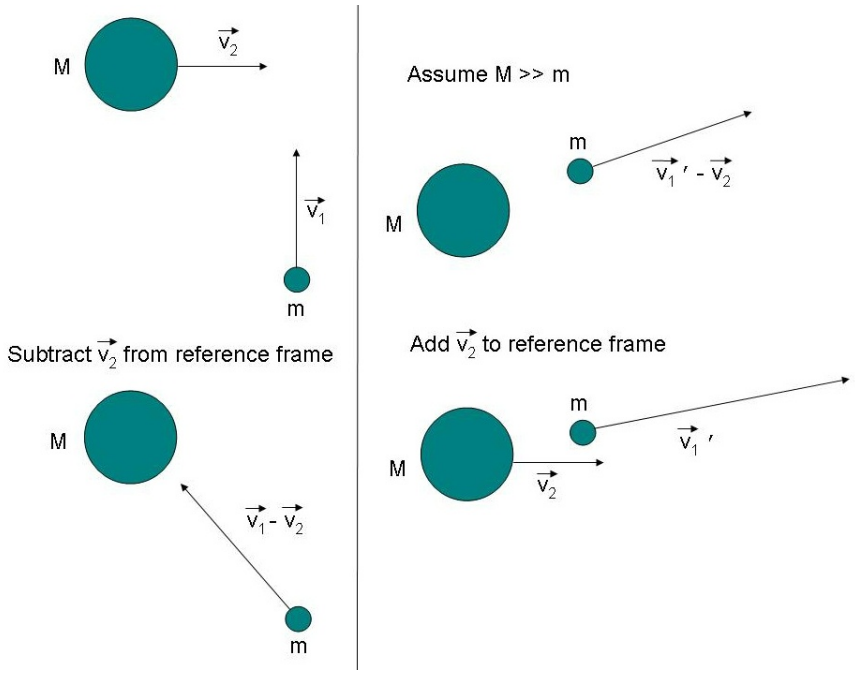
$\frac{1}{2}m{v_1}^2 - \frac{1}{2}m{\hat{v_1}}^2 + \frac{1}{2}M{\hat{v_2}}^2$

The general solution for $\hat{v_1}$ and $\hat{v_2}$ is straightforward but fairly complex. Since almost all of our calculations involve the condition that $M >> m$, we can simplify these equations to obtain a final approximation that will be extremely accurate.

Again, we change to the reference frame where M is stationary. The condi-

tion $M >> m$ allows us to say that M remains stationary, and conservation of energy then requires that the final speed of m in this frame equals the initial speed in this frame. As is the case in real life, m may bounce off at any possible angle. As seen in the picture below, we obtain the final velocity of m by adding back in $\vec{v_2}$.

Figure 3: Two Dimensional Collision, $M >> m$



Slingshot Maneuvers

The reason that we spend time on two dimensional elastic collisions is that it forms the basis for slingshot maneuvers. First, we note that gravitational interactions are almost always elastic collions - a collision does not require that the two bodies physically crash together, just that they interact in a way to change the trajectories. Moreover, by the nature of a gravitational interaction, there is essentially no energy lost during the interaction, so the

total (kinetic) energy is conserved automatically. (The only inelastic collisions would be if a spacecraft actually crashed into a planet).

Now, by using the theory developed above, we can plot the ingoing angle of a spacecraft that is heading towards a planet, and we can decide what outgoing speed we might wish to obtain (up to the possible maximum). This will determine the desired outgoing angle. We can then tweak the ingoing trajectory at a large distance from the planet, using only a small amount of energy, to cause it to go either nearer or farther from the planet, so as to adjust the "collision" until it produces the desired outgoing angle (and speed).

This method allows us to patch together trajectories from, say, Earth to Pluto by way of a slingshot about Jupiter in a fairly efficient manner.

Lagrange Points

Introduction

Newton's Laws can be solved exactly for a system of two masses (often unequal in size), where each mass attracts the other. One solution is when each of the two masses orbits the common center of mass in a circle. Essentially, Newton's third law says that the forces on the two bodies are equal in magnitude (and opposite in direction), but, because of the difference in masses, the acceleration is different for each. Thus, the larger mass will have a smaller orbit about the center of mass, and the larger will have a bigger orbit. We treat the center of mass as the origin, and it turns out that each of the two masses M_1 and M_2 orbits with the same angular frequency, which we will call Ω.
Lagrange Points are special positions in such a system, where a test body would rotate about the common center at the same angular speed as the two attracting masses, at a constant distance. Thus, in a reference frame that is rotating with the attracting masses, the test body would also appear stationary.

Rotational Physics

Kepler's Law still holds in the form
$\Omega^2 R^3 = G(M_1 + M_2)$,
where R is the distance between M_1 and M_2 and the angular frequency is $\frac{2\pi}{T}$, T being the orbital period. If R is constant, and we choose a reference frame that rotates with the same angular velocity about the origin, it will appear that M_1 and M_2 are stationary in that reference frame. This makes it fairly straightforward to test for the existence of equilibrium points.
In a rotating frame, we must include "artificial forces" as the price of the convenience of rotating along with a system. These include the centrifugal force and the coriolis force. The coriolis force depends on an object moving (in the rotating frame), so we can ignore it in our search for equilibrium points, which will be stationary. In the rotating frame, the force on a test body m at position $\vec{r}$ from the origin will be
$\vec{F_\Omega} = \vec{F} - m\vec{\Omega} \times (\vec{\Omega} \times \vec{r})$. Vector Ω is the angular velocity, and $\vec{F}$ is the force measured in the non-rotating frame, viz., the attractive force due to M_1 and M_2. If we restrict ourselves to the plane defined by the orbits of M_1 and M_2,

we see that the centrifugal force has magnitude $mr\Omega^2 = \frac{mv^2}{r}$, which is just the amount of centripetal force required to keep an object in a circular orbit.

Potential Energy in the Rotating Frame

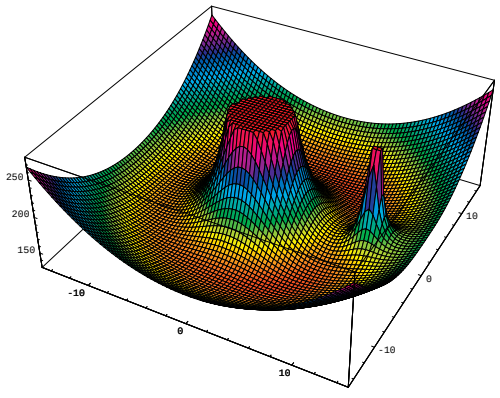
We also have that the gravitational force field has an associated specific potential energy,

$$U_\Omega = U - \frac{1}{2}r^2\Omega^2,$$

the latter term just being, in magnitude, $\frac{v^2}{2}$, the specific kinetic energy in the non-rotating frame, and where U is the potential energy due to the two bodies M_1 and M_2.

If we plot the potential energy (in the rotating frame), we see some interesting features. In the first graph, we are plotting the negative of the potential energy, and we can see the large peaks where the masses M_1 and M_2 are located. We plotted this in Mathematica, using the value k= 80 and the command
Plot3D[10*k/Sqrt[(x + 1)^ 2 + y^ 2 + 0.0001] + 1*k/Sqrt[(x - 10)^ 2 + y^ 2 + 0.0001] + 1*(x^ 2 + y^ 2)/2, {x, -15, 15}, {y, -15, 15}, ColorFunction → Hue, PlotPoints → 80, ImageSize → 500].

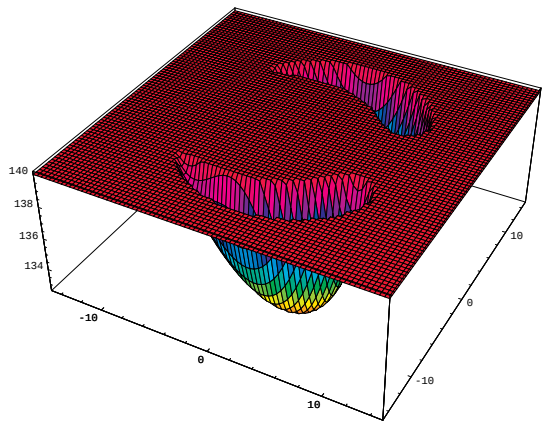
Figure 4: Masses M_1 and M_2



M_1 was chosen to be 10 times as large as M_2, which means that the position of M_1 was one unit to the left of the origin, and that of M_2 is ten units to the right. Ω was chosen to equal 1, and one can see that we are adding potential and kinetic (centrifugal) energies.

As well as a couple of saddle points in the first figure, there are two other equilibrium points off to the sides. To see these, we enhanced the negative values and cut off the large values around the two masses. This was done with the Mathematica command
Plot3D[Min[10*k/Sqrt[(x + 1)^ 2 + y^ 2 + 0.0001] + 1*k/Sqrt[(x - 10)^ 2 + y^ 2 + 0.0001] + 1*(x^ 2 + y^ 2)/2, 140], {x, -15, 15}, {y, -15, 15}, ColorFunction → Hue, PlotPoints → 80, ImageSize → 500]

Figure 5: Off-Axis Lagrange Points

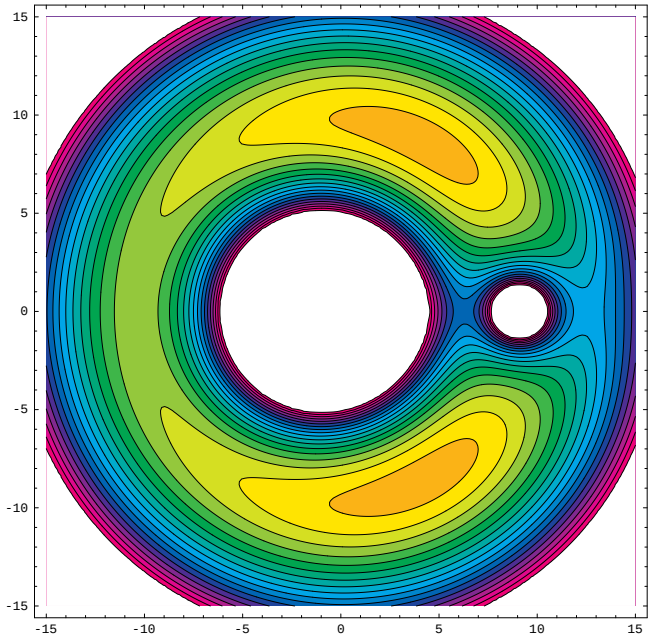


Again, we are plotting the negative of the potential energy, so the two dips are really hills. These would, in a non-rotating frame, be unstable equilibrium points.

Finally, we made a contour plot, looking down from above. Here we can clearly see two saddle points, two of them near M_2, one of which is between M_1 and M_2, and the other to the right of M_2. Then there are the two points on top of the hills, off to the sides of the M_1-M_2 axis. Finally, there is actually a fifth saddle point to the left of M_1, in the middle of the broad, light

green region that extends from the two hills.

Figure 6: Potential Energy Contours



This graph was produced using the Mathematica command ContourPlot[Min[m1/Sqrt[(x + a*r)^ 2 + y^ 2 + 0.0001] + m2/Sqrt[(x - b*r)^ 2 + y^ 2 + 0.0001] + (m1 + m2)/(r^ 3)*(x^ 2 + y^ 2)/ 2, .22], {x, -15, 15}, {y, -15, 15}, Contours → 18, ColorFunction → (If[# > .95, RGB-Color[1, 1, 1], Hue[(# + .14)*.80]] &), PlotPoints → 200, ImageSize → 500] with the values $r = 10$, $m1 = 1$, $m2 = 1/10$, $a = \frac{m2}{m1+m2}$, $b = \frac{m1}{m1+m2}$.

Finding the Positions of L1, L2, and L3

To solve for the positions of the Lagrange points, we find the force on a test body in the potential field. First, we define the fractional masses of the two attractive bodies:

$$\alpha = \frac{M_2}{M_1 + M_2}, \beta = \frac{M_1}{M_1 + M_2}$$

Note that $\alpha + \beta = 1$. In the rotating frame, M_1 has coordinates $(-\alpha R, 0)$ and M_2 has coordinates $(\beta R, 0)$.
Then, taking $\vec{F_\Omega}$ as above, with m, the test mass, equal to 1, and noting that this is simply $\vec{F} + \Omega^2 \vec{r}$, we obtain

$$\begin{aligned}
\vec{F_\Omega} &= (x \cdot \Omega^2 - \frac{GM_1(x + \alpha R)}{((x + \alpha R)^2 + y^2)^{\frac{3}{2}}} - \frac{GM_2(x - \beta R)}{((x - \beta R)^2 + y^2)^{\frac{3}{2}}})\hat{i} \\
&\quad + (y \cdot \Omega^2 - \frac{GM_1 y}{((x + \alpha R)^2 + y^2)^{\frac{3}{2}}} - \frac{GM_2 y}{((x - \beta R)^2 + y^2)^{\frac{3}{2}}})\hat{j} \\
&= \Omega^2(x - \frac{\beta(x + \alpha R)R^3}{((x + \alpha R)^2 + y^2)^{\frac{3}{2}}} - \frac{\alpha(x - \beta R)R^3}{((x - \beta R)^2 + y^2)^{\frac{3}{2}}})\ \hat{i} \\
&\quad + \Omega^2(y - \frac{\beta y R^3}{((x + \alpha R)^2 + y^2)^{\frac{3}{2}}} - \frac{\alpha y R^3}{((x - \beta R)^2 + y^2)^{\frac{3}{2}}})\ \hat{j}
\end{aligned}$$

where we also used $\frac{1}{\Omega^2} = \frac{R^3}{GM_1 + GM_2}$.

Setting this equal to 0, which is the condition for an equilibrium point, we can then find the solutions. First, we specialize to the case along the x-axis, where $y = 0$. This automatically forces the y component of the force to be zero, leaving the equation for the x component. Then we need to solve

$$x - \frac{\beta(x + \alpha R)R^3}{|x + \alpha R|^3} - \frac{\alpha(x - \beta R)R^3}{|x - \beta R|^3} = 0$$

Substituting $w = \frac{x}{R}$, and dividing the equation by R and rearranging terms, we obtain

$$w - \frac{\beta(w + \alpha)}{|w + \alpha|^3} - \frac{\alpha(w - \beta)}{|w - \beta|^3} = 0$$

where w is in units of the distance between M_1 and M_2, R.
Finally, substituting $s = w - \beta$, so that s is the coordinate distance, in units of R of the Lagrange Point(s) from M_2, we obtain

$$s + \beta - \frac{\beta(s+1)}{|s+1|^3} - \frac{\alpha s}{|s|^3} = 0$$

where, again, we have used $\alpha + \beta = 1$.

Because of the absolute value signs, there are three cases to consider, $s > 0$, which is the Lagrange Point to the right of M_2, $0 > s > -1$, which is the Lagrange Point just to the left of M_2, and $-1 > s$, which is the Lagrange Point to the far left of M_1. We will solve the case for $s > 0$, as the solutions in the other two cases are very similar.

With $s > 0$, we have

$$s + \beta - \frac{\beta}{(s+1)^2} - \frac{\alpha}{s^2} = 0$$

We now substitute in $\beta = 1 - \alpha$, clear the denominators, and obtain
$s^3(3 + 3s + s^2) = \alpha(1 + 2s + s^2 + 2s^3 + s^4)$

This is an equation for s of degree five, so we know that it has at least one real root (it is of odd degree, and complex roots occur in conjugate pairs), which we could solve for numerically, given the value of α.
It is often the case, though, that $M_1 >> M_2$, or, equivalently, that $\alpha << 1$. For example, for the sun-earth system, $\alpha \approx 3 \cdot 10^{-6}$. In this situation, we assume that the solution s is also small and solve to first order in α. After this, of course, we need to check that our assumption was consistent, i.e., that s really is small. With the assumptions that $s << 1$ and $\alpha << 1$, we have
$3s^3 \approx \alpha$, or that $s \approx (\frac{\alpha}{3})^{\frac{1}{3}}$.

We now need to check that s is, indeed, much less than 1. This is a bit "fuzzy", or not clearly defined, but, if we consider the earth-moon system as representative, $\alpha \approx .012$, giving a value of s of about .16, which is probably good enough. For the sun-earth system, s is much less than 1. Thus, our assumption is justified, and our solution is close to the true solution.

In terms of our original coordinate x, we then have
$x = R(\beta + (\frac{\alpha}{3})^{\frac{1}{3}}) = R(1 - \alpha + (\frac{\alpha}{3})^{\frac{1}{3}})$.

Since a (positive) number that is less than one has a cube root that is bigger than the original number (but still less than one), we could drop the α term, if α is small enough. The Lagrange Point for which we have solved is usually called L2. The one between the two masses is called L1, and the one to the left of the larger mass is called L3. The solutions for L1 and L3 are found in a similar way to that of L2. For cases where $M_2 << M_1$, we have, approximately, the coordinate positions

$$\text{L1}: \ (R[1 - (\frac{\alpha}{3})^{\frac{1}{3}}], 0)$$
$$\text{L2}: \ (R[1 + (\frac{\alpha}{3})^{\frac{1}{3}}], 0)$$
$$\text{L3}: \ (-R[1 + \frac{5}{12}\alpha], 0)$$

Physical Description of L1, L2, and L3

Physically, a satellite that is orbiting, say, the earth will go around much faster if it is in a low earth orbit, both in real speed and in angular speed, than if it is in a higher orbit, because the centripetal force is higher, which pulls it around faster. The Lagrange Points find where the orbital speed exactly matches that of the moon. Normally, the L1 point, being closer to the earth, would be going around faster than the moon, but the pull of the moon partially counteracts the pull from the earth, which reduces the acceleration to exactly the point where the the corresponding reduced angular speed matches that of the moon. Similarly, the L2 point, being farther out from the earth than is the moon, would normally be orbiting more slowly than the moon. However, the pull of the moon adds to the pull of the earth in this case, increasing the acceleration to the point where its angular speed also matches that of the moon. Finally, the L3 point, which is slightly farther away from the earth than is the moon, also feels a slight extra pull from the moon (on the far side of the earth), so its angular speed also aligns with that of the moon.

Finding the Positions of L4 and L5

We noted that there are two other Lagrange Points, on the tops of the two "hills", offset from the x-axis. To solve for these, we also use the force equation from above,

$$\begin{aligned}\vec{F}_\Omega &= \Omega^2(x - \frac{\beta(x+\alpha R)R^3}{((x+\alpha R)^2+y^2)^{\frac{3}{2}}} - \frac{\alpha(x-\beta R)R^3}{((x-\beta R)^2+y^2)^{\frac{3}{2}}})\ \hat{i} \\ &+ \Omega^2(y - \frac{\beta y R^3}{((x+\alpha R)^2+y^2)^{\frac{3}{2}}} - \frac{\alpha y R^3}{((x-\beta R)^2+y^2)^{\frac{3}{2}}})\ \hat{j}\end{aligned}$$

and set it equal to zero. It makes it a bit more clear, though, if we decompose the force vector into a component that is parallel to $\vec{r}$ and one perpendicular to that. The two relevant vectors to project onto are $\frac{1}{r}(x\hat{i} + y\hat{j})$ and $\frac{1}{r}(y\hat{i} - x\hat{j})$, where $r = \sqrt{x^2+y^2}$.
Taking the dot product of these two vectors with $\vec{F_\Omega}$ gives

$$F_\Omega^\perp = k_1[xy - \frac{\beta(x+\alpha R)R^3 y}{((x+\alpha R)^2+y^2)^{\frac{3}{2}}} - \frac{\alpha(x-\beta R)R^3 y}{((x-\beta R)^2+y^2)^{\frac{3}{2}}} - xy + \frac{\beta x y R^3}{((x+\alpha R)^2+y^2)^{\frac{3}{2}}} + \frac{\alpha x y R^3}{((x-\beta R)^2+y^2)^{\frac{3}{2}}}].$$

where k_1 is the appropriate value, which will not be important as we will be setting the force components to zero. Simplifying, we obtain

$$F_\Omega^\perp = \hat{k_1}[-\frac{1}{((x+\alpha R)^2+y^2)^{\frac{3}{2}}} + \frac{1}{((x-\beta R)^2+y^2)^{\frac{3}{2}}}].$$

where $\hat{k_1}$ is another unimportant nonzero value. Setting the piece inside the brackets to zero implies that $(x-\beta R)^2 = (x+\alpha R)^2$. Since $-\alpha R$ and βR are the x-coordinates of the two bodies M_1 and M_2, we see that the x-coordinate of both of these Lagrange Points must lie halfway between the positions of the two bodies, namely $x = \frac{1}{2}(\beta - \alpha)R$. Thus, these Lagrange Points must be at the apexes of isosceles triangles with bases equaling the

segment between the two bodies.

By the similar project , we obtain the component of the force that is parallel to the radius vector. This reduces to

$$F_{\Omega}^{\parallel} = \hat{k_2}[\frac{1}{R^3} - \frac{1}{((x+\alpha R)^2+y^2)^{\frac{3}{2}}}].$$

Setting the part inside the brackets to zero and simplifying gives

$$(x+\alpha R)^2 + y^2 - R^2 = 0.$$

Now substituting the value of x determined from the perpendicular component, we have

$$(x+\alpha R)^2+y^2-R^2 = (\tfrac{1}{2}(\beta-\alpha)R+\alpha R)^2+y^2-R^2 = (\tfrac{1}{2}(\alpha+\beta)R)^2+y^2-R^2 = (\tfrac{1}{2}R)^2+y^2-R^2 = 0,$$

which solves to $y = \pm\frac{\sqrt{3}}{2}R$. We recognize these y values as the apexes of the equilateral triangle whose bases are the segment between the two bodies. Thus, the Lagrange Points L4 and L5 are located at the opposite points of the two equilateral triangles that share the base segment joining M_1 and M_2.

Physical Description of L4 and L5

Physically, the earth pulls more strongly on a test body at these Lagrange Points, with the contribution of the moon's pull adding as a vector to yield a net pull, of just the right size, towards the center of mass to cause these Lagrange Points to rotate at the same angular speed as the system.

Stability of the Lagrange Points

Although the contour plot seems to show that L1, L2, and L3 are equilibrium saddle points, and L4 and L5 are unstable equilibrium points, because we are in a rotating frame, we need to be a bit cautious. It turns out that L1, L2,

and L3 are indeed saddle points that are dynamically unstable, with L3 being the "least unstable". On the other hand, although L4 and L5 are hilltops in the contour plot, because of the coriolis force, a small perturbation off of the hilltop will cause a test body to orbit around L4 or L5, as long as the larger mass, M_1 is at least approximately 25 times as large as M_2 (this is true for the earth-moon or sun-earth systems).

Use of the Lagrange Points

In recent decades, there has been much discussion and some actual physical use of the Lagrange Points. As might be inferred from the contour potential, the L1 and L2 points allow access into M_2's potential well with smaller energy than might otherwise be required. Moreover, if a spacecraft has just enough speed to get through the "gate" at, say, L2, and then it performs a low-energy burn to reduce its speed, it will be trapped inside the well. Thus, it is possible to utilize the Lagrange Points in accessing regions that might otherwise seem to be forbidden, and also possible to utilize them to transfer to an orbit about M_2 in a low-energy manner.

Numerical Analysis

Introduction and Basic Requirements

We are trying to solve spacecraft trajectories, which are solutions of equations of the form $\frac{d^2\vec{x}}{dt^2} = \vec{f}(\vec{x})$, where $\vec{f}$ is the force term. For example, for a single attractive body,
$\vec{f}(\vec{x}) = -\frac{GM\vec{x}}{x^3}$.
Note that the force is not explicitly dependent on time or the velocity of any of the bodies, only on the position vector.
Although we are dealing with (in general) 3-dimensional vector quantities, these all separate out into their respective coordinate parts, so we will suppress the vector notation in the following derivation.

We have the second order equation $\frac{d^2x}{dt^2} = f(x)$, but we change to a system of first order equations as

$$\begin{cases} \frac{dx}{dt} = v \\ \\ \frac{dv}{dt} = f(x) \end{cases}$$

We want to find the position and velocity at time $t = 1$, given their values at $t = 0$. First we express x_1 and v_1 as a Taylor series about the values at $t = 0$. This gives us

$$x_1 = x_0 + v_0(\Delta t) + \frac{1}{2}a_0(\Delta t)^2 + \frac{1}{3!}j_0(\Delta t)^3 + \frac{1}{4!}s_0(\Delta t)^4 + \frac{1}{5!}c_0(\Delta t)^5 + O((\Delta t)^6)$$
and

$$v_1 = v_0 + a_0(\Delta t) + \frac{1}{2}j_0(\Delta t)^2 + \frac{1}{3!}s_0(\Delta t)^3 + \frac{1}{4!}c_0(\Delta t)^4 + p_0(\Delta t)^5 + O((\Delta t)^6)$$

where j_0 is commonly called the "jerk" (after a sudden change in acceleration), and s_0, c_0, and p_0 are called the "snap", "crackle", and "pop". [1]

[1] Piet Hut and Jun Makino, The Art of Computational Science

If we use the system of differential equations and equate terms, we have the requirements that

$$a_0 = f(x(0)) = f(x_0) = f_0 \tag{6}$$

$$j_0 = \frac{d}{dt}a(t)|_{t=0} = \frac{df(x)}{dx}|_{x_0} \cdot \frac{dx}{dt}|_{t=0} = f_0' v_0 \tag{7}$$

$$s_0 = \frac{d^2}{dt^2}a(t)|_{t=0} = f_0'' v_0^2 + f_0' f_0 \tag{8}$$

$$c_0 = f_0''' v_0^3 + 3 f_0'' f_0 v_0 + (f_0')^2 v_0 \tag{9}$$

$$p_0 = f_0^{[4]} v_0^4 + 6 f_0''' f_0 v_0^2 + 3 f_0'' f_0^2 + 5 f_0'' f_0' v_0^2 + (f_0')^2 f_0 \tag{10}$$

We will make use of these requirements extensively, below.

A Simple System

First, we note that, if we just try integrating $f(x_0) = f_0$, we obtain

$$x_1 = \frac{1}{2} f_0 (\Delta t)^2 + b(\Delta t) + c$$

$$v_1 = \frac{1}{2} f_0 (\Delta t) + b$$

With the initial conditions, we get

$$x_1 = x_0 + v_0(\Delta t) + \frac{1}{2} f_0 (\Delta t)^2$$

$$v_1 = v_0 + f_0(\Delta t).$$

This matches the first few terms of the Taylor series, and we see that it is first order in v and, ostensively as least, second order in x.

A Second Order Method

To get higher-order accuracy, we need different terms in the definition of x_1 and v_1. Runge-Kutta (RK) methods approximate the Taylor series by appropriate choices of when and where to evaluate the function $f(x(t))$. (see

more detail, below, in the General RK4 section)

For example, if we choose

$$x_1 = x_0 + v_0(\Delta t) + \frac{1}{2} f(x_0)(\Delta t)^2$$

$$v_1 = v_0 + f(x_0 + \frac{1}{2} v_0(\Delta t))(\Delta t)$$

we can see this is accurate through second order (in (Δt)) by matching the Taylor series. We do this as follows:

First notice that x_1 exactly matches the first three terms of the Taylor series for x_1, with the identification of a_0 with $f_0 = f(x_0)$. For v_1, we need to expand f about x_0. Thus, $f(x_0 + \frac{1}{2} v_0(\Delta t)) = f_0 + f_0'(\frac{1}{2} v_0(\Delta t)) + O((\Delta t)^2)$. Substituting this in gives

$$v_1 = v_0 + f_0(\Delta t) + \frac{1}{2} f_0' v_0 (\Delta t)^2 + O((\Delta t)^3).$$

Since $j_0 = f_0' v_0$, we see that v_1 also matches exactly the first three terms of the Taylor series. Thus, both r_1 and v_1 are second-order accurate. This is called an RK2 (second order Runge-Kutta) approximation. To implement this in code, one would do something like the following:

$$x_{\frac{1}{2}} = x_0 + \frac{1}{2} v_0(\Delta t)$$

$$v_{\frac{1}{2}} = v_0 + \frac{1}{2} a_0(\Delta t)$$

$$x_1 = r_0 + v_0(\Delta t) + \frac{1}{2} a_0(\Delta t)^2$$

$$v_1 = v_0 + a_{\frac{1}{2}}(\Delta t),$$

where $a_{\frac{1}{2}}$ is the function f evaluated at $x_{\frac{1}{2}}$.

A Fourth Order Method

For an approximation that is accurate through fourth order in (Δt), we need to estimate the acceleration at three values. Let

$k_0 = f_0 = f(x_0)$

$k_1 = f(x_0 + \frac{1}{2}v_0(\Delta t) + \frac{1}{8}k_0(\Delta t)^2)$

$k_2 = f(x_0 + v_0(\Delta t) + \frac{1}{2}k_1(\Delta t)^2)$

Then let

$x_1 = x_0 + v_0(\Delta t) + \frac{1}{6}(k_0 + 2k_1)(\Delta t)^2$ and

$v_1 = v_0 + \frac{1}{6}(k_0 + 4k_1 + k_2)(\Delta t)$.

To show that this has the desired accuracy, we expand k_1 and k_2 as

$k_1 = f_0 + f_0'(\frac{1}{2}v_0(\Delta t) + \frac{1}{8}f_0(\Delta t)^2) + \frac{f_0''}{2}(\frac{1}{2}v_0(\Delta t) + \frac{1}{8}f_0(\Delta t)^2)^2 + \frac{f_0'''}{6}(\frac{1}{2}v_0(\Delta t) + \frac{1}{8}f_0(\Delta t)^2)^3 + O((\Delta t)^4)$

$k_2 = f_0 + f_0'(v_0(\Delta t) + \frac{1}{2}k_1(\Delta t)^2) + \frac{f_0''}{2}(v_0(\Delta t) + \frac{1}{2}k_1(\Delta t)^2)^2 + \frac{f_0'''}{6}(v_0(\Delta t) + \frac{1}{2}k_1(\Delta t)^2)^3 + O((\Delta t)^4)$, or

$k_2 = f_0 + f_0'v_0(\Delta t) + \frac{1}{2}(f_0'k_1 + f_0''v_0^2)(\Delta t)^2 + (\frac{1}{2}f_0''v_0k_1 + \frac{1}{6}f_0'''v_0^3)(\Delta t)^3 + O((\Delta t)^4)$

Collecting terms for k_1, we obtain

$k_1 = f_0 + \frac{1}{2}f_0'v_0(\Delta t) + (\frac{1}{8}f_0f_0' + \frac{1}{8}f_0''v_0^2)(\Delta t)^2 + (\frac{1}{16}f_0''f_0v_0 + \frac{f_0'''}{48}v_0^3)(\Delta t)^3 + O((\Delta t)^4)$.

Substituting this into the expression for k_2 and collecting terms, we have

$k_2 = f_0 + f_0' v_0(\Delta t) + \frac{1}{2}(f_0' f_0 + f_0'' v_0^2)(\Delta t)^2 + (\frac{1}{4}(f_0')^2 v_0 + \frac{1}{2} f_0'' f_0 v_0 + \frac{1}{6} f_0''' v_0^3)(\Delta t)^3 + O((\Delta t)^4)$

Substituting the expression for k_1 into the definition of x_1 and collecting terms gives

$x_1 = x_0 + v_0(\Delta t) + \frac{1}{6}(f_0 + 2f_0)(\Delta t)^2 + \frac{1}{6} f_0' v_0 (\Delta t)^3 + \frac{1}{24}(f_0 f_0' + f_0'' v_0^2)(\Delta t)^4 + O((\Delta t)^5)$,

which simplifies to

$x_1 = x_0 + v_0(\Delta t) + \frac{1}{2} f_0 (\Delta t)^2 + \frac{1}{3!} f_0' v_0 (\Delta t)^3 + \frac{1}{4!}(f_0 f_0' + f_0'' v_0^2)(\Delta t)^4 + O((\Delta t)^5)$

Comparing with the Taylor series for x_1 shows us that they match exactly to the $O((\Delta t)^5)$ term, demonstrating that the expression for x_1 is accurate through order 4.

Now we must check v_1. Substituting the expressions for k_1 and k_2 into the definition of v_1 gives

$v_1 = v_0 + \frac{1}{6}(\Delta t) \cdot f_0 + \frac{2}{3}(\Delta t)(f_0 + \frac{1}{2} f_0' v_0 (\Delta t) + \frac{1}{8} f_0 f_0' (\Delta t)^2 + \frac{1}{8} f_0'' v_0^2 (\Delta t)^2 + \frac{1}{16} f_0'' f_0 v_0 (\Delta t)^3 + \frac{1}{48} f_0''' v_0^3 (\Delta t)^3) + \frac{(\Delta t)}{6}(f_0 + f_0' v_0 (\Delta t) + \frac{1}{2} f_0' f_0 (\Delta t)^2 + \frac{1}{2} f_0'' v_0^2 (\Delta t)^2 + \frac{1}{4}(f_0')^2 v_0 (\Delta t)^3 + \frac{1}{2} f_0'' f_0 v_0 (\Delta t)^3 + \frac{1}{24} f_0''' v_0^3 (\Delta t)^3) + O((\Delta t)^5)$, or

$v_1 = v_0 + (\Delta t)(\frac{1}{6} f_0 + \frac{2}{3} f_0 + \frac{1}{6} f_0) + (\Delta t)^2(\frac{1}{3} f_0' v_0 + \frac{1}{6} f_0' v_0) + (\Delta t)^3(\frac{1}{12} f_0 f_0' + \frac{1}{12} f_0'' v_0^2 + \frac{1}{12} f_0' f_0 + \frac{1}{12} f_0'' v_0^2) + (\Delta t)^4(\frac{1}{24} f_0'' f_0 v_0 + \frac{1}{72} f_0''' v_0^3 + \frac{1}{24}(f_0')^2 v_0 + \frac{1}{12} f_0 f_0'' v_0 + \frac{1}{144} f_0''' v_0^3) + O((\Delta t)^5)$.

Simplifying, we have

$v_1 = v_0 + f_0(\Delta t) + \frac{1}{2} f_0' v_0 (\Delta t)^2 + \frac{1}{3!}(f_0 f_0' + f_0'' v_0^2)(\Delta t)^3 + \frac{1}{4!}[(f_0')^2 v_0 + 3 f_0'' f_0 v_0 +$

$f_0'''v_0^3](\Delta t)^4 + O((\Delta t)^5)$.

Again, comparing with the Taylor series for v_1, and using the conditions on the terms in the series, we see that we have an exact match up to terms of order 5, showing that the expression for v_1 is also accurate through order 4. This is an RK4 code.

General RK4 Algorithms

There is, in fact, a family of RK4 codes. One starts with the idea that one should estimate the acceleration ($f(x)$) at three positions, iterating to produce the values of k below. Aside from the initial choice of x_0, though, we express the other two positions in terms of linear combinations of previous ks and powers of (Δt). Let

$k_0 = f_0 = f(x_0)$

$k_1 = f(x_0 + \sigma v_0(\Delta t) + \xi k_0(\Delta t)^2)$

$k_2 = f(x_0 + v_0(\Delta t) + (\eta k_0 + \kappa k_1)(\Delta t)^2)$

Then let

$x_1 = x_0 + v_0(\Delta t) + (\alpha_1 k_0 + \alpha_2 k_1 + \alpha_3 k_2)(\Delta t)^2$ and

$v_1 = v_0 + (\beta_1 k_0 + \beta_2 k_1 + \beta_3 k_2)(\Delta t)$.

One proceeds as we did in the specific RK4 example above, expanding k_1 and k_2 as Taylor series in (Δt), then expanding the expressions for x_1 and v_1 in (Δt). Finally, one collects all like powers of (Δt) and matches each power with the general Taylor series expression for x_1 and v_1 above (the $v_0, a_0, j_0, s_0, c_0, p_0$ terms, and more if higher order algorithms are being derived). This matching yields a set of equations in the constants used in the definitions of k_0, k_1, k_2 as well as the constants $\alpha_1, \alpha_2, \alpha_3, \beta_1, \beta_2, \beta_3$ used in the combinations to define x_1 and v_1. Typically, there is at least one free parameter when this system of linear equations is solved, which yields different particular versions of RK4.

Reference Orbits

Reference Ellipse

In order to test our numerical approximation schemes, we chose a reference orbit, an ellipse with the semi-major axis twice as long as the semi-major axis. This was picked to have a region where the gravitational attraction was much stronger, as well as the region at the other end of the ellipse, where gravity was much weaker, to ensure that the approximations were tested fairly.

We picked an orbital period of $10^7 s$, which is about $\frac{1}{3}$ of a year (the nearest even power of 10) and a semi-major axis equal to $a = 150 \cdot 10^9 m$, 1 AU (Astronomical Unit, equal to the average distance of the earth to the sun). Using these values (and $a = 2b$) and the eccentricity $e = \frac{\sqrt{3}}{2}$, we obtain the equation of the ellipse in polar coordinates to be
$r(\theta) = \frac{a}{4} \cdot \frac{1}{1 - e \cdot cos(\theta)}$.
We call this ellipse the reference orbit.

The reference orbit obeys Kepler's Laws, which (among other things) says that

$$\frac{T^2}{4\pi^2} = \frac{a^3}{GM},$$

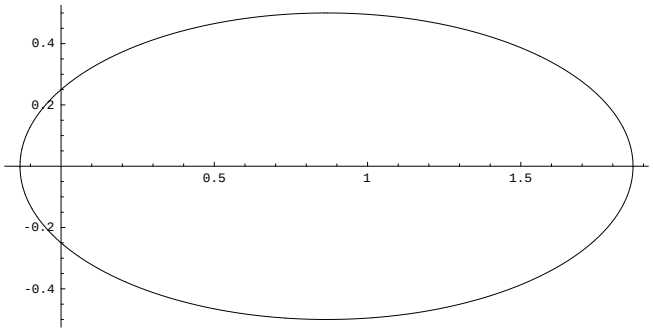
where T is the period of the orbit and a is the semi-major axis. Plugging in the specified values gives us a value of $GM = 1.35\pi^2 \cdot 10^{20}$ (mks units). This gives a mass of about $1.9976 \cdot 10^{31} kg$, which is about 10 solar masses.

Since the central mass M is at one focus of the ellipse, we can determine other parameters of the orbit. (We will use the terms aphelion and perihelion rather than the less familiar apastron and periastron.) The aphelion distance is $r_0 = a \cdot (1 + e)$ and the perihelion distance $r_1 = a \cdot (1 - e)$. We will normally choose a test trajectory to start at aphelion, where the speed is $v_{ap} = \frac{\sqrt{2GM} \cdot \sqrt{\frac{r_1}{r_0}}}{\sqrt{r_0 + r_1}} \approx 25253.6\ m/s$. Finally, the specific energy (or energy for a unit mass) of a spacecraft in this orbit is $\frac{1}{2}v^2 - \frac{GM}{r} \approx$

$-4.4413219804902 \cdot 10^9\ J/kg$. This is a constant of the orbit.

The figure shows the reference orbit, scaled so that $a = 2$ and $b = 1$. Note that the attractive mass M is at the left-hand focus, which is chosen to be the coordinate origin.

Figure 7: Reference Ellipse



Reference Hyperbola

In order to better check our code, we also created a reference hyperbola, to test slingshot predictions. We chose eccentricity $e = \sqrt{2}$ (greater than 1, as required for a hyperbola) closest approach distance, perihelion $r_1 = 10^{10}\ m$, $M = 2 \cdot 10^{30}\ kg$ (about the mass of the sun), which gave us a hyperbola with asymptotes at 45°. The equation is

$$r(\theta) = \frac{r(1+e)}{1 - e \cdot \cos\theta}$$

Notice, in the graph, that the sun is at the focus of the hyperbola, and the closest approach distance is, indeed, 10 million kilometers.
We have the relation

$$\frac{l^2}{GM} = r_1 \cdot (1+e)$$

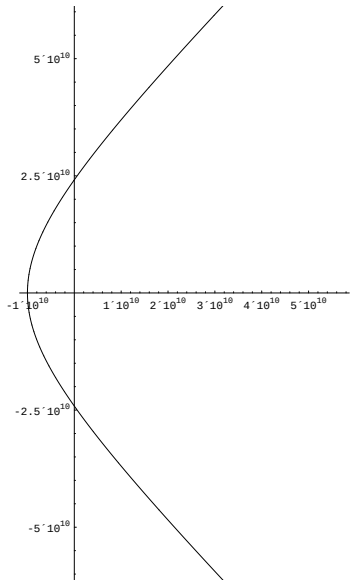
where $l^2 = (r_1 v_1)^2$, and v_1 is the speed at perihelion. This gives us

$$v_1^2 = \frac{(1+e)GM}{r_1}$$

which is about 179459 m/s.
We also know that the constant, total (specific) energy is
$E = \frac{1}{2}v^2 - \frac{GM}{r}$,

which we can compute at perihelion to be about $2.7628 \cdot 10^9$ J.

Figure 8: Reference Hyperbola

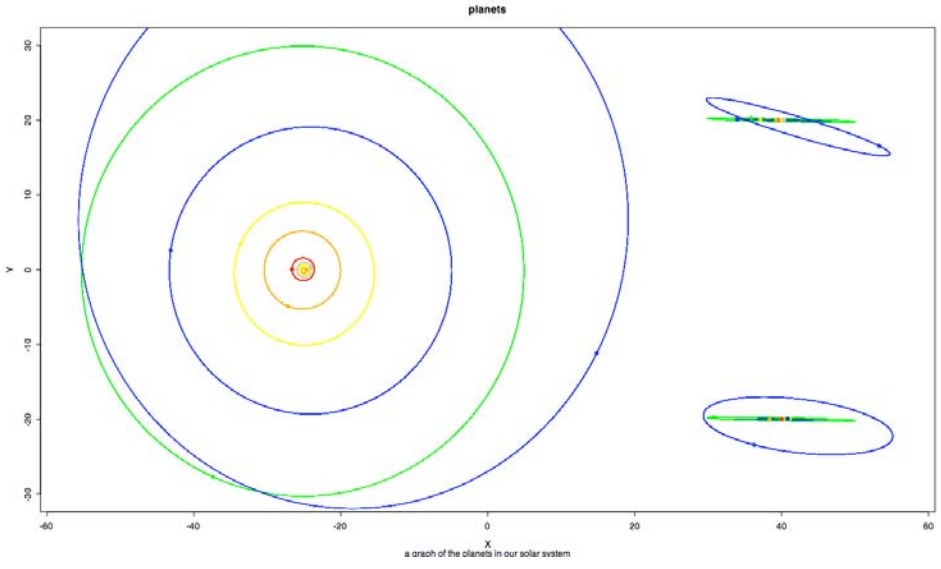


The nice thing about a reference hyperbola is that we can use it to test our slingshot theory against our numerical methods. We know the exact values to use to produce (in theory) the reference hyperbola where the sun is fixed. Now we can add any velocity vector to both the Sun and the spacecraft starting conditions, and the spacecraft should still "collide" with the sun. By varying this velocity vector, we can test to see if the spacecraft indeed gains speed (velocity vector going towards the right) or loses speed (velocity towards the left) and everything in between.

Functions and Structure of Delta

Our program, Delta, combines various calculations to optimize the trajectories of the spacecraft. First, we calculate the positions of the planets on an ellipse. The positions are accurate to the specified day. We initialize an array of spacecraft 200 km above Earth with a set initial velocity and launch angle. The main program loop then cycles through each spacecraft and planet and calculates how the planets' gravitational attractions affect the spacecraft. To decide how long the next timestep should be, we measure the distance between the planet we are trying to reach and the spacecraft that is closest to any planet. Delta scales the system and writes the color and identification information into an image file array. These images are then displayed in the browser. By using ISMAP, we can click on a path and Delta will return which spacecraft or planet was at that position. Finally, using a recenter function, the spacecraft array is re-defined so that the initial conditions are closer to those of the optimal spacecraft, and the entire program runs again.

The picture below is an example of how the program calculates the initial position of each planet. The equation is for Earth at 00:00 in the morning of January 1st, 2004. Note that this example only gives us the answer for one planet at one specific time. In the program, we actually solve this equation numerous times, using a for loop to find the planet' positions at each time. This creates the illusion that the planets are moving. Each time we find an X and a Y, we put them together to create the result below.



The array of spacecraft is launched from 200 km above the Earth's surface. Each spacecraft is placed INITIALLAUNCHEDGE meters apart and launched at LEAVEORBITMS meters/second at angle launchangle1 away from Earth in the earth/moon rotating frame. Since the velocity vector of each spacecraft depends on the velocity of Earth, the ellipse function is called twice in order to deduce the Earth's velocity vector. We add the vector to the velocity of each spacecraft and it results in an array of spacecraft with the same velocity but slightly different positions.

The main calculations of the program take place in a loop that cycles through each spacecraft and planet. The gravitational effects of each planet are calculated and used to modify the velocity vectors of the spacecraft. Delta updates spacecraft[i].closestpos, the smallest distance between the spacecraft and the target planet. As Delta goes through all the spacecraft, it keeps track of the smallest distance between a spacecraft and any planet, mindist, which will be used later. Delta also checks each spacecraft to make sure it has not crashed into a planet. If it has, Delta no longer calculates its path. On the first calculation after the start of every other new day, Delta prints information. This information includes mindist, the day, and the distance between the target and the closest spacecraft. Finally, the spacecraft are moved dt seconds in the direction of the velocity vector.

Delta uses mindist to determine how many seconds the program will calculate next. When mindist is small, Delta calcualtes a small dt so as to preserve accuracy. When all spacecraft are far away from planets, dt is larger. It generally rises to 500 second intervals. For this reason, we limit the size of dt to a pre-set amount.

For each timestep, Delta also creates pictures. For a system that has only one or two principal bodies, we view the orbits in a frame where the principal bodies appear stationary. Each picture is described in terms of vectors (which stabilize the picture to make the different planets stationary) and a scale factor. Delta writes onto imagedata1 (or imagedata2, 3, 4, 5, 6, or 7) files with the RGB color and also fills the last bit with either the number of the spacecraft (+10) or the number of the planet. Thus, each object had a unique ID number.

The pictures are displayed as ISMAPs in the browser. ISMAP allows an image to be clicked on as a link, and it returns the coordinates of the spot on which the user clicked. The program goes back into the corresponding imagedata array and returns the identification of the object that was last written on that pixel.

Using closestpos data, the ISMAP identification system, or other, similar methods, the user can go back and slightly change initial conditions to refine where the spacecraft go. (We use a large number of spacecraft so that we can test any number of variations at once.) Recenter is one way of refining the simulation. It centers the spacecraft array on the initial position of the chosen spacecraft and also decreases the distance between the spacecraft. This is very effective in allowing Delta to move all the spacecraft closer to the goal.

Delta on the Cluster

Given the large amounts of computation involved to execute and plan low energy orbital paths, Delta was separated into pieces that can be run in parallel. It was executed on the cluster *Dax*, a three-node system which uses Windows 2000 network commands and a control program.

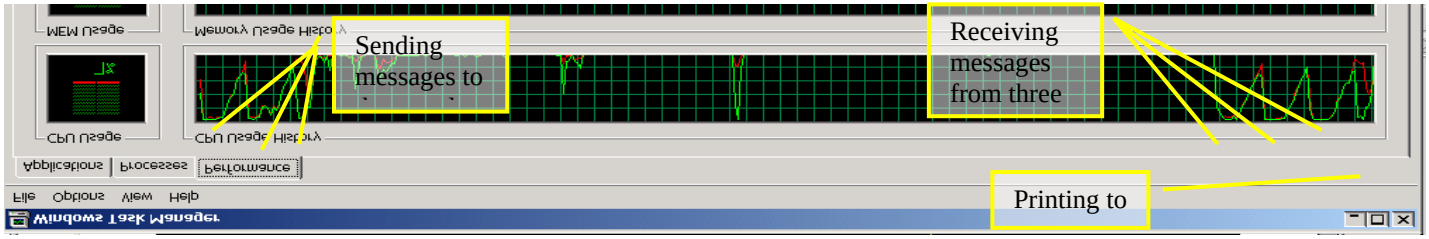
The orbits function, which calculates spacecraft ID numbers, takes as parameters the upper and lower ranges of the ID numbers. On a single computer, this range goes from 0 to the total number of spacecraft. When executed in parallel, the cluster program calculates how many spacecraft each node should process, given each node's relative speed. Each node calls the orbits function for the given range of spacecraft. Although there are calculations common to each node (for example, each node calculates the positions of the planets at each timestep and depicts the position of objects in multiple pictures), the total calculation time for the cluster was about three times less than the total calculation time on a single computer. The dramatic decrease in calculation time allowed us to calculate the paths of spacecraft to distant solar bodies, such as Pluto. Starting from Earth, using Jupiter for a slingshot, and ending at Pluto took 12 to 20 minutes on the cluster, depending on the maximum timestep allowed. The same calculation on a single computer took 30 to 60 minutes. As this calculation requires manual optimization, the time decrease is crucial.

However, allocating the spacecraft among different nodes creates some instability in the computation. After the entire array of spacecraft is divided among the nodes, each node executes the calculations individually and disregards the spacecraft it has not been assigned. Delta cycles through each spacecraft and planet to find the spacecraft closest to a planet, and then it uses this distance to decide an appropriate timestep. When the closest spacecraft is very near a planet, Delta uses a smaller timestep in order ensure accuracy. As the spacecraft move away from the planets, the timestep will increase until it reaches the maximum pre-defined limit. However, when the spacecraft are divided between nodes, some nodes will calculate spacecraft passing farther away from planets and, therefore, will calculate fewer times. These spacecraft would have been calculated with a higher degree of accuracy had they all been on one computer. Although the spacecraft's

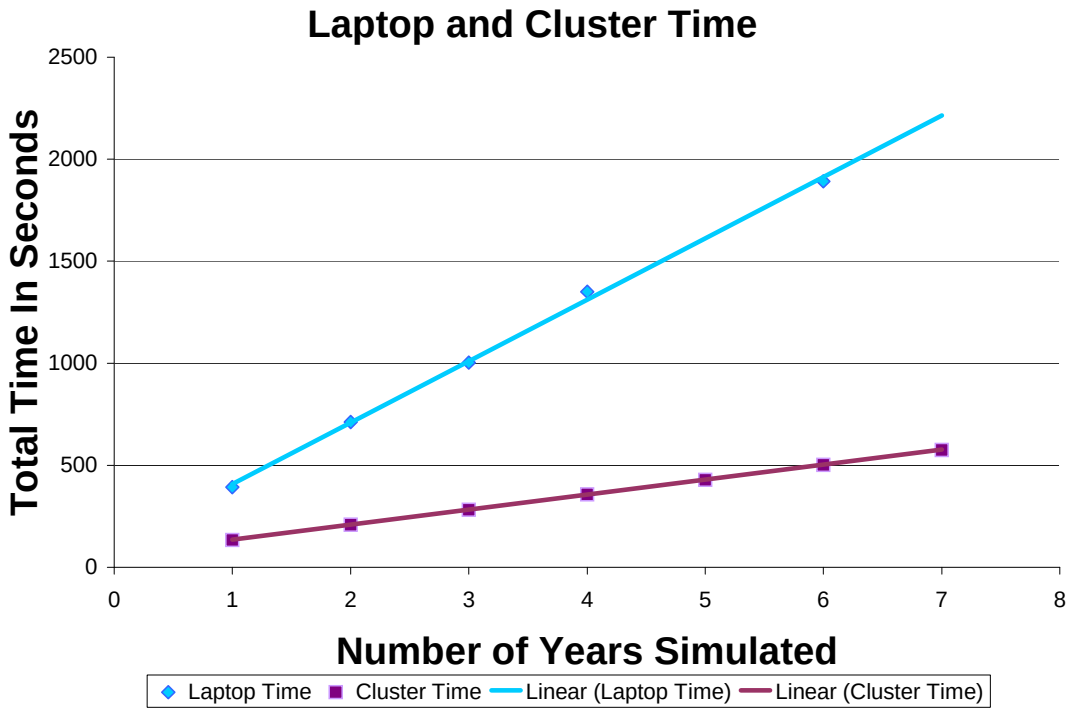
positions in the array are distributed fairly evenly among the nodes, the change in accuracy had a huge effect on a simulation of a spacecraft reaching to Pluto. Unfortunately, because of this, the recenter function resulted in further instabilities, and it could not be used.

Running the simulation on the cluster was significantly faster than running it on a single computer. We continued to use standard Newtonian physics to calculate both the positions of the planets and the spacecraft.

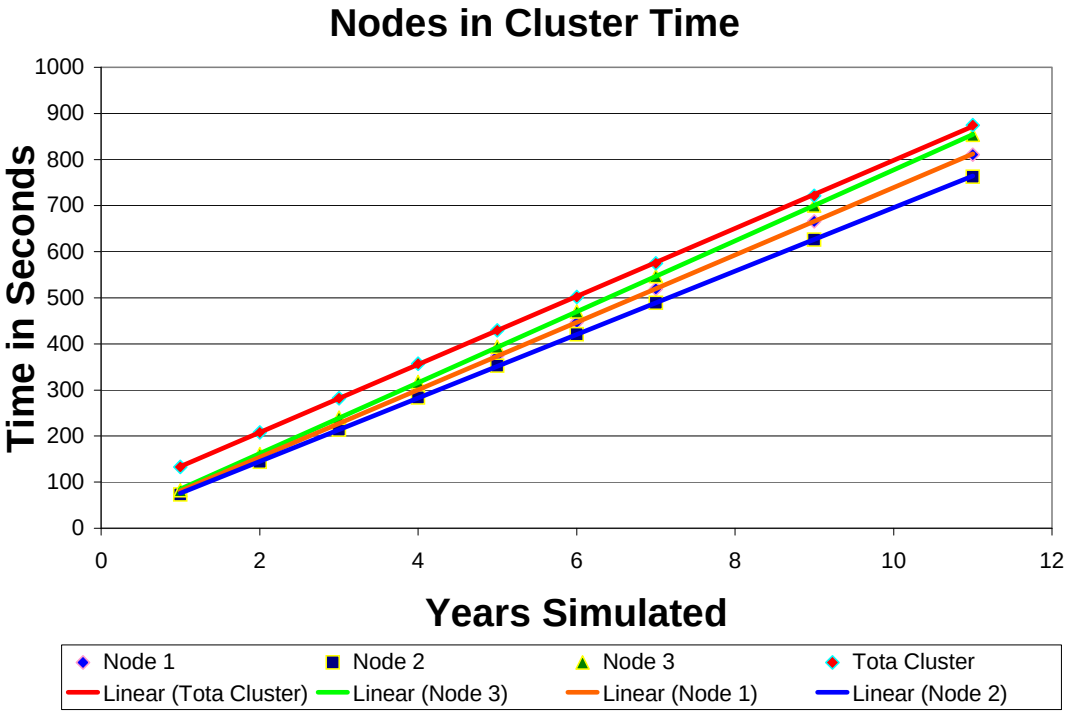
This is a page freeze of task manager during a three-year simulation.



The graph below compares the run times of the program on the cluster and on a laptop. The laptop takes more than three times the time to run. In part, this may be because the laptop is 800 MHz and the cluster's three nodes are 800 MHz, 1 GHz, and 1 GHz. Also, the laptop had other processes running at the same time as the program, such as a browser. Even so, the program runs much faster on the cluster.



This graph shows the relative speeds of each node in the cluster as compared to the total time. When there are fewer calculations, the nodes run for the same amount of time. With more calculations, however, small imperfections in the weighting of the nodes make the run times different. Interestingly, the difference in the finish times of the nodes makes the total run time closer to the slowest node's run time, since the master computer has more time to process the results of the individual nodes.



When we divided the code so it could run on the cluster, some imperfections in the efficiency of Delta were made more obvious, since inefficient calculations are repeated on a cluster and therefore have a greater effect on the run time. For example, the calculations of the movements of the objects, the calculations for tracking close objects, and the writing to picture files are all significant calculations cannot be removed without loss of functionality, but they are often performed on multiple nodes and could be simplified. Writing to picture files was the most inefficient of the three calculations, since it had a circle function for drawing round planets. This function was called at every timestep on every node for every planet in every picture. Fixing this inefficiency decreased the run time by more than a factor of eight. Running Delta on the cluster thus helped us to increase the efficiency of our code.

Results

Comparison of Approximation Methods

In order to test the effectiveness of each method, Delta was set to calculate six spacecraft. All the spacecraft started at the aphelion, ((1.+sqrt(3.)/2.)* 150e9, 0), and had initial velocity (0, sqrt(2.7) *PI*1e10* sqrt((2.-sqrt(3.))/(2.+sqrt(3.)))/ sqrt(2.* 150e9)) around the Sun. No other planets were involved. If the method used was accurate, all of the spacecraft would travel in an ellipse and return to exactly their starting position at the aphelion. Therefore, we were able to measure the ending distance from the aphelion to determine the accuracy of each method.

We used four different numerical methods and one standard:

Method 1: Newton

Method 2: Runge-Kutta 2 (RK2)

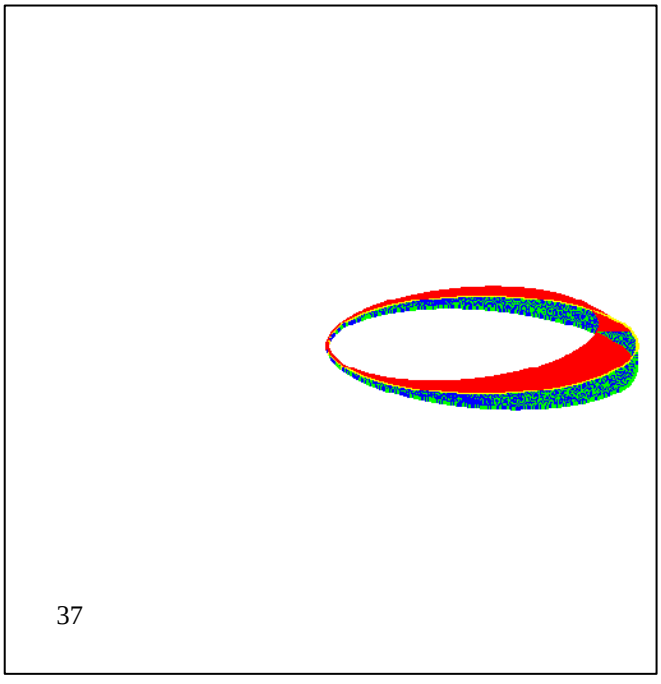
Method 3: Runge Kutta 4 (RK4A)

Method 4: Runge Kutta 4 (RK4B)

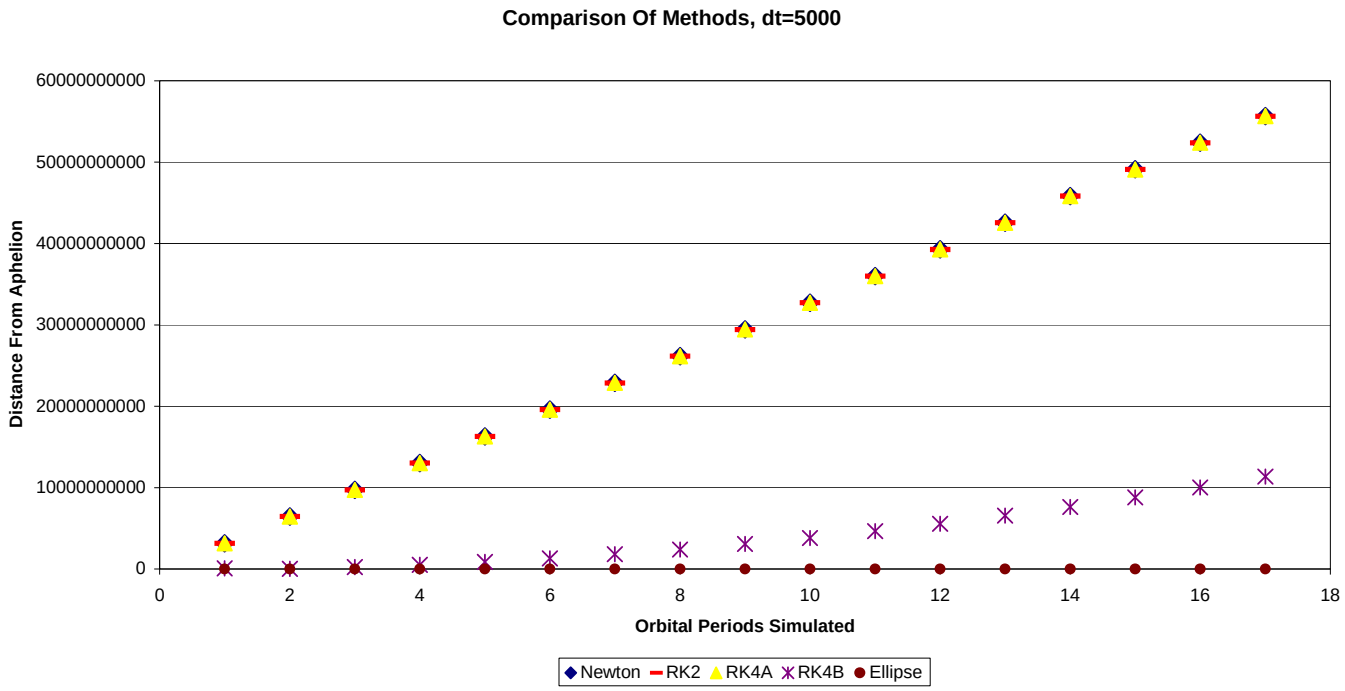
Method 5: Standard Ellipse

The graphs of the distances from the aphelion are organized into two categories: the accuracy of each method at different timesteps and the effect of the timestep on specific methods' performances.

This picture shows the setup of the accuracy test in the rotating Sun-Aphelion frame (the Sun is in the middle). The green/blue orbits are the RK2 and RK4A methods, the red orbit is an inaccurate method that we decided to not use, and the yellow orbit is RK4B (it is difficult to see because it diverges very little). All of the spacecraft start at the right edge of the ellipse. Over the course of the 20-orbital-period simulation, RK2 and RK4A gain a small amount of energy. RK4B does not gain or lose a significant amount of energy.

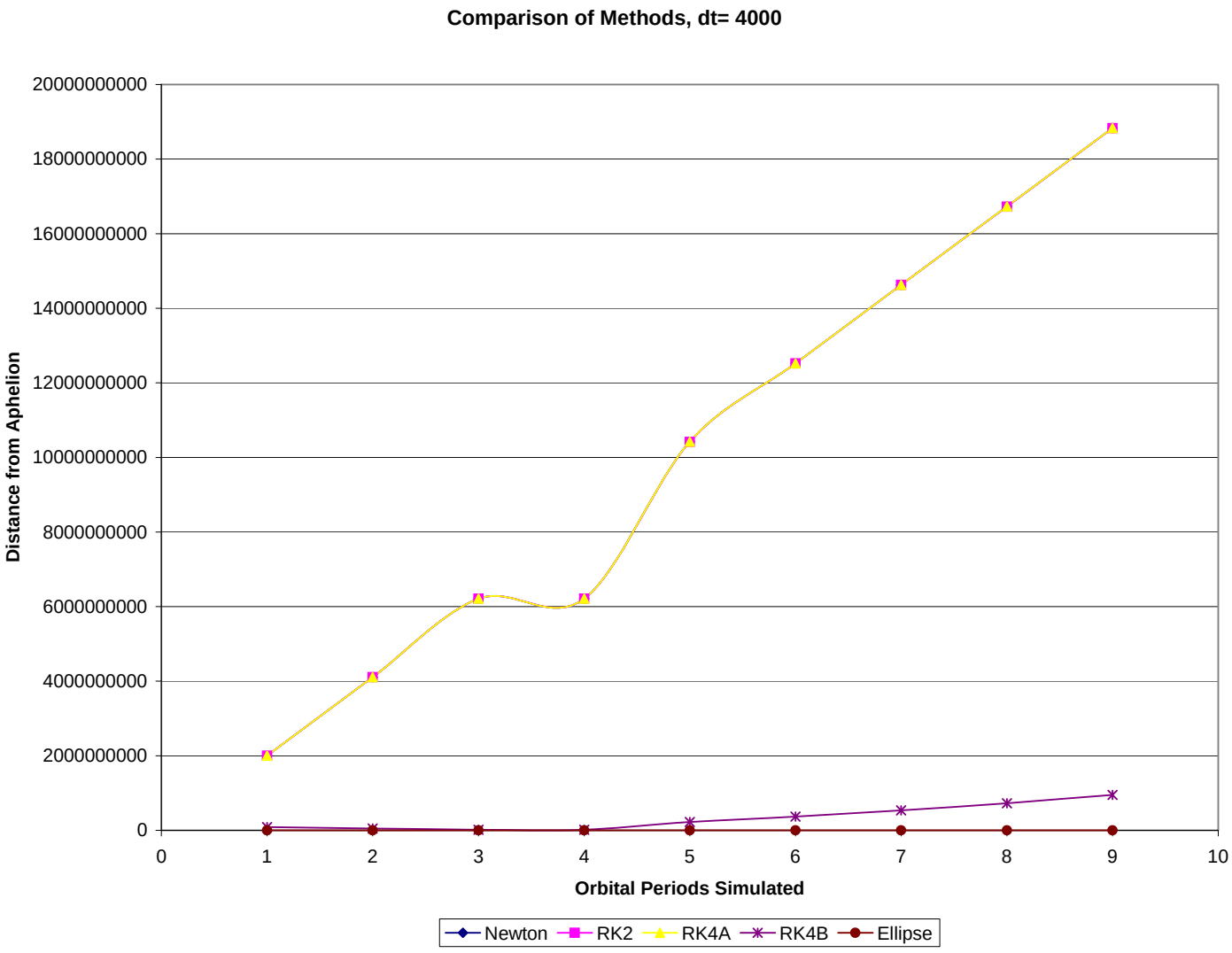


Timestep 5000



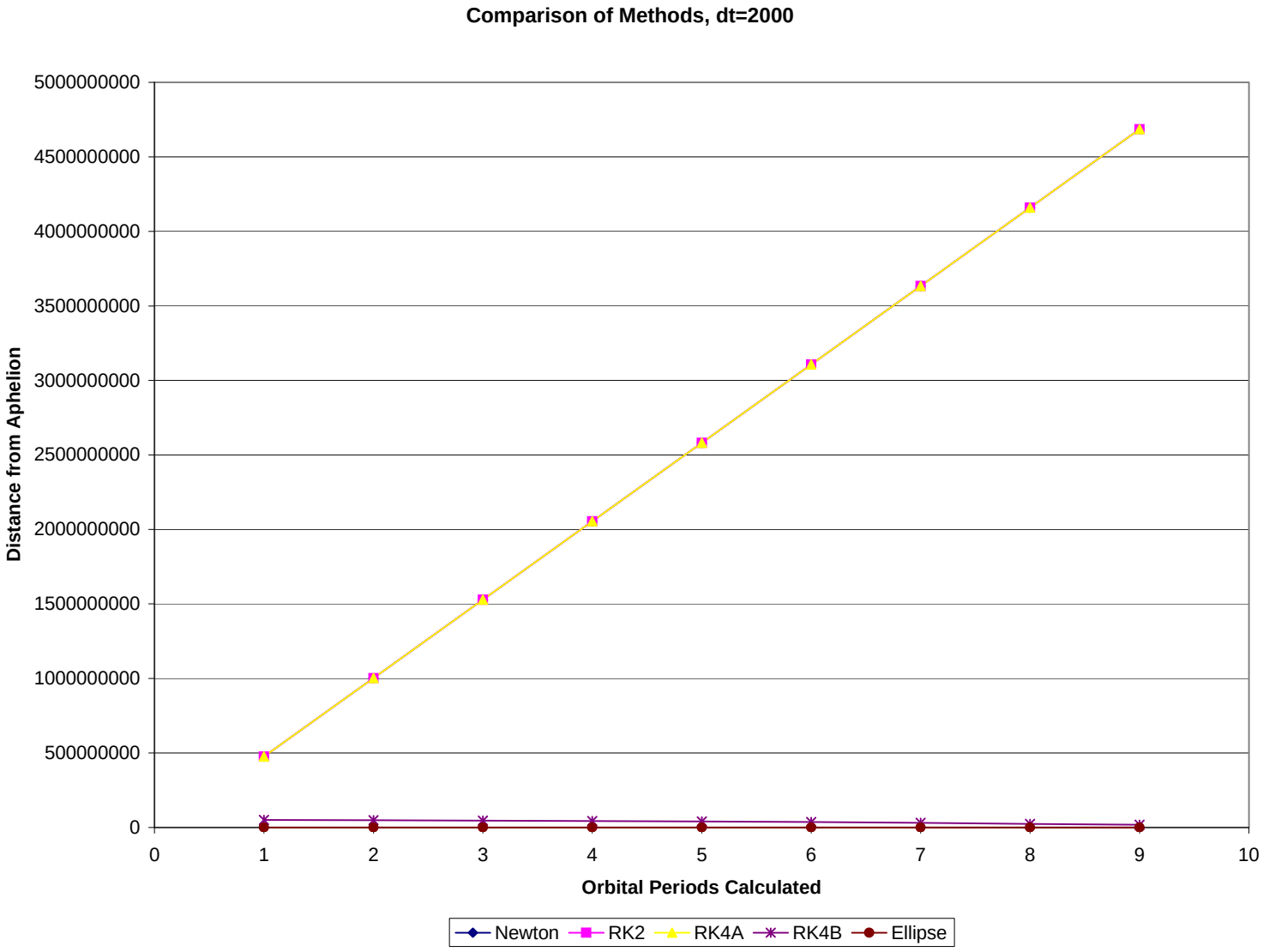
This shows the divergence of each method as a function of the number of orbital periods. RK4B is the most accurate numerical method. Also, the other methods diverge linearly with the number of orbits.

Timestep 4000

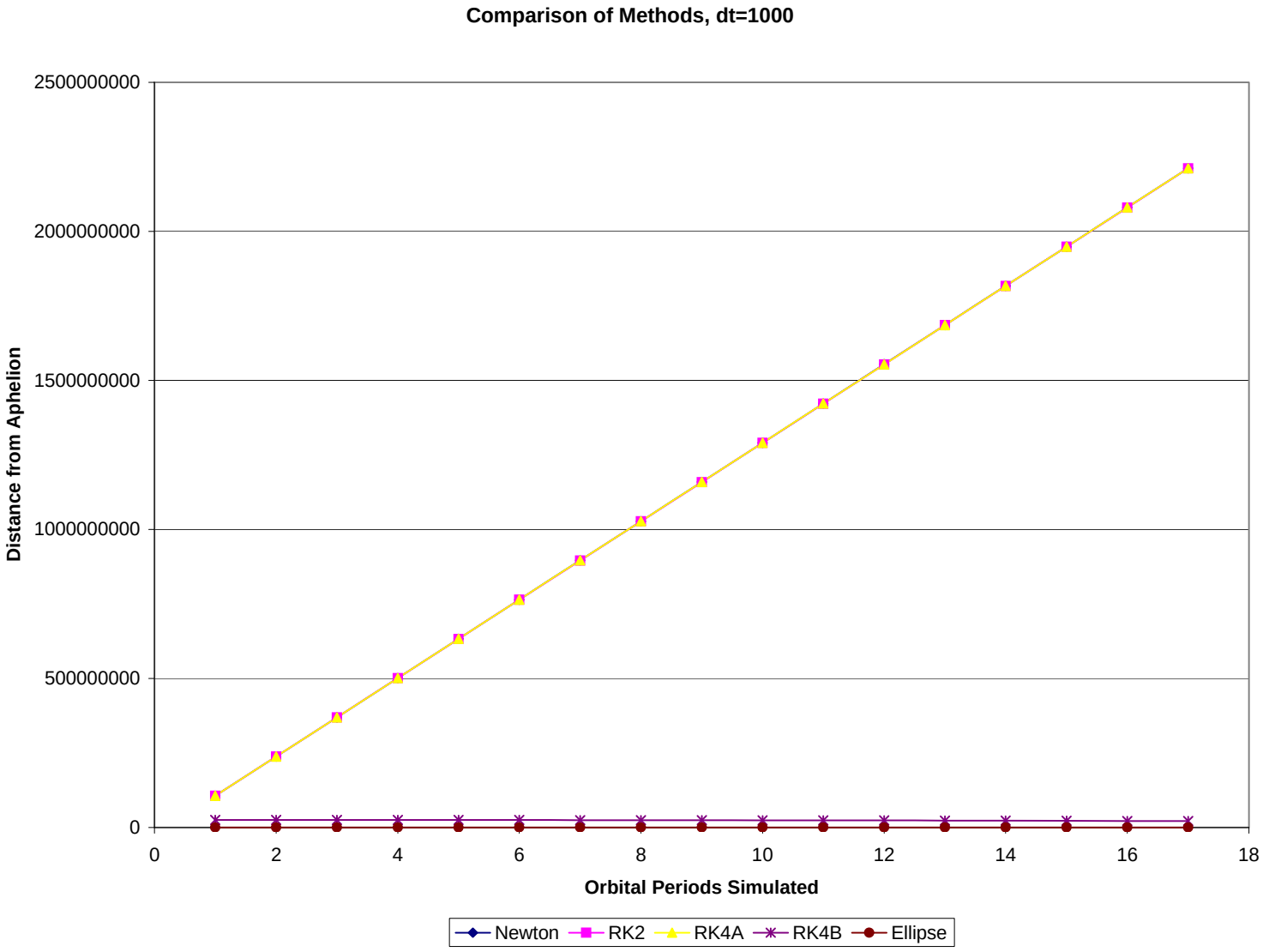


Notice that the Newton, RK2, and RK4A methods exhibit some oscillation at this timestep. This may be because the timestep is fairly long.

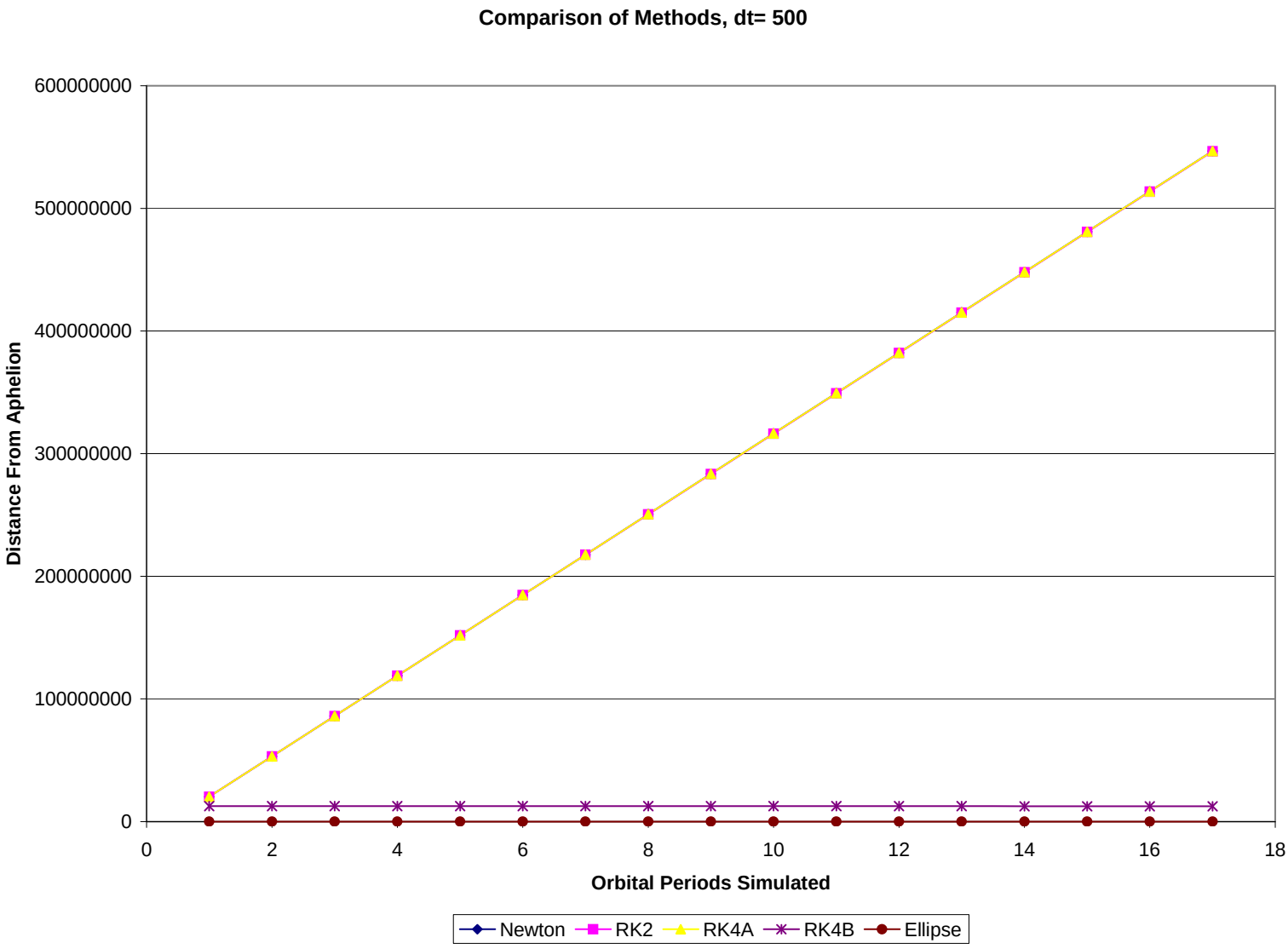
Timestep 2000



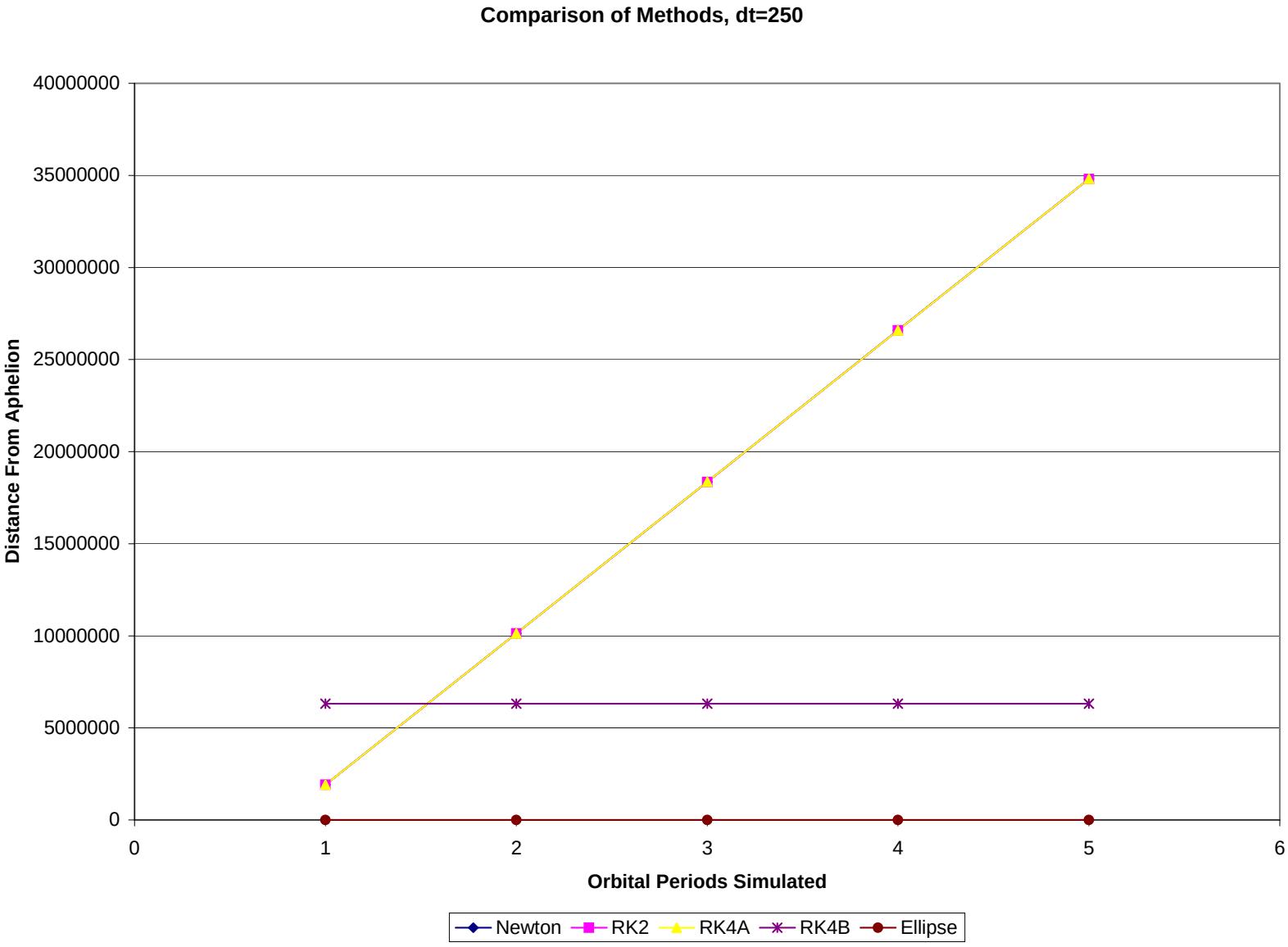
Timestep 1000



Timestep 500



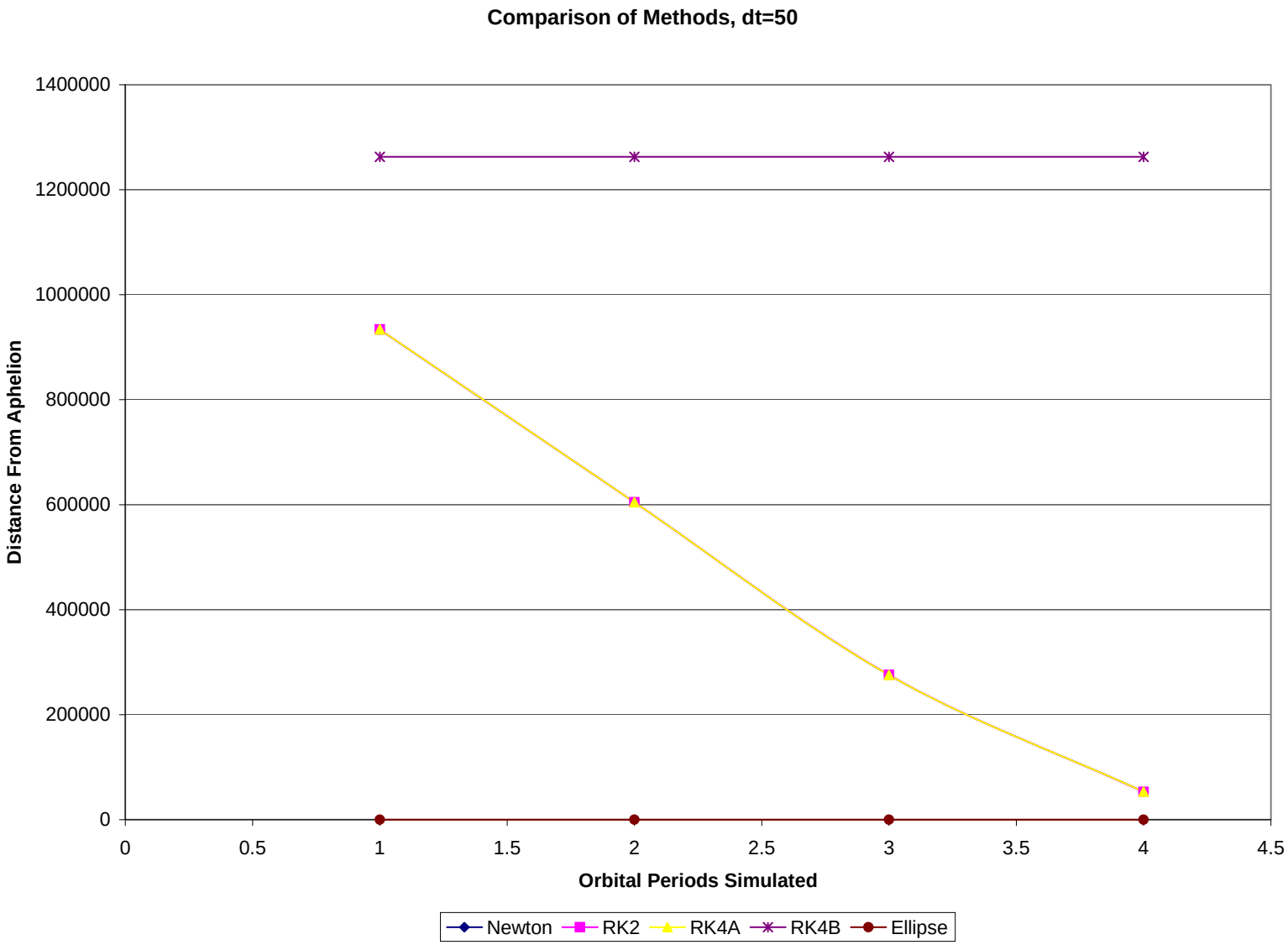
Timestep 250



The interesting feature of this graph lies in the fact that the Newton, RK2, and RK4A methods start out as being more accurate than RK4B, and then they diverge and become less accurate.

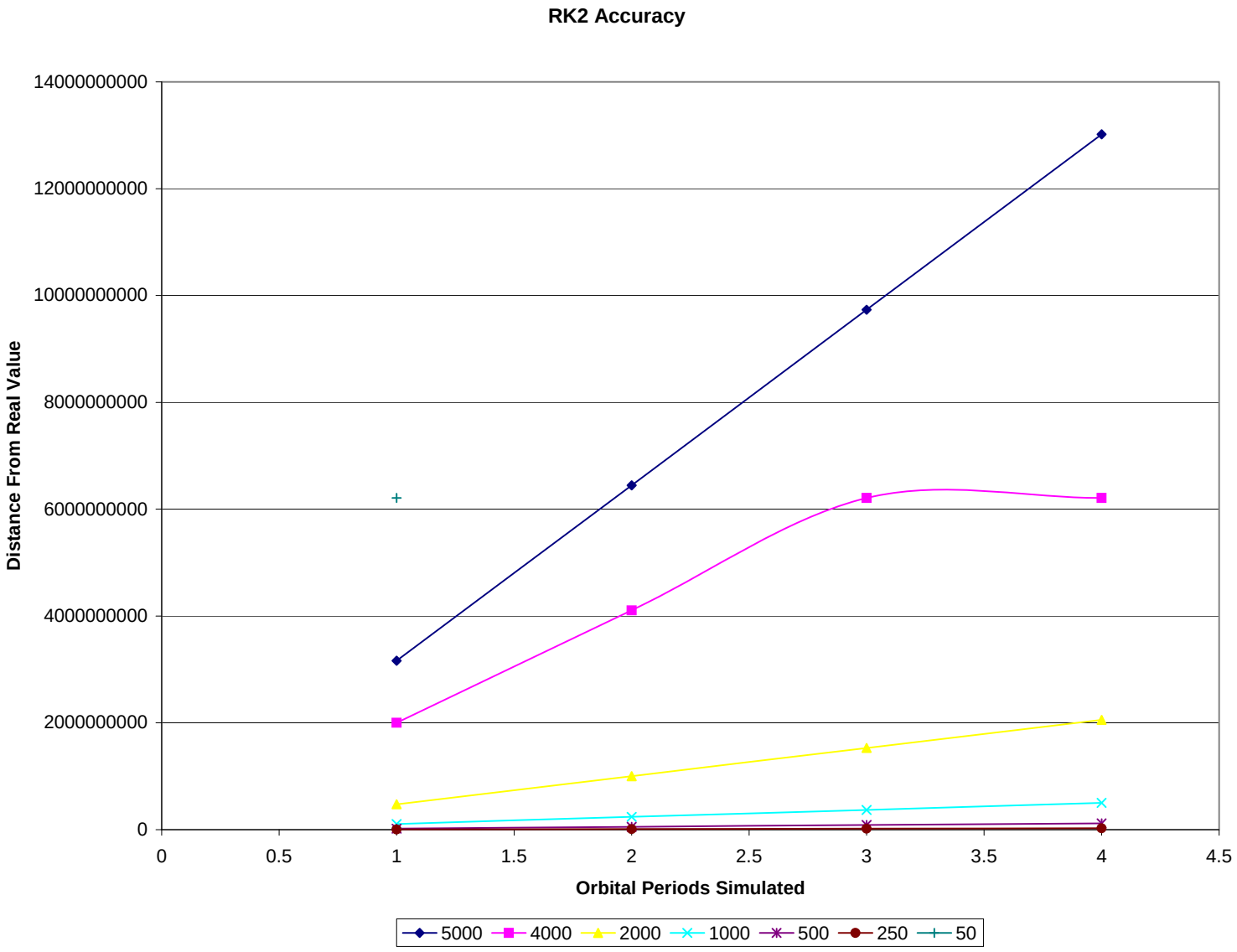
.

Timestep 50



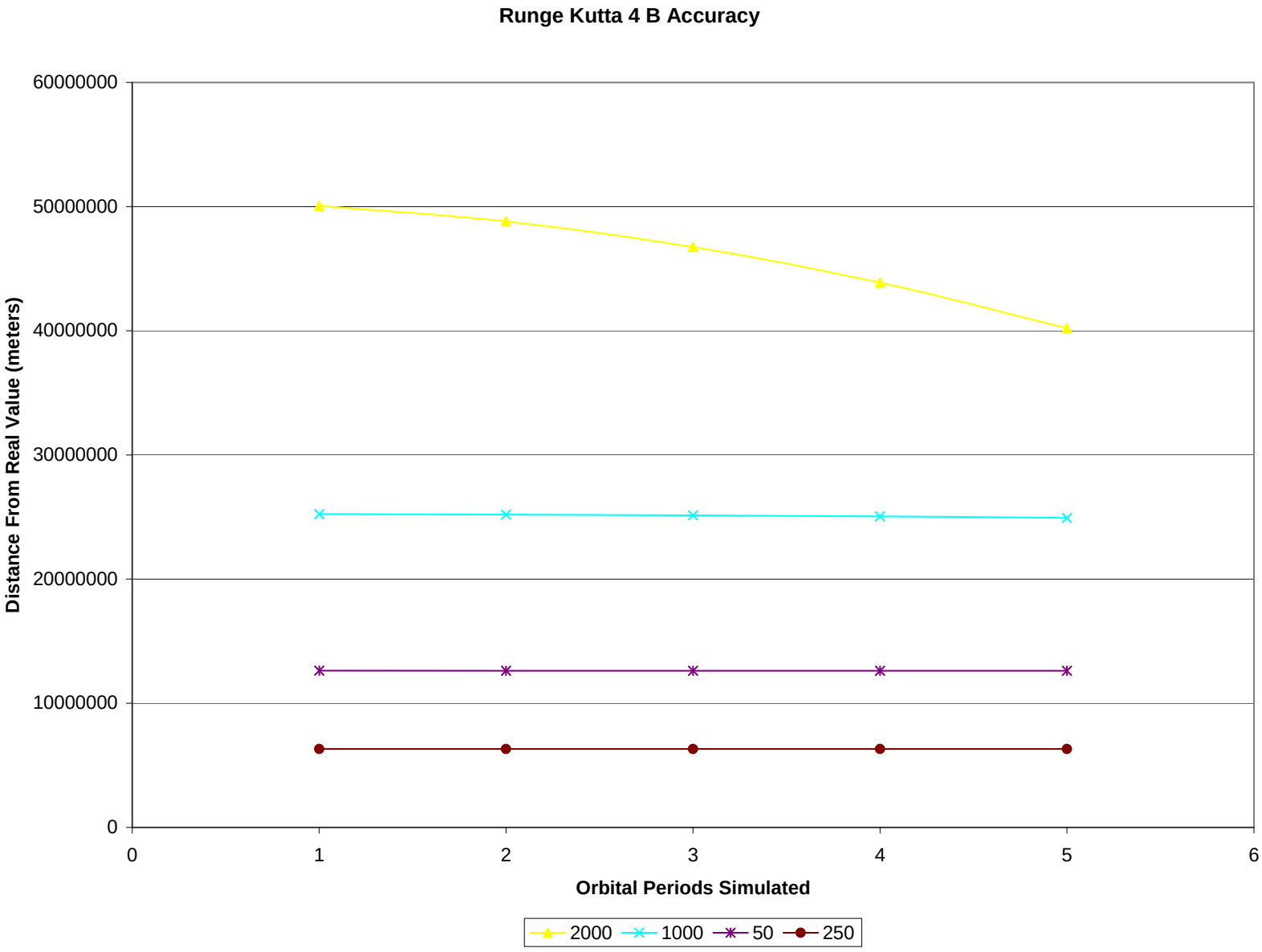
The small timestep has an interesting effect on the accuracy of the methods, making the Newton, RK2, and RK4A methods increasingly accurate.

RK2

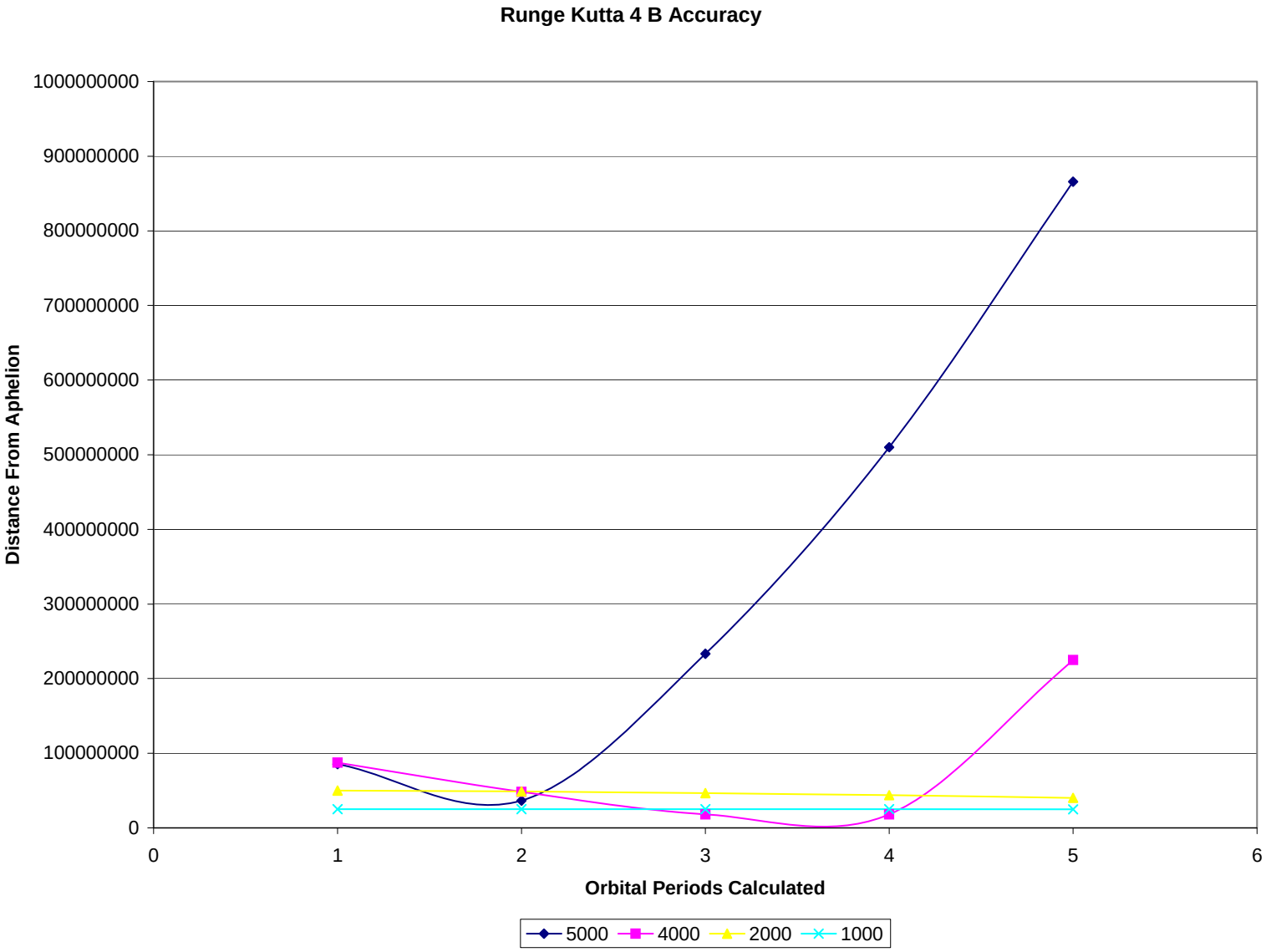


RK2, a second order equation, has increased accuracy for linear timestep decreases.

RK4B (Lower Timesteps)



RK4B (Higher Timesteps)

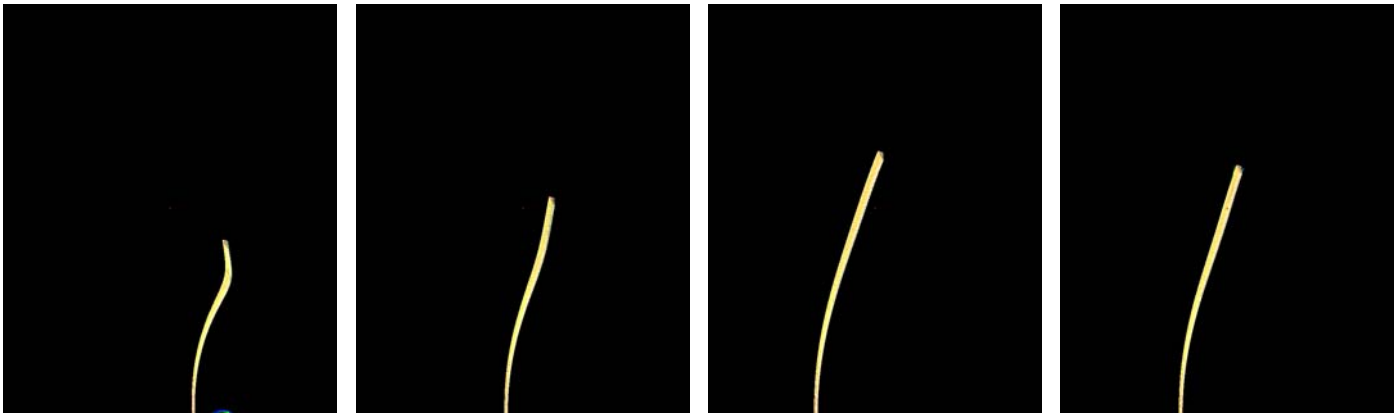


The 5000 and 4000 timesteps both show similar oscillatory behavior for RK4B as is exhibited by Newton, RK2, and RK4A methods in the dt=4000 graph.

Path to Mars

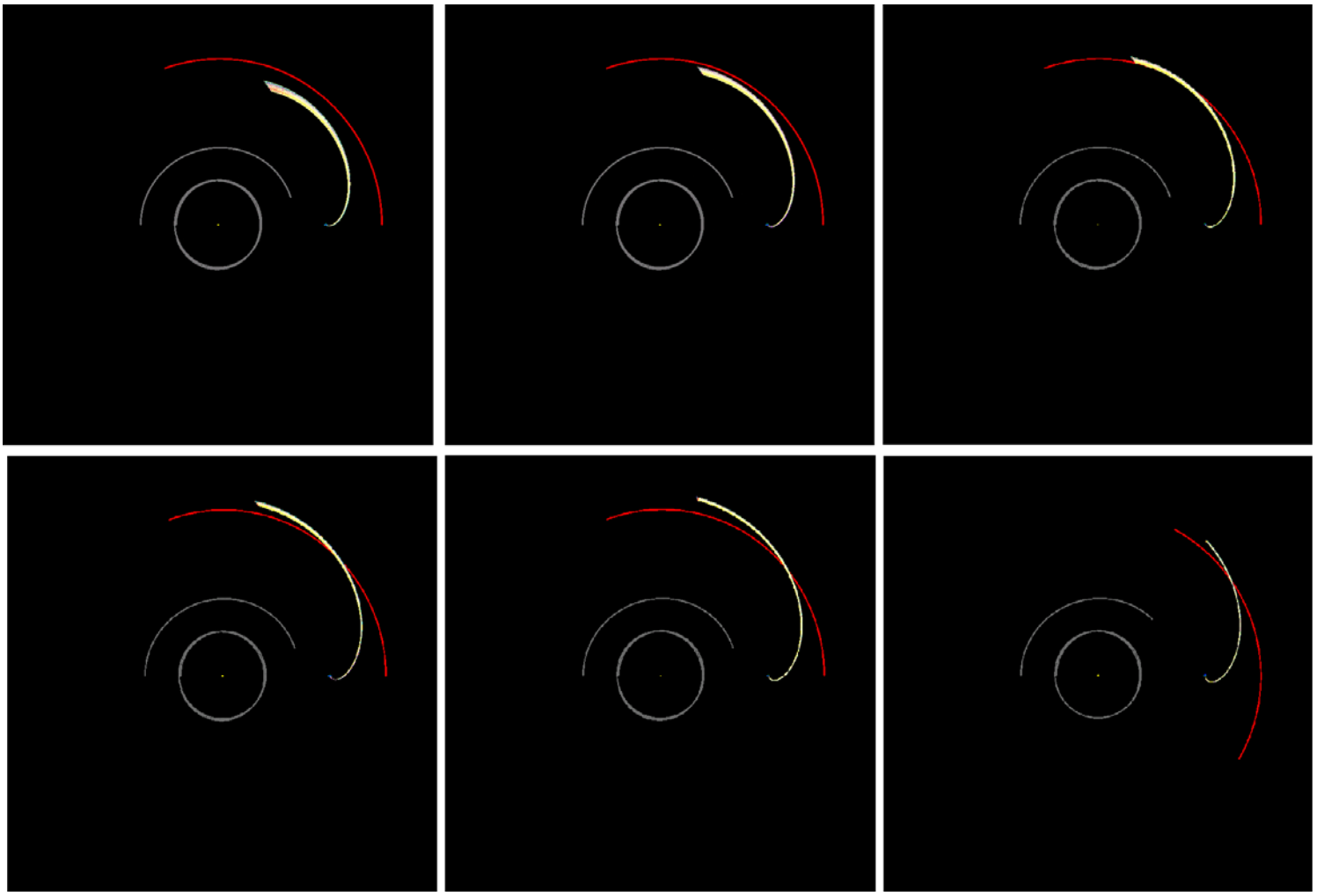
By using a gravity assist off of the moon and recentering the array of spacecraft on the optimal initial conditions, Delta was able to send multiple spacecraft to Mars.

There are three ways to optimize the way the spacecraft are launched in order to get them to Mars. Below is the first.



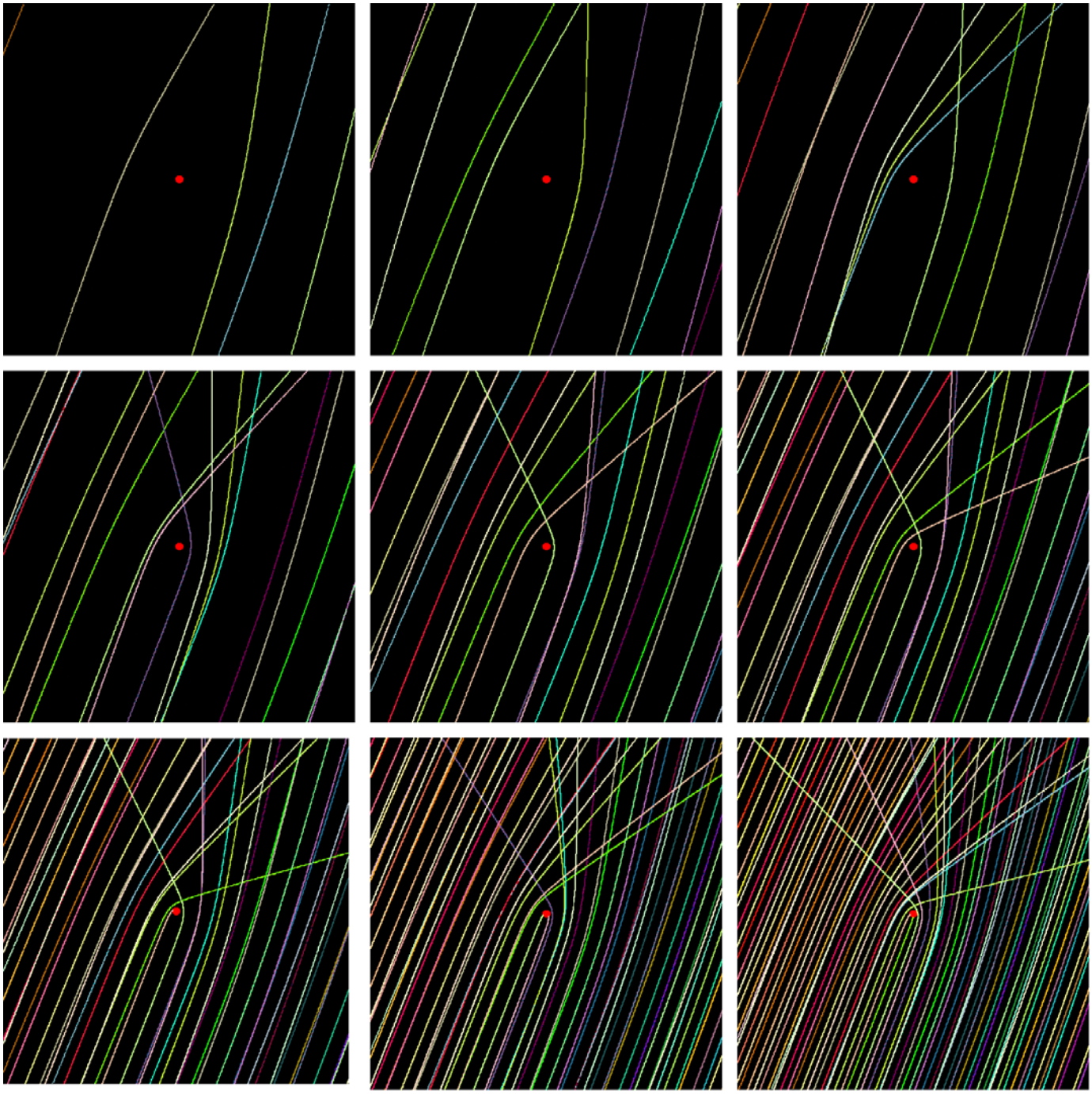
This series of pictures shows how changing the time of launch affects the spacecraft array's position, relative to Mars.
These pictures are stabalized to Mars' "point of view", with Mars in the center.
The spacecraft first pass behind Mars, then in front of Mars, and then, in the final picture, they finally reach the orbit of Mars.

The second optimization:

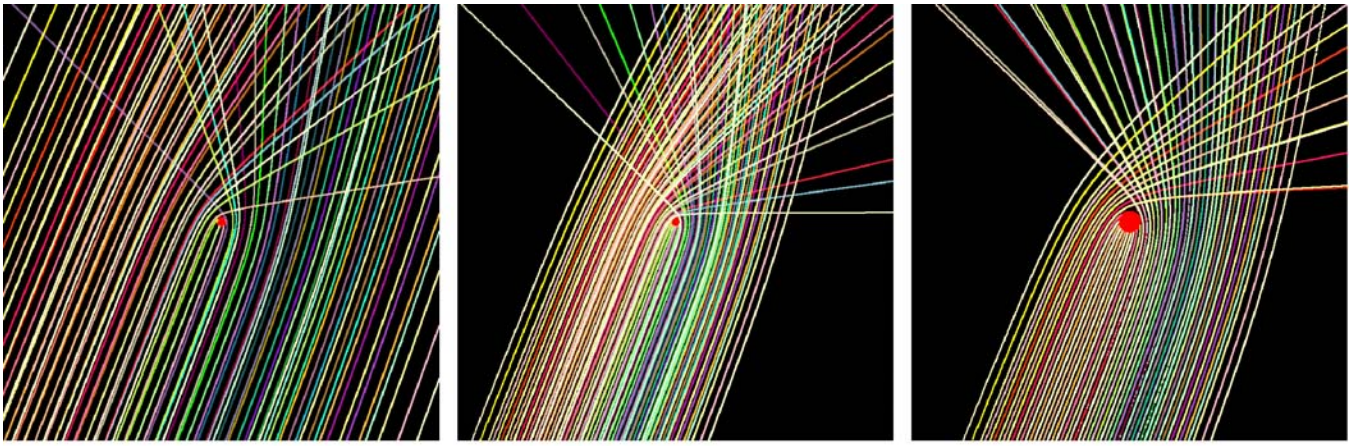


This series of pictures shows the effect of changing the initial launch angle so as to get the biggest possible gravity boost from the moon.
The angle is optimal in the last picture because the spacecraft gains the more speed from the moon than it did at any other angle. This allows it to reach Mars more quickly.
This picture is in the sun-earth rotating frame.

The third optimization:

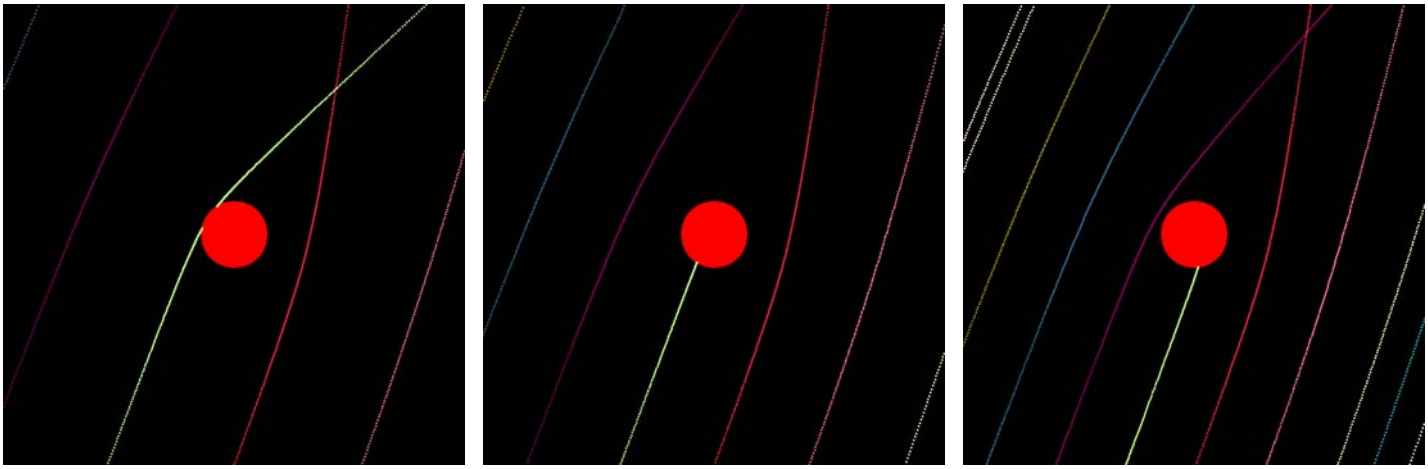


(continued)

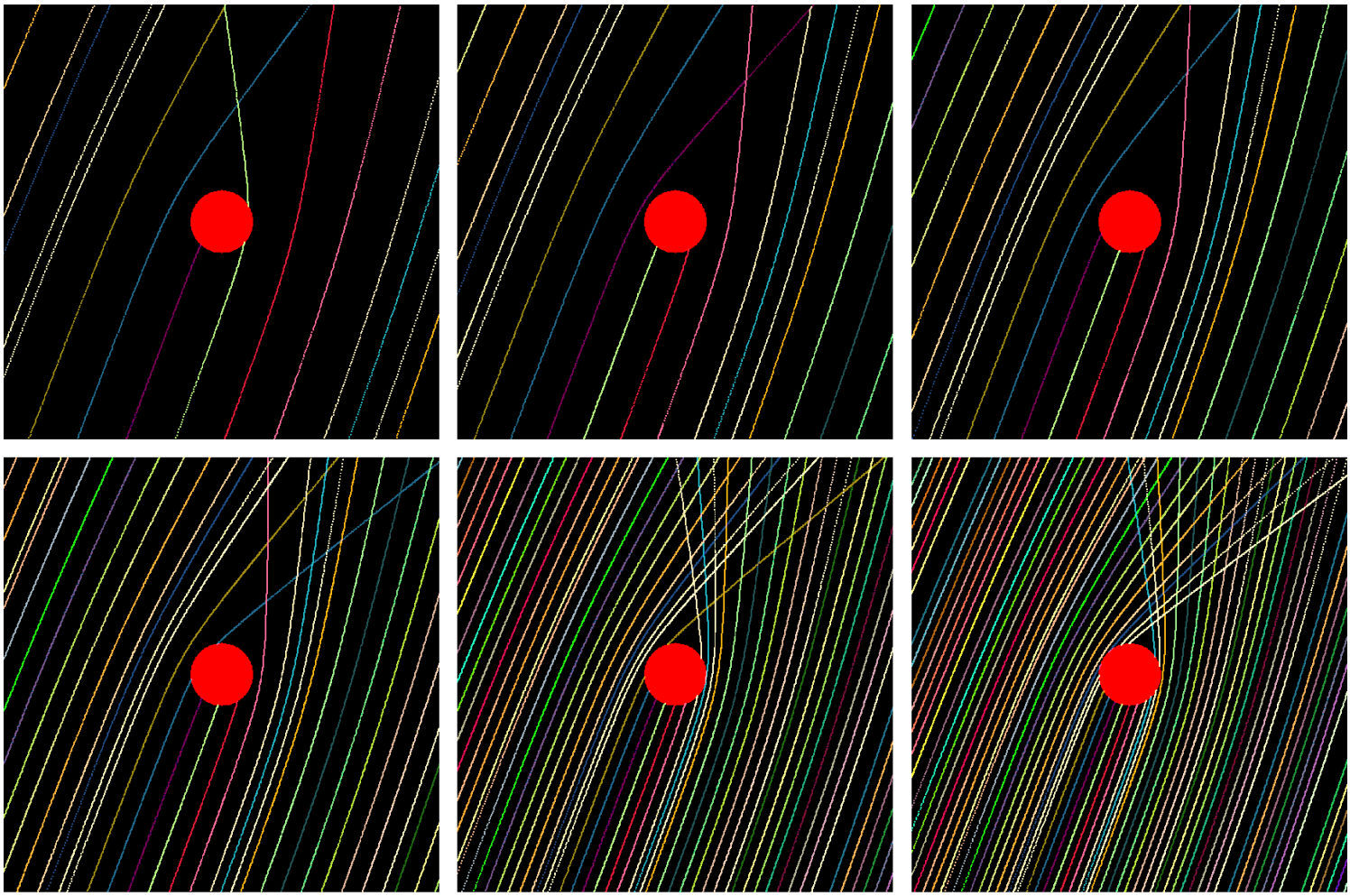


This series of pictures is taken over the course of a number of runs with the recenter function. This function reports which spacecraft went the closest to Mars and then re-centers the array of spacecrafts so that they launch with the same trajectory as the optimal spacecraft. Recenter works well for this application because there is little instability.

This picture is an enlargement of a previous step.



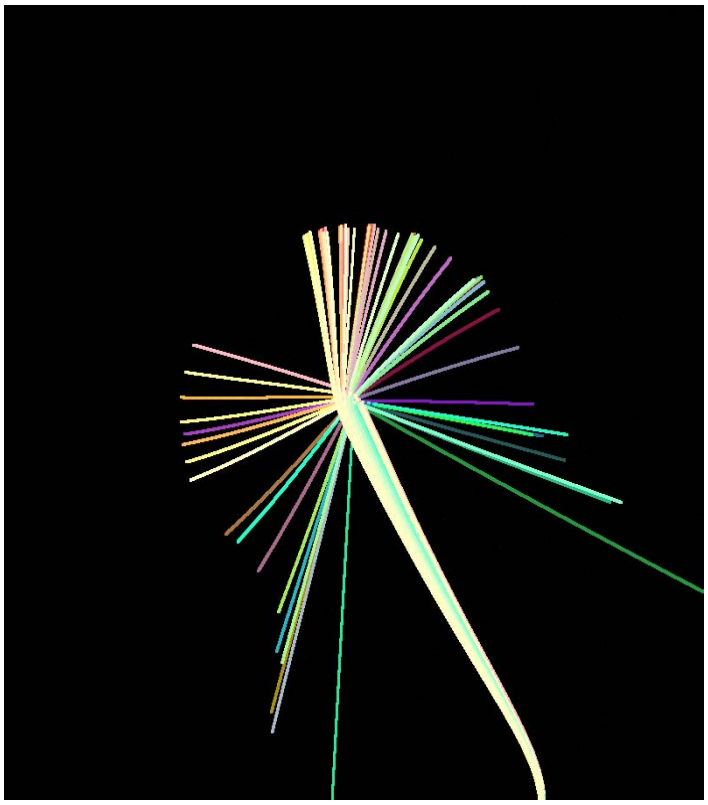
(continued)



These pictures also use the recenter function, but the function is called more.
Also, the pictures are zoomed in on Mars.

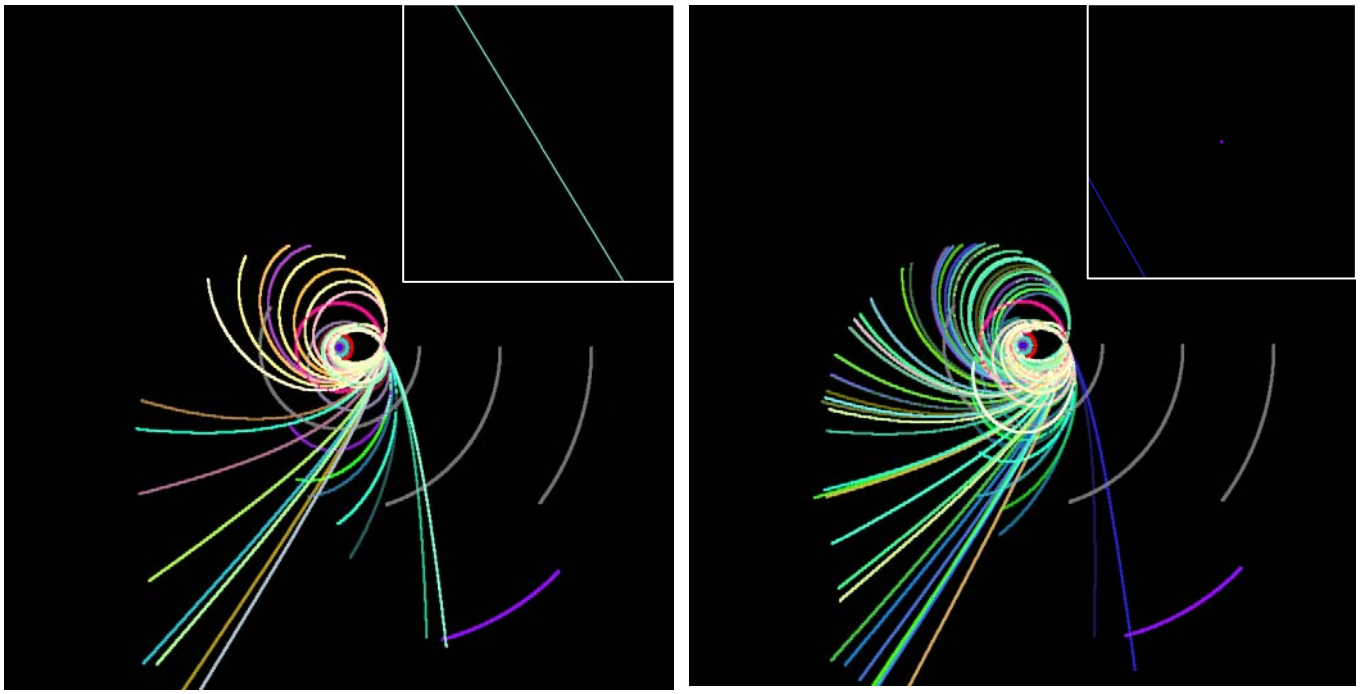
Path to Pluto

In order to reach Pluto, the spacecraft performed multiple gravity boosts. First, the spacecraft performed a slingshot off of the Moon. Then, upon reaching Jupiter, the spacecraft excecuted another slingshot. This gave them enough momentum to reach Pluto.



These two pictures show how the spacecraft diverge when they reach Jupiter.
The first is picture is non-stabalized. The second picture has Jupiter as its center.
The greatest acceleration boost comes from passing close Jupiter on the outside edge. This can be seen in the first picture.

The Effect of Recenter

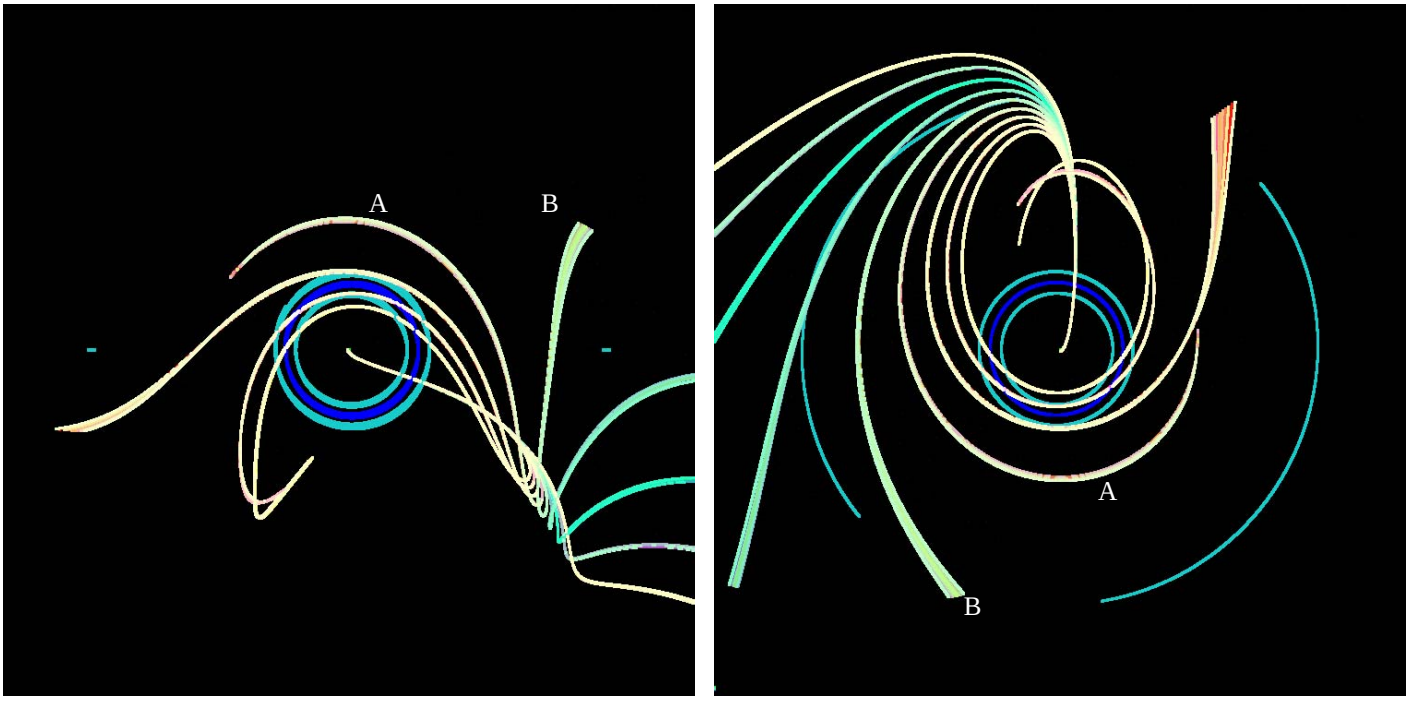


These pictures are in the Sun-centered frame of reference. The inserts in the upper right corner are from Pluto's "point of view".
In the left picture, a spacecraft passes very close to Pluto (center of insert).
In the right picture, after using the recenter function, more of the rockets pass by the moon. However, the spacecraft that comes closest to Pluto passes far in front of the moon because of its extra speed.
The instability in the recenter function was caused by running the program on the cluster for such a long simulation.

Low-Energy Path to Moon

As described in Martin Lo's Paper, *Low Energy Transfer to the Moon*[1], it is possible to use the gravity of Lagrange Points to get to the moon using 20% less energy than the traditional method. We use information from both the earth-sun frame of reference and the earth-moon frame of reference. With a three-part path and only one fuel burn, it is possible to send the spacecrafts into a loose orbit around the moon.

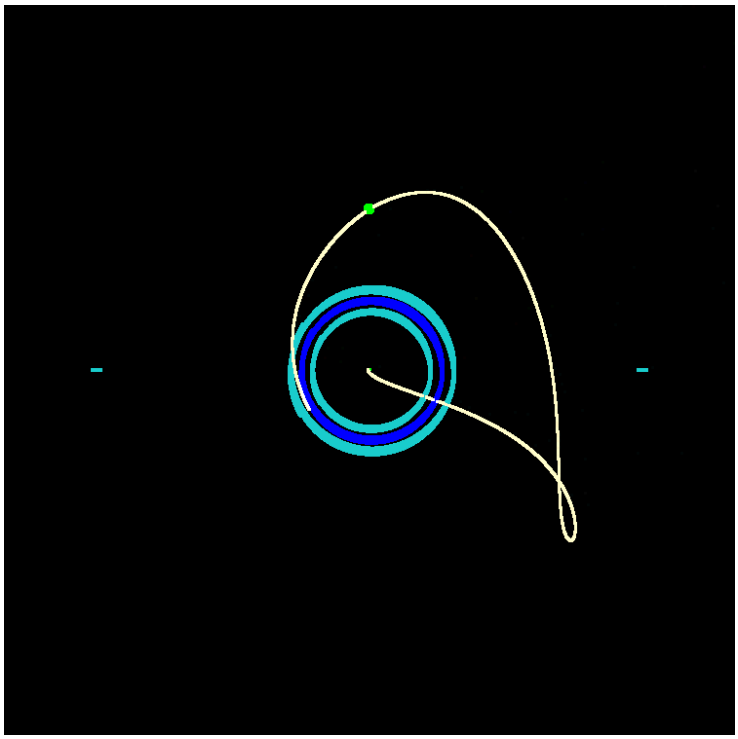
Once the spacecraft reaches the Earth-Sun Lagrange point, it has many possibilities for its end position, since there are essentially zero-energy paths to different points both inside and outside the orbital radius. In other words, the spacecraft has access to the most points for the least fuel. This versatility allows the spacecraft to carefully place itself to fall into the moon-earth system's lunar capture portion. The extra velocity required to reach the L1 point (about an additional 50 meters/second) is small in comparison to the fuel the spacecraft will save in its burn from L1 to the moon. It is thus beneficial to use L1 to get to the moon.



These two pictures show the first step in executing this maneuver. The first picture is centered on Earth with the earth-sun L1 point on the right. The second picture is also

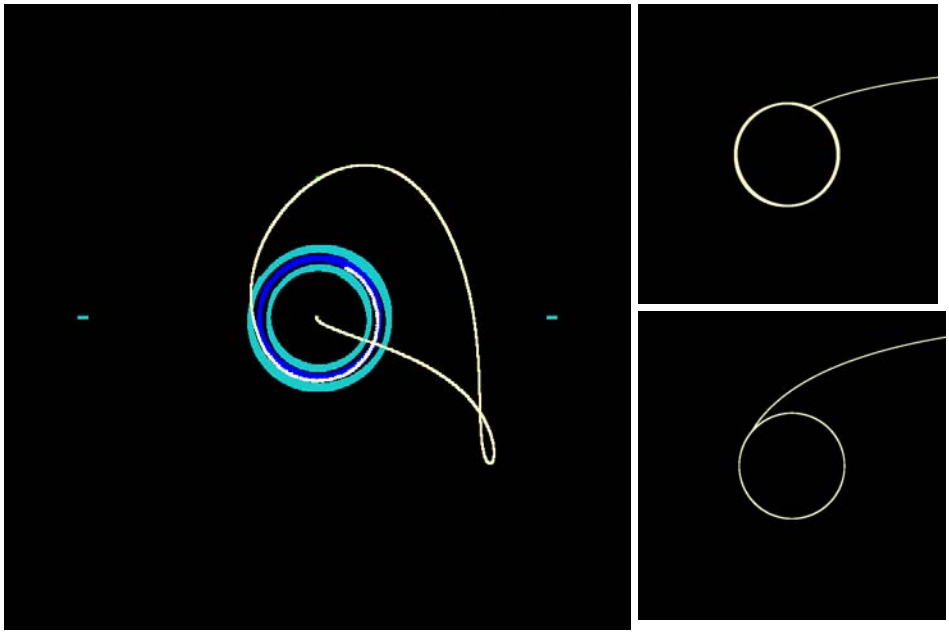
[1] Koon, W.S., M.W. Lo, J.E. Marsden and S.D. Ross [2001]

centered on the earth. The pictures demonstrate how slightly different initial positions can cause the spacecraft to either fall into orbit around the earth or to leave the system. The optimal path for the spacecraft lies between points A and B. In terms of the left picture, it runs perpendicular to the Earth-L1 line after the turn. After the spacecraft circle once, they will approach the moon's orbit from behind. This means that their initial angle must be similar to the needed trajectory.



Next, the spacecraft must perform a burn. This will change its trajectory slightly so that it better fits the moon's orbit. The green point shows where the burn occurred. This particular setup crashed into the moon, but a slight change in the burn fixed this.

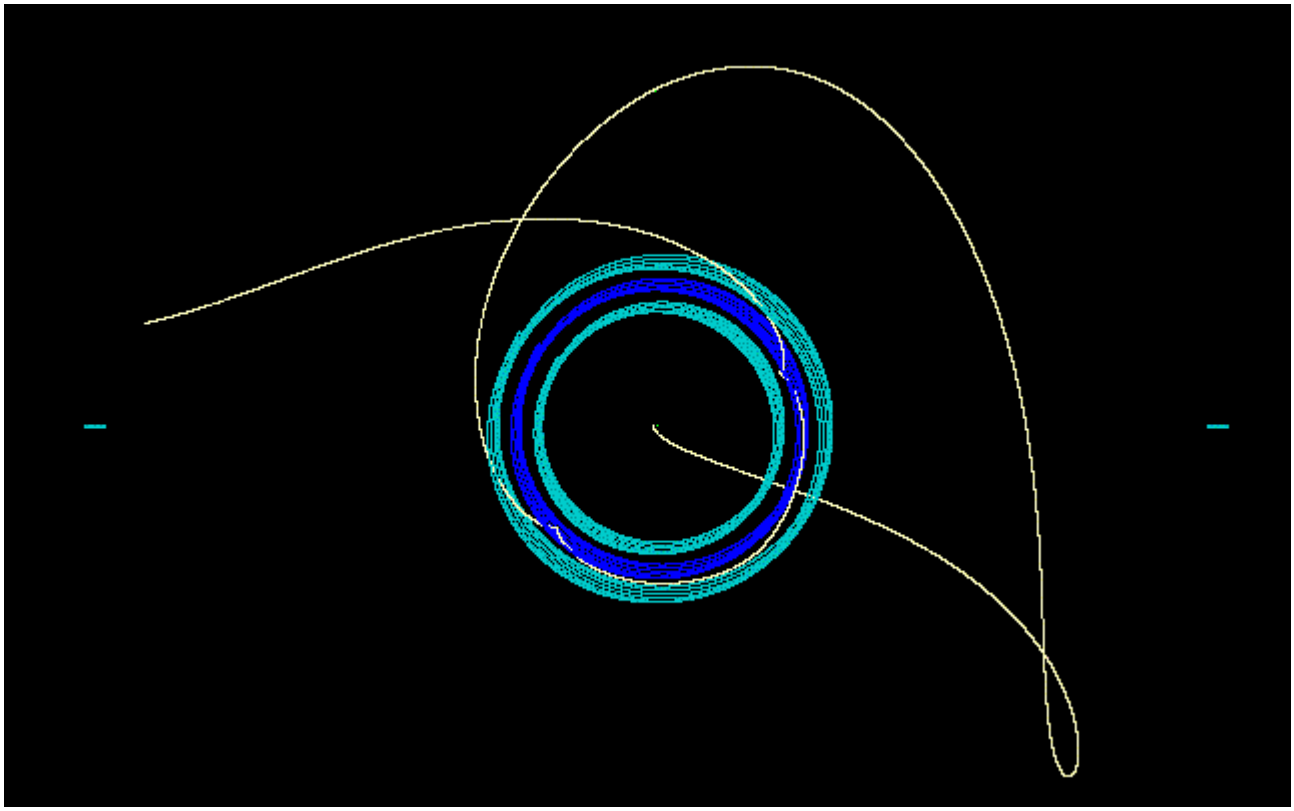
Once the moon is reached, there are two possibilities. The spacecraft will either fall into a loose or a tight orbit around the moon. A loose orbit is more natural for the spacecraft, and a tighter orbit requires another burn. However, it still uses less energy than the traditional method.



Here we show the spacecraft orbiting the moon (left) as well as two angles, controlled by the amount of burn, which change the amount of fuel needed to get into a tight orbit (right). The first angle requires more energy in both burns because the angle is steeper. The second angle requires less energy both times.

The spacecraft starts out from Earth with a velocity of 10,948.12 m/s. The second burn (at the green dot) takes off 33.5 m/s. At the very end the spacecraft gets near the moon and changes velocity by 498 m/s.

The other option involves the looser, more natural orbit around the moon. This orbit becomes unstable quickly, but it does not require a third burn.



This is an example of a loose orbit. The spacecraft orbits the moon two full times before becoming unstable.

Overall, the energy required to get to the moon using a traditional method requires11,499 m/s of thrust. With the tight orbit, the low-energy can actually take longer – it uses 11,580 m/s of thrust. However, using the loose orbit only takes 11,010 m/s of thrust. This is a 23% improvement over the traditional method.

Next Steps

During the next three weeks, we hope to accomplish several things.

(1) Characterize the accuracy of the different approximations – it is possible that the aphelion difference is not the best measure, since it requires each numerical method to progress through an entire orbit. As an alternative, we might look at the total energy as a measure of accuracy. Since energy is conserved on a real trajectory, it is not necessary to complete an orbit. Also, the energy includes the effect of the velocity, which the aphelion difference doesn not (except implicitly, to a degree). Additionally, we hope to numerically verify the order of the numerical methods and compare the efficiencies of the different algorithms. That is, compare the computational efficiency with the computational accuracy.

(2) Verify that L4 is a (weakly) stable equilibrium point, using an RK4 algorithm.

(3) Run numerical experiments with the slingshot using the standard hyperbola.

(4) Further develop the use of L1 and L2 in creating energy-efficient trajectories. The plan for further extension in this area is to send two arrays of rockets into two planets' halo orbits and then splice the final intersection. We believe that this will create a low-energy path between most pairs of planets.

Conclusion

Over the course of the project, several solutions were found which differed in their trajectory times, fuel required, and complexity. Launching an array of spacecraft and re-centering it on a selected spacecraft was suitable for gravity assisted routes. Using Lagrange points to plan low-energy paths resulted in trajectories that were about 20% more fuel-efficient, but this required increased complexity. Using the cluster improved the run time significantly, but caused some instability. The approximation methods varied in accuracy, and higher order approximations performed better. The most accurate, fast, fuel-efficient spacecraft paths came from running Delta on the cluster, creating a low-energy path with the RK4 approximation method.

Acknowledgements

We would like to thank Dr. Erik DeBenedictis and Dr. William Cordwell for mentoring our team, Mrs. Kerrie Sena for her encouragement and sponsorship of our project, and Sandia and Los Alamos National Labs for sponsoring the Supercomputing Challenge.

Finally, we would like to thank the Supercomputing Challenge consult team for their patience.

Bibliography and References

Abramowitz, M. and Stegun, I.A., Handbook of Mathematical Functions, Dover, 1965

Halliday, D., Resnick, R., and Walker, J., Fundamentals of Physics, 6th ed., J. Wiley and Sons, New York, 2001

Hut, P. and Makino, J., The Art of Computational Science, 2007, (published on the Web)

Baez, J., "Lagrange Points", http://math.ucr.edu/home/baez/lagrange.html

Cornish, N., "The Lagrange Points", http://www.physics.montana.edu/faculty/cornish/lagrange.pdf

Koon, W., Lo, M., Marsden, J., and Ross, S., "Low Energy Transfer to the Moon", http://www.gg.caltech.edu/~mwl/publications/papers/lowEnergy.pdf

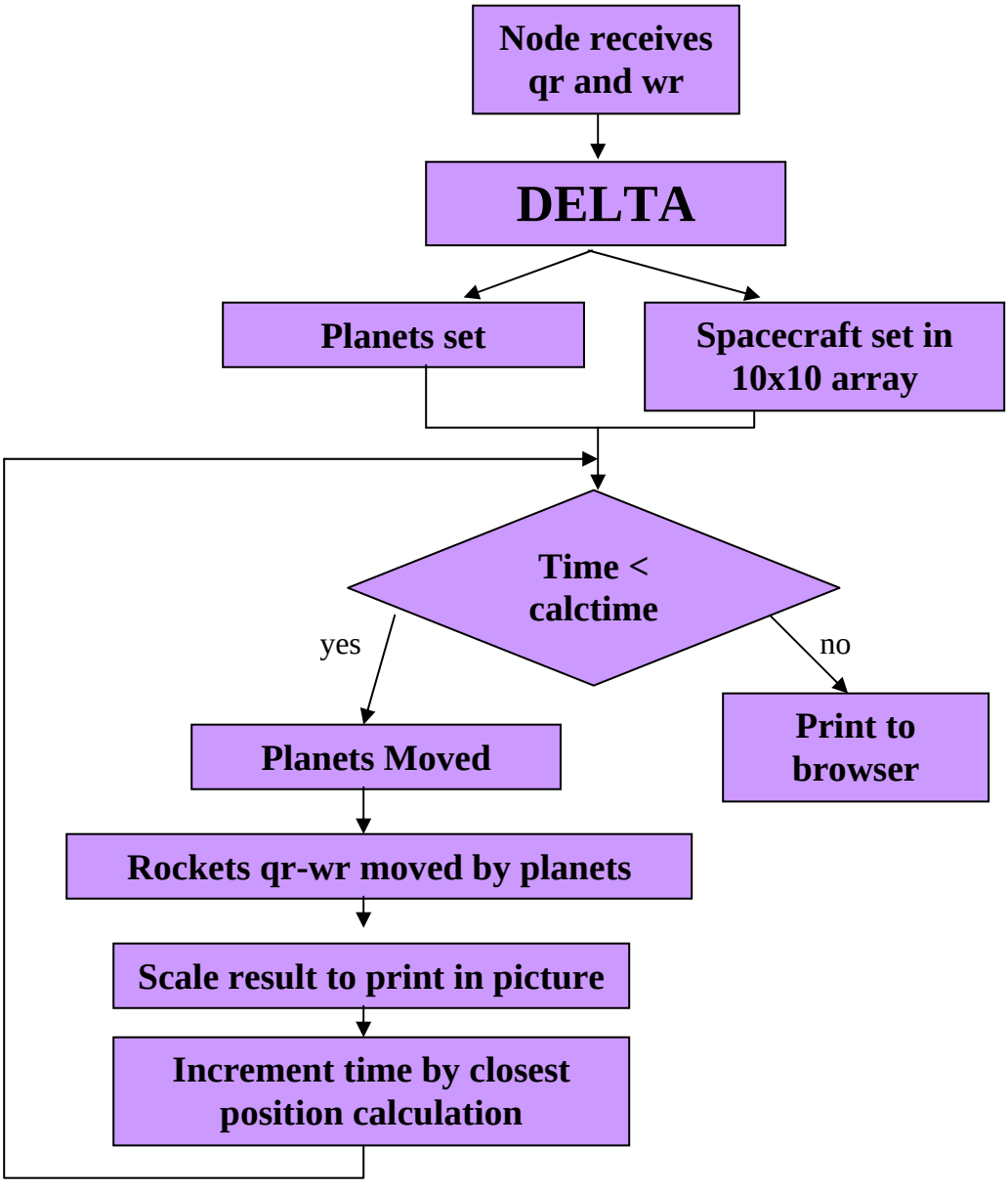
Lo, Martin, "The InterPlanetary Superhighway and the Origins Program", http://www.gg.caltech.edu/~mwl/publications/papers/IPSAndOrigins.pdf

Wikipedia, "Runge-Kutta methods", http://en.wikipedia.org/wiki/Runge-Kutta

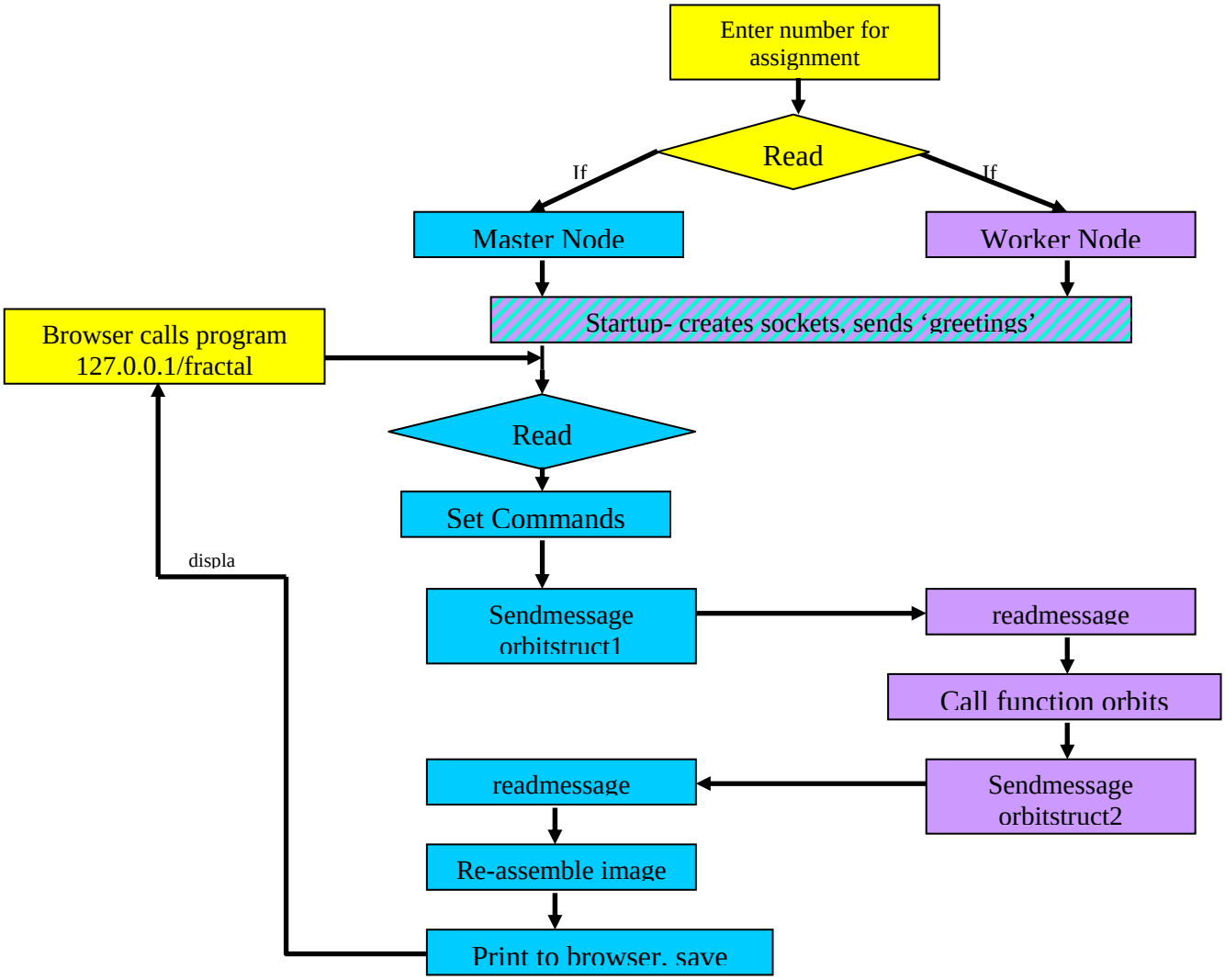
Wikipedia, "n-body problem", http://en.wikipedia.org/wiki/N-body_problem

Appendix A - Flowcharts

Flowchart of Delta



Flowchart of Cluster



Appendix B – Code

Osim.cpp
- file where function orbits is located.

```
#include "stdafx.h"                                    // must be present for Visual C++. For Unix,
stdafx.h can be empty
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

#include <crtdbg.h>
#include <direct.h>
#include <time.h>
#include <malloc.h>
#include <winsock.h>

#include "osim.h"
#include "orbits.h"
#include "body.h"
#include "data.h"


const int LEVR = 6;                                    // number of levels of red in 256 color palette
const int LEVG = 6;                                    // number of levels of green in 256 color
palette
const int LEVB = 6;                                    // number of levels of blue in 256 color palette

const int HSIZE = 5003;

const int IMGWID = 750; //HELLO IMAGE SIZE
const int IMGHGT = 750;

#define PI 3.14159265358979323846264338 3
//Functions


/************************
data::PythagoreanTheorem()
Passed: double a, double b (legnth of X and Y)
Returns: double c (distance between the points)

Note: overloaded.  other function takes three input variables.
************************/
double PythagoreanTheorem( double a, double b )
{
   return sqrt(a*a + b*b);
```

```
}


/************************
data::PythagoreanTheorem()
Passed: double x, double x, double z (legnth of X, Y, and Z)
Returns: double c (distance between the points)

Note: overloaded.  other function takes two input variables.

Update: no longer in 3D, change back? My poor little laptop
can't handle the extra # crunching. (hint hint dad)
************************/
double PythagoreanTheoremv2( double x, double y )
{
   return sqrt(x*x + y*y);
}



/************************
data::CalcLnPoint()
Passed: X, Y, and Z for the bigger body and the smaller body, the masses of these two (for L1,
L2, and L3)
Returns: nothing, sets the current X, Y, and Z for the L1 and L2 points

Calculates the X, Y, and Z of the L* Lagrangian point.
Note: x1 is the smaller body, x2 is the bigger body, x3 is the Lagrange point, x4 is the
alternate Lagrange point

Update: again, no longer in 3D
************************/
#define cbrt(x) pow(x, 1./3.)  //for some reason this didn't show up in my library

void CalcL12Point( double x_heavy, double y_heavy, double x2, double y2, double* x3, double* y3,
double* x4, double* y4, double mass_heavy, double mass2 )
{       //Equations in this function came from WikiPedia (en.wikipedia.org).
   double ChangeX = x_heavy - x2;
   double ChangeY = y_heavy - y2;
   double R = PythagoreanTheorem( ChangeX, ChangeY);
   double Hypotenuse1 = R*(cbrt(mass2 / (3 * mass_heavy)));
   double angle = atan2(ChangeY, ChangeX);
   double Hypotenuse2 = R - Hypotenuse1;   //L1 specific
   *x3 = x_heavy - cos( angle ) * Hypotenuse2;
   *y3 = y_heavy - sin( angle ) * Hypotenuse2;
   Hypotenuse2 = R + Hypotenuse1;          //L2 specific
   *x4 = x_heavy - cos( angle ) * Hypotenuse2;
   *y4 = y_heavy - sin( angle ) * Hypotenuse2;
```

```
//    *z3 = *z4 = x_heavy;     //not exact, but close.  Need to find this exactly!!!
}

void CalcL45Point( double x1, double y1, double z1, double x2, double y2, double z2, double* x3,
double* y3, double* z3, double* x4, double* y4, double* z4 )
{       //Equations in this function came from WikiPedia (en.wikipedia.org).
   double ChangeX = ( x1 - x2 ) / 2;
   double ChangeY = ( y1 - y2 ) / 2;
   double ChangeZ = ( z1 - z2 ) / 2;   //These are actualy half the real value.  I divide by two here, to avoid
doing it over and over again, later.  (preformance)
   *z3 = z2 + ChangeZ;
   *z4 = z2 + ChangeZ;     //final values
   double E = PythagoreanTheorem( ChangeY , ChangeZ );
   double F = PythagoreanTheorem( ChangeX , ChangeY  /*,ChangeZ */);
   double Theta2 = asin( E / F );  //I need to do all of the trig in radians...
   double Theta1 = 0.5 - Theta2;
   double G = F / tan( 1 / 6 );
   double I = G * sin( Theta1 );
   double J = G * cos( Theta1 );
   *x3 = x2 + ChangeX + J;
   *y3 = y2 + ChangeY + I;
   *x4 = x2 + ChangeX - J;
   *y4 = y2 + ChangeY - I;
}

/************************
classes
Vector has +,-,*,/, and rotate (rotates by a radian), and norm (absolute value)
Planet's positions and velocities are vectors
************************/
class vector{
public:

        double x,y;
        vector(double i, double j=0){
                x= i;
                y= j;
        }

        vector () { x = y = 0.0; }

        vector rotate (double theta){
                        vector draw3;
                        draw3.x= (x *(cos(theta)))+(y*sin(theta));
                        draw3.y= (x *(-sin(theta)))+(y*cos(theta));
                        return draw3;
        }

        vector operator+(double kh){vector rval= vector (x+kh, y+kh); return rval;}
```

```
        vector operator-(double kh){vector rval= vector (x-kh, y-kh); return rval;}
        vector operator*(double kh){vector rval= vector (x*kh, y*kh); return rval;}
        vector operator/(double kh){vector rval= vector (x/kh, y/kh); return rval;}

        vector operator+(vector kh){vector rval= vector (x+kh.x, y+kh.y); return rval;}
        vector operator-(vector kh){vector rval= vector (x-kh.x, y-kh.y); return rval;}
        vector operator*(vector kh){vector rval= vector (x*kh.x, y*kh.y); return rval;}
        vector operator/(vector kh){vector rval= vector (x/kh.x, y/kh.y); return rval;}


        void operator+=(vector kh){x+=kh.x, y+=kh.y;}
        void operator-=(vector kh){x-=kh.x, y-=kh.y;}

};
double norm(vector kh){return sqrt (kh.x*kh.x+kh.y*kh.y);}


#define sun 0  //originally, earth, moon, sun
#define moon 1
#define earth 2
#define mercury 3
#define venus 4
#define mars 5
#define jupiter 6
#define saturn 7
#define uranus 8
#define neptune 9
#define planets 1
#define nbody (planets)

#define nspacecrafts 1

#define spacecraftnum (nspacecrafts*nspacecrafts)

#define emlone 0
#define emltwo 1
#define selone 2
#define seltwo 3
#define nlagrange 4

/************************
startpoint is the postion of the array that I'm launching
you can get the original spacecrafts placement in an array by tellign get which spacecraft it is in terms of right and down
recenter takes a spacecraft's number printed out at the end of the program and figures out where it was, also re-places the array
************************/
class startpoint{
public:
```

```
	vector centerpoint;
	double edgesize;

	startpoint(double x, double y, double rot, double es) {
		centerpoint.x = x;
		centerpoint.y = y;
		centerpoint = centerpoint.rotate (rot);
		edgesize = es;
	}

	vector get (int i, int j) {
		double fi = (1.0*i-(nspacecrafts-1.0)/2.); // -nspacecrafts/2 .. nspacecrafts/2
		double fj = (1.0*j-(nspacecrafts-1.0)/2.);
		fi = fi/nspacecrafts;			// -1/2 .. 1/2
		fj = fj/nspacecrafts;

		vector rval;
		rval.x=centerpoint.x+fi*edgesize;
		rval.y=centerpoint.y+fj*edgesize;// -edgesize/2 .. edgesize/2
		printf("(%d, %d) -->\tDelta=(%.2f, %.2f)\tpos=(%.0f, %.0f)\n", i, j, fi, fj, rval.x, rval.y);
		return rval;
	}

	vector recenter(int prefspacecraft) { //6
		int aa = prefspacecraft/nspacecrafts;
		int bb = prefspacecraft - aa*nspacecrafts;
		centerpoint = get(aa, bb);
		edgesize *= .75;
		return centerpoint;
	}

} ;

/************************
objects are all things which have mass, namely, planets
are set with the set function
includes:
	what planet the body orbits
	how long it's year is
	the initial radian position of the planet
	position
	velocity
	mass
	color (in RGB)
	diameter
************************/

class object {				//anything that has mass
public:					//position
```

```
	int whatiorbit;
	double orbitdays;
	double initialrotation;
	vector pos;
	vector v;
	double m;
	double gm;
	COLORREF c;
	double diam;
	object () { diam = 1; c = RGB(128, 128, 128); }
	void set(double, double am, COLORREF ac, double ad, int orbits, double od, double ia =0.);
} body[nbody];

void object::set(double adist, double am, COLORREF ac, double ad, int orbits, double od, double ia) {
	pos = vector (adist,0);
	m = am;
	c = ac;
	diam= ad;
	whatiorbit=orbits;
	orbitdays=od;
	initialrotation = ia;

	double evel = 2*(norm(pos))*3.1415926535/(orbitdays*24.*60.*60.);
	vector vv(0, -evel);
	vv=vv.rotate(ia);
	v=body[whatiorbit].v+vv;

	pos= pos.rotate(ia);
	pos = body[whatiorbit].pos+pos;
	//printf("%f %f %f\n", evel, norm(av), evel-norm(av));
}

/************************
robject- big suprise! it's a spacecraft!
set with set... include:
	position
	velocity
	color
	the closest they've gotten to the target planet
	weather they've died or not
************************/
struct robject { //spacecrafts
	vector lastes1, lastes2, pos, v; //position, velocity
	COLORREF c;
	vector closestpos; //closest position to target
	int deadbit;
	int method;
	double miss;
	void set(double, double, COLORREF ac, int orbits, double od, double ia =0.);
```

```
} spacecraft[spacecraftnum];

void robject::set(double x, double y, COLORREF ac, int orbits, double velocity, double ia) {
	lastes1 = vector(0, 0);
	lastes2 = vector(0, 0);
	pos = vector (x, y);
	pos=pos.rotate(ia);
	pos=body[orbits].pos+pos;

	c = ac;

	v= vector(velocity,0);
	v= v.rotate(ia);
	v=v+body[orbits].v;
	//printf("%f %f %f %f\n", pos.x, pos.y, v.x, v.y);
}

/************************
lobject- lagrange points
only have position
************************/
struct lobject { //lagrange points
	vector pos; //position
	lobject () {}
} lagrange[nlagrange];


vector dv(vector Pos, double Deltat) {
	double g=6.673e-11;                                          //*2.5;
	vector Deltav;
	if (1) for (int j= 0; j< nbody; j++) {
		double distance = norm(Pos-body[j].pos);
		double force = (body[j].gm)/(distance*distance); //g*(body[j].m)
		vector ijvec= Pos-body[j].pos;
		ijvec=ijvec*(force/norm(ijvec));
		Deltav += ijvec*(Deltat);
	}
	return Deltav;
}

/************************
now for the function!
orbits is called in Orbits.cpp when you get a http://127.0.0.1/orbits
************************/

void orbit(SOCKET theClient) {
		send(theClient, "HTTP/1.0 200\r\n", 14, 0);
		send(theClient, "Content-type: ", 14, 0);
		send(theClient, "text/html", 9, 0);
```

```
		send(theClient, "\r\n", 2, 0);
		send(theClient, "\r\n", 2, 0);

		FILE *index = ArchiveFile("index", "htm", 0, NULL);		//saving all images
produced

		fprintf(index, "<html><head><title>Index Page Orbits</title></head><body><CODE>");
		fprintf(index, "<CENTER>Welcome to Orbits!<br>");

//introducing and setting the pictures to initial values
		static int *imagedata1[IMGWID], id1[IMGWID][IMGHGT];
		static int *imagedata2[IMGWID], id2[IMGWID][IMGHGT];
		static int *imagedata3[IMGWID], id3[IMGWID][IMGHGT];
		static int *imagedata4[IMGWID], id4[IMGWID][IMGHGT];
		static int *imagedata5[IMGWID], id5[IMGWID][IMGHGT];
		static int *imagedata6[IMGWID], id6[IMGWID][IMGHGT];
		static int *imagedata7[IMGWID], id7[IMGWID][IMGHGT];
		if (1) for (int i = 0; i < IMGWID; i++) {
			imagedata1[i] = id1[i];
			imagedata2[i] = id2[i];
			imagedata3[i] = id3[i];
			imagedata4[i] = id4[i];
			imagedata5[i] = id5[i];
			imagedata6[i] = id6[i];
			imagedata7[i] = id7[i];
		}
		// display
		if (1) for (int i = 0; i < IMGWID; i++)
			for (int j = 0; j < IMGHGT; j++) {
				imagedata1[i][j] = RGB(0, 0, 0);
				imagedata2[i][j] = RGB(0, 0, 0);
				imagedata3[i][j] = RGB(0, 0, 0);
				imagedata4[i][j] = RGB(0, 0, 0);
			}

//these 6 parameters I change a lot, so they're up top.
#define MARSSTART 180+30+5+5+3.5+3+3-.25		//initial angle of mars
#define INITIALLAUNCHEDGE 70000
#define MAXFRACTIONTHREDHOLD .05
#define LAUNCHVELOCITY 18370
		double launchangle1;;
		launchangle1 = (49.7)/57.29577951;		//positive is clockwise
		int calctime = 350*60*60;					//10e7 is one orbit
		double largescalefactor = 150e9;
		double au= 149597870691.0;				//meters
		double sf1= (50000000000);			// 5e8 for everything, 1e8 for body[earth] only,
1e11 for sun/ earth
		double sft1= (0.002*au);
		double sft2= (0.0002*au);
```

```
		double sf2= 5e8;			// 5e8 for everything, 1e8 for body[earth] only, 1e11 for sun/ earth
		double sf3 =(50000000);			// Mars
		double sf4 = (5000000);		// 3x moon diameter
		vector origin; origin.x=0; origin.y=0;


		fprintf(index, "sf1=%f<br>\n", (double)sf1);
		fprintf(index, "sft1=%f<br>\n", (double)sft1);
		fprintf(index, "sft2=%f<br>\n", (double)sft2);
		fprintf(index, "initiallaunchedge=%f<br>\n", (double)INITIALLAUNCHEDGE);
		fprintf(index, "launchangle1=%f<br>\n", launchangle1*57.29577951);
		fprintf(index, "MARSSTART=%f<br>\n", (double)MARSSTART);
		fprintf(index, "calctime=%f<br>\n", (double)calctime);

		int LastPrintDay=0;
		double cx = 125., cy = 125.;
		double orbitvel = 2.45;
		double g=6.673e-11;					//*2.5;
		double rot = 60;
		int bestMRangle = 40;

		//setting planets
		body[sun].set(0, 1.98892e30, RGB(0, 255, 255), 1380000*1000, sun, 1);
		body[earth].set(au, 5.9742e24, RGB(0, 255, 0), 12756.3*1000, sun, 365.26, 180/57.29577951); //initial positions;
		body[moon].set(768000e3/2, body[earth].m/81, RGB(255, 0, 0), 3764*1000, earth, 27.3, (200+bestMRangle)/57.29577951);
		body[mercury].set(au*.39, body[earth].m*(0.055), RGB (110, 110, 110), 4878*1000, sun, 88.);
		body[venus].set(au*.72, body[earth].m*(.82), RGB (110, 110, 110), 12103*1000, sun, 225.);
		body[mars].set(au*1.52, body[earth].m*(.11), RGB(0, 0, 255), 6794*1000, sun, 687., (MARSSTART)/57.29577951);
		body[jupiter].set(au*5.2, body[earth].m*(318), RGB (110, 110, 110), 142800*1000, sun, (11.9*365.26));
		body[saturn].set(au*9.54, body[earth].m*(95.2), RGB (110, 110, 110), 120000*1000, sun, (29.5*365.26));
		body[uranus].set(au*19.2, body[earth].m*(14.5), RGB (110, 110, 110), 51118*1000, sun, (84.*365.26));
		body[neptune].set(au*30.1, body[earth].m*(17.1), RGB (110, 110, 110), 49532*1000, sun, (165.*365.26));
		body[sun].gm=1.35*PI*PI*1e20;

		data PlanetData;
		PlanetData.d += .5/60/60/24;
		PlanetData.calcData();
		PlanetData.outputBodies(&body[sun].v.x, &body[sun].v.y,
				&body[mercury].v.x,  &body[mercury].v.y,
```

```
                        &body[venus].v.x,  &body[venus].v.y,
                        &body[earth].v.x,  &body[earth].v.y,
                        &body[moon].v.x,  &body[moon].v.y,
                        &body[mars].v.x,  &body[mars].v.y,
                        &body[jupiter].v.x,  &body[jupiter].v.y,
                        &body[saturn].v.x,  &body[saturn].v.y,
                        &body[uranus].v.x,  &body[uranus].v.y,
                        &body[neptune].v.x,  &body[neptune].v.y );
            printf("Earth (%.15f %.15f) au\n", body[earth].v.x/au, body[earth].v.y/au);
            printf("Mars (%.15f %.15f) au\n", body[mars].v.x/au, body[mars].v.y/au);

            PlanetData.d -= 1./60/60/24;
            PlanetData.calcData();
            PlanetData.outputBodies(&body[sun].pos.x, &body[sun].pos.y,
                        &body[mercury].pos.x,  &body[mercury].pos.y,
                        &body[venus].pos.x,  &body[venus].pos.y,
                        &body[earth].pos.x,  &body[earth].pos.y,
                        &body[moon].pos.x,  &body[moon].pos.y,
                        &body[mars].pos.x,  &body[mars].pos.y,
                        &body[jupiter].pos.x,  &body[jupiter].pos.y,
                        &body[saturn].pos.x,  &body[saturn].pos.y,
                        &body[uranus].pos.x,  &body[uranus].pos.y,
                        &body[neptune].pos.x,  &body[neptune].pos.y );
            printf("Earth (%.15f %.15f) au\n", body[earth].pos.x/au, body[earth].pos.y/au);
            printf("Mars (%.15f %.15f) au\n", body[mars].pos.x/au, body[mars].pos.y/au);

            int wjk=0;
            while (wjk<nbody){
                printf("(%f %f) (%f %f)\n", body[wjk].v.x, body[wjk].v.y, body[wjk].pos.x,
body[wjk].pos.y);
                body[wjk].v = body[wjk].v-body[wjk].pos;
                wjk++;

            }
            printf("Earth (%.15f %.15f)\n", body[earth].v.x, body[earth].v.y);
            printf("Earth (%.15f %.15f)\n", body[mars].v.x, body[mars].v.y);

            PlanetData.d += .5/60/60/24;
            PlanetData.calcData();
            PlanetData.outputBodies(&body[sun].pos.x, &body[sun].pos.y, &body[mercury].pos.x,
&body[mercury].pos.y, &body[venus].pos.x,  &body[venus].pos.y, &body[earth].pos.x,
&body[earth].pos.y, &body[moon].pos.x,  &body[moon].pos.y, &body[mars].pos.x,  &body[mars].pos.y,
&body[jupiter].pos.x,  &body[jupiter].pos.y, &body[saturn].pos.x,  &body[saturn].pos.y,
&body[uranus].pos.x,  &body[uranus].pos.y, &body[neptune].pos.x,  &body[neptune].pos.y );
            printf("Earth (%f %f) au\n", body[earth].pos.x/au, body[earth].pos.y/au);

            if (1) for (int wjk = 0; wjk < nbody; wjk++) {
                printf("%d: pos=(%f %f) au v=(%f %f) km/s\n", wjk, body[wjk].pos.x/au,
body[wjk].pos.y/au, body[wjk].v.x/1000., body[wjk].v.y/1000.);
```

```
		}

		body[wjk].pos = body[body[wjk].whatiorbit].pos+body[wjk].pos;

		//setting the spacecrafts

		vector testlaunchpoint; testlaunchpoint.x=((sqrt(3)/2)*largescalefactor);  testlaunchpoint.y
=0;

		vector launchpoint; launchpoint.x =  (12756300.0+200/0.62);
		launchpoint.y =  0;
		launchpoint= launchpoint.rotate (launchangle1);
		startpoint timeone(testlaunchpoint.x, 12756300.0+200/0.62, 0.0,
INITIALLAUNCHEDGE); //x, y, rot, es

/*		timeone.recenter(58);
		timeone.recenter(31);
		timeone.recenter(77);
		timeone.recenter(59);
		timeone.recenter(40);
		timeone.recenter(40);
		timeone.recenter(21);
		timeone.recenter(12);
		timeone.recenter(68);
		timeone.recenter(40);
		timeone.recenter(40);
		timeone.recenter(40);
		timeone.recenter(78);
		timeone.recenter(78);
		timeone.recenter(78);
		timeone.recenter(37); */


		if (1) for (int i = 0; i < nspacecrafts; i++)
			if (1) for (int j = 0; j < nspacecrafts; j++) {
				spacecraft[i*nspacecrafts + j].deadbit= 0;
				int bluecolor = 200-(5*i*i*i+1);
				int redcolor = 255-(j*j*j+1);
				int greencolor = 200-(9*j*i*j*i-50);
				spacecraft[i*nspacecrafts + j].set(timeone.get(i, j).x,
					timeone.get(i, j).y,
					RGB(bluecolor, greencolor, redcolor),
					earth,
					LAUNCHVELOCITY/0.62/60/60*1000,
//					(202.8+bestMRangle)/57.29577951
					launchangle1
					);
			}
```

```
		spacecraft[0].pos.y= -1.73e10;
		spacecraft[0].pos.x= 1.1249e10;
		spacecraft[0].v.y= 20000;
		spacecraft[0].v.x= 0;
		spacecraft[0].method=3;
		spacecraft[0].c= RGB(255,0,0);			// blue Newton

		fprintf(index, "spacecraft[0].v.y=%f<br>\n", (double)spacecraft[0].v.y);

/*		spacecraft[1].pos.x= (1.+sqrt(3.)/2.)*largescalefactor;
		spacecraft[1].pos.y=0;
		spacecraft[1].v.y=sqrt(2.7) *PI*1e10* sqrt((2.-sqrt(3.))/(2.+sqrt(3.)))/
sqrt(2.*largescalefactor);
		spacecraft[1].v.x=0;
		spacecraft[1].method=2;
		spacecraft[1].c= RGB(0,255,0);			// green 2nd order

		spacecraft[2].pos.x= (1+sqrt(3)/2)*largescalefactor;
		spacecraft[2].pos.y=0;
		spacecraft[2].v.y=sqrt(2.7) *PI*1e10* sqrt((2.-sqrt(3.))/(2.+sqrt(3.)))/
sqrt(2.*largescalefactor);
		spacecraft[2].v.x=0;
		spacecraft[2].method=3;
		spacecraft[2].c= RGB(0,0, 255);			// red RK2

		spacecraft[3].pos.x= (1+sqrt(3)/2)*largescalefactor;
		spacecraft[3].pos.y=0;
		spacecraft[3].v.y=sqrt(2.7) *PI*1e10* sqrt((2.-sqrt(3.))/(2.+sqrt(3.)))/
sqrt(2.*largescalefactor);
		spacecraft[3].v.x=0;
		spacecraft[3].method=4;
		spacecraft[3].c= RGB(0,255,255);	// yellow RK4

		spacecraft[4].pos.x= (1+sqrt(3)/2)*largescalefactor;
		spacecraft[4].pos.y=0;
		spacecraft[4].v.y=sqrt(2.7) *PI*1e10* sqrt((2.-sqrt(3.))/(2.+sqrt(3.)))/
sqrt(2.*largescalefactor);
		spacecraft[4].v.x=0;
		spacecraft[4].method=5;
		spacecraft[4].c= RGB(255,255,255); // white other method

		spacecraft[5].pos.x= (1+sqrt(3)/2)*largescalefactor;
		spacecraft[5].pos.y=0;
		spacecraft[5].v.y=sqrt(2.7) *PI*1e10* sqrt((2.-sqrt(3.))/(2.+sqrt(3.)))/
sqrt(2.*largescalefactor);
		spacecraft[5].v.x=0;
		spacecraft[5].method=6;
		spacecraft[5].c= RGB(255,255,0);	// blue green is real oval */
```

```
        body[sun].pos.x= 0;
        body[sun].pos.y= 0;
        body[sun].v.x= 13000;
        body[sun].v.y= 0;
        body[sun].gm= (10000000000000000);

        /***********
        to be consistant, the format for lagrang points names' will be larger body, smaller body,
        l#. We'll deal with problems about the planets having same names later, it'll probably
        only be for a few
        ***********/
        lagrange[emlone].pos= 0; //earth-moon L1 point
        lagrange[emltwo].pos= 0; //earth-moon L2 point
        lagrange[selone].pos= 0; //earth-moon L1 point
        lagrange[seltwo].pos= 0; //earth-moon L2 point


        /**********
        start of main program calculations
        **********/
        double dt = 0.25; //Delta t
        int KeepGoing = 1;
        int LastDayPrinted = -1;
        double minimumdistancetomars = 1e50;
        int minimumdistancetomarsnum = 0;

        for (double t= 0; t<calctime && KeepGoing; t+=dt){          //days*hours*minutes*seconds
                KeepGoing = 0;
                double day= t/(60*60);


/*              data PlanetData;

        PlanetData.d = day;
        PlanetData.calcData();
        PlanetData.outputBodies(&body[sun].pos.x, &body[sun].pos.y,
    &body[mercury].pos.x,  &body[mercury].pos.y,
    &body[venus].pos.x,  &body[venus].pos.y,
    &body[earth].pos.x,  &body[earth].pos.y,
    &body[moon].pos.x,  &body[moon].pos.y,
    &body[mars].pos.x,  &body[mars].pos.y,
    &body[jupiter].pos.x,  &body[jupiter].pos.y,
    &body[saturn].pos.x,  &body[saturn].pos.y,
    &body[uranus].pos.x,  &body[uranus].pos.y,
    &body[neptune].pos.x,  &body[neptune].pos.y ); */

                body[0].pos += body[0].v*dt;
```

```
                double maxfraction = 0;
                if (1) for (int a = 0; a < nspacecrafts; a++)                    //altering spacecrafts
calculator
                        if (1) for (int b = 0; b < nspacecrafts; b++) {
                                int i= (a*nspacecrafts + b);
                                if (spacecraft[i].deadbit==1){ //don't calculate dead spacecrafts
                                        continue;
                                }

#define rkf(p, tt) \
                                if (1) for (int j= 0; j< nbody; j++) { \
                                        vector pp = p; \
                                        double distance = norm(pp-body[j].pos); \
                                        double force = body[j].gm/(distance*distance); \
                                        vector ijvec= pp-body[j].pos; \
                                        ijvec=ijvec*(force/norm(ijvec)); \
                                        Deltav += ijvec*(tt); \
                                }
                                vector Deltav;

if (spacecraft[i].method==1){
                                // update position and velocity
                                spacecraft[i].pos += spacecraft[i].v*dt;

                                // calculate Deltav at point p for time dt
                                vector p = spacecraft[i].pos;
                                double tt = dt;
                                rkf(p, tt);

                                spacecraft[i].v -= Deltav;

}

else if (spacecraft[i].method==2){
                                // SECOND ORDER NEWTON
                                // calculate Deltav at point p for time dt
                                vector p = spacecraft[i].pos;
                                vector Deltav1=dv(p, dt);

                                // update position and velocity
                                spacecraft[i].pos += spacecraft[i].v*dt-Deltav1*0.5*dt;


                                spacecraft[i].v -= Deltav1*.5;

                                p = spacecraft[i].pos;
                                vector Deltav2=dv(p, dt);

                                spacecraft[i].v -= Deltav2*.5;
```

```
                                        // SECOND ORDER NEWTON
}

else if (spacecraft[i].method==3){
                                        // RK2
                                        // calculate Deltav at point p for time dt
                                        vector rhalf = spacecraft[i].pos+(spacecraft[i].v*0.5*dt);
                                        vector Deltav= dv(rhalf, dt)*(-1);
                                        vector vhalf= spacecraft[i].v+(Deltav*.5);
                                        vector rr= spacecraft[i].pos+(vhalf*dt);
                                        vector Deltavhalf= dv(rhalf, dt)*(-1);
                                        vector vv= spacecraft[i].v+ Deltavhalf;

                                        spacecraft[i].pos= rr;
                                        spacecraft[i].v= vv;
                                        // RK2
}

else if (spacecraft[i].method==4) {
#if 1
                                        double n11 = 0;                 // (199) in ch04.html
                                        double n21 = (2.0/3.0);
                                        double n22 = 1./3.; //(2.0/9.0);
                                        double alpha1= (.25);
                                        double alpha2= (.25);
                                        double beta1= (.25);
                                        double beta2=(.75);
#else
                                        double n11 = (1./3.);   // (201) in ch04.html
                                        double n21 = 1;
                                        double n22 = (2.0/3.0);
                                        double alpha1= (.5);
                                        double alpha2= (0);
                                        double beta1= (.75);
                                        double beta2=(.25);
#endif
                                        vector x0 = spacecraft[i].pos;
                                        vector v0 = spacecraft[i].v;

                                        vector k1 = dv(x0 + v0*n11*dt, 1.0)*-1;
                                        vector k2 = dv(x0 + v0*n21*dt + k1*n22*dt*dt, 1.0)*-1;

                                        vector x1= x0 + v0*dt + (k1*alpha1+k2*alpha2)*dt*dt;
                                        vector v1= v0 + (k1*beta1+k2*beta2)*dt;

                                        spacecraft[i].pos=x1;
                                        spacecraft[i].v=v1;

                                        // RK4
```

```
}

else if (spacecraft[i].method==5) {  //version 5, section 5.2, ch05.html

                                        double alpha= 0;
                                        double eta= .5;
                                        double zeta= 0;

                                        double n11 = 0;
                                        double n21 = (eta);
                                        double n22 = .5*eta*eta; //(2.0/9.0);
                                        double n32 = .5- zeta;
                                        double n33= zeta;
                                        double alpha1= (1./3. + alpha*(eta-1));
                                        double alpha2= (alpha);
                                        double alpha3= ((1./6.)-alpha*eta);
                                        double beta1= ((3*eta-1)/(6*eta));
                                        double beta2=(1/(6*eta*(1-eta)));
                                        double beta3= (2-3*eta)/(6*(1-eta));
#if 1  //checking to make sure these are correct
#define ASSERTAEQ(a, b) if ((b) != 0.) { double aa = (a), bb = (b), q = aa/bb; ASSERT(q < 1.001 && q
> .999); } else ASSERT(abs(a) < 1e-6);
                                        ASSERTAEQ(alpha1 + alpha2 + alpha3, .5);
                                        ASSERTAEQ(alpha2*n21 + alpha3, 1./6.);
                                        ASSERTAEQ(beta1 + beta2 + beta3, 1);
                                        ASSERTAEQ(beta2*n21 + beta3, .5);
                                        ASSERTAEQ(beta2*n21*n21 + beta3, 1./3.);
                                        ASSERTAEQ(beta2*n22 + beta3*(n32+n33), 1./6.);
                                        ASSERTAEQ(n32 + n33, .5);
#endif



                                        vector x0 = spacecraft[i].pos;
                                        vector v0 = spacecraft[i].v;

                                        vector k1 = dv(x0 + v0*n11*dt, 1.0)*-1;
                                        vector k2 = dv(x0 + v0*n21*dt + k1*n22*dt*dt, 1.0)*-1;
                                        vector k3 = dv(x0 + v0*dt + k1*n32*dt*dt + k2*n33*dt*dt, 1.0)*-1;

                                        vector x1= x0 + v0*dt + (k1*alpha1 + k2*alpha2 +
k3*alpha3)*dt*dt;
                                        vector v1= v0 + (k1*beta1 + k2*beta2 + k3*beta3)*dt;

                                        spacecraft[i].pos=x1;
                                        spacecraft[i].v=v1;

                                        // version 5 */
                                        }
```

```
else if (spacecraft[i].method==6){
                                        // ellipse
                                        // calculate Deltav at point p for time dt
                                        double rr= (largescalefactor/4)/(1-(sqrt(3)/2)*(cos((2*PI*t)/10e7)));
                                        double rx= rr*cos((2*PI*t)/10e7);
                                        double ry=rr*sin((2*PI*t)/10e7);
                                        spacecraft[i].pos.x=rx;
                                        spacecraft[i].pos.y=ry;
                                        // ellipse
}

                                        // various min distance things
                                        double mindist = 1e50;
                                        for (int j= 0; j< nbody; j++){                          //check to see if
you've reached a new best distance
                                                double distance = norm(spacecraft[i].pos-body[0].pos);
                                                if (mindist > distance) mindist = distance;

                                                if (norm(spacecraft[i].pos-
body[sun].pos)>norm(spacecraft[i].closestpos)) //increments record closeness for each spacecraft
                                                        spacecraft[i].closestpos=norm(spacecraft[i].pos-
body[sun].pos);

                                                double suntotarget=norm(body[mars].pos-body[sun].pos);
//finds distances for stopping
                                                double suntoprobe=norm(spacecraft[i].pos-body[sun].pos);
                                                double marstoprobe=norm(spacecraft[i].pos-body[sun].pos);
                                                if (minimumdistancetomars > marstoprobe) {
                                                        minimumdistancetomars = marstoprobe;
                                                        minimumdistancetomarsnum = i;
                                                }
                                                        KeepGoing=1;
                                        }

                                        //check if the spacecraft has hit a planet
                                        if (norm(body[moon].pos-spacecraft[i].pos)<body[moon].diam/2) {
                                                spacecraft[i].deadbit=1;
                                                        printf ("spacecraft %d has died from moon collision
on day %f (%f %f)\n", i, day, norm(body[moon].pos-spacecraft[i].pos), body[moon].diam/2);
                                                spacecraft[i].pos= vector (-au*1.2);
                                                spacecraft[i].v=0;
                                                spacecraft[i].c = RGB(0, 0, 0);                         //dead
spacecrafts=white
                                        }

                                        if (norm(body[mars].pos-spacecraft[i].pos)<body[mars].diam/2) {
                                                spacecraft[i].deadbit=1;
                                                        printf ("spacecraft %d has died from mars collision on
day %f (%f %f)\n", i, day, norm(body[mars].pos-spacecraft[i].pos), body[mars].diam/2);
```

```
                                        spacecraft[i].pos= vector (-au*1.2);
                                        spacecraft[i].v=0;
                                        spacecraft[i].c = RGB(0, 0, 0);                 //dead
spacecrafts=white
                                }

                                if (t>5*24*60*60) { //they're allowed to be at earth for the first 5
days
                                        if (norm (body[earth].pos-
spacecraft[i].pos)<body[earth].diam/2) {
                                                spacecraft[i].deadbit=1;
                                                printf ("spacecraft %d has died from earth collision
on day %f \n", i, day);
                                                spacecraft[i].pos= vector (-au*1.2);
                                                spacecraft[i].v=0;
                                                spacecraft[i].c = RGB(0, 0, 0);
        //dead spacecrafts=white
                                        }
                                }
                                double fraction = norm(spacecraft[i].v)*dt/mindist;
                                if (maxfraction < fraction) maxfraction = fraction;
                        }

                int Today = (int)day;           //print just once a day (this was overly complicated)
                if (LastDayPrinted/2 != Today/2) {
                        LastDayPrinted = Today;
                        double percentdone= (calctime/t);
                        printf ("%.5f, (dt=%f), day %f (dist mars %.0fkm %d)\n", maxfraction, dt,
day, minimumdistancetomars/1000, minimumdistancetomarsnum); //max percent dist change, time,
percent done
                        printf("probe velo x= %f, probe velo y %f\n", spacecraft[0].v.x,
spacecraft[0].v.y);
                        minimumdistancetomars = 1e50;
                }

                if (1) for (int i=0; i<(nspacecrafts*nspacecrafts); i++){
                        double a = fmod(t, 10e7);
                        double b = fmod(t-dt, 10e7);
                        if (a < b) {
//                              if (t>misstime*10e7 && t-dt< misstime*10e7){
                                vector dd; dd.x= spacecraft[i].pos.x-
((1.+sqrt(3.)/2.)*largescalefactor); dd.y= spacecraft[i].pos.y-0;
                                spacecraft[i].miss=norm(dd);
//                              printf ("day %f, spacecraft %d reached aphelion %f\n", day, i, dd);
//printing when it's got to the aphelion
                                printf ("%f\n", spacecraft[i].miss);
                                if (i == 7)
                                        double pi = 3.14;
                        }
```

```
                }

                //incriment how many seconds we're calculating at once if the spacecrafts aren't
near anything cool
                dt *= MAXFRACTIONTHREDHOLD/maxfraction;
                maxfraction = 0;
                if (dt>0.15) dt=0.25;
                if (dt<0.05){
                        int pi=3.14;
                        dt=0.1;
                }

                //and finally we calculate lagrange points!
        //      CalcL12Point(body[earth].pos.x, body[earth].pos.y, body[moon].pos.x,
body[moon].pos.y, &lagrange[emlone].pos.x, &lagrange[emlone].pos.y,  &lagrange[emltwo].pos.x,
&lagrange[emltwo].pos.y,  body[earth].m, body[moon].m);
                //earth-moon L1 and L2 point re-calculation
        //      CalcL12Point(body[sun].pos.x, body[sun].pos.y, body[earth].pos.x,
body[earth].pos.y, &lagrange[selone].pos.x, &lagrange[selone].pos.y,  &lagrange[seltwo].pos.x,
&lagrange[seltwo].pos.y,  body[sun].m, body[earth].m);
                //and the earth-moon L1 + L2 points


                /**********
                end of main program calculations
                **********/

                /***********
                now for the drawing section. First it un-rotates the stuff (first 2 picture) then
                assigns each body/ point a # and the color for the #. Writes to imagedata1, 2, 3, and
4.
                ***********/

/***********
sun in middle but with rotation, takes sf1
***********/

        if (1) for (int i= 0; i< nbody; i++){

                COLORREF t = body[i].c;

                vector es= (body[i].pos-origin) * ((IMGHGT/2)/sf1) + vector ((IMGWID/2),
(IMGHGT/2));
                        double diam = body[i].diam * ((IMGHGT/2)/sf1);

                imagedata1[IMGWID/2][IMGHGT/2]=RGB(255,255,255);

                if (es.x> IMGWID || es.x<0 || es.y> IMGHGT || es.y<0) ;
                else
```

```
                        imagedata1[(int)es.x][(int)es.y]= t;
                }

                if (1) for (int i= 0; i< nspacecrafts*nspacecrafts; i++){

                        COLORREF t = spacecraft[i].c;

                        vector es= (spacecraft[i].pos-origin) * ((IMGHGT/2)/sf1) + vector ((IMGWID/2),
(IMGHGT/2));

                        if (es.x> IMGWID || es.x<0 || es.y> IMGHGT || es.y<0) ;
                        else
                                imagedata1[(int)es.x][(int)es.y]= t;
                }


/************
sun in middle but with rotation, takes sft1
************/

                vector movevecone; movevecone.x= -((1.+sqrt(3.)/2.)*largescalefactor); movevecone.y=0;

                if (1) for (int i= 0; i< nbody; i++){

                        COLORREF t = body[i].c;

                        vector es= (body[i].pos-origin+movevecone) * ((IMGHGT/2)/sft1) + vector
((IMGWID/2), (IMGHGT/2));
                                double diam = body[i].diam * ((IMGHGT/2)/sft1);

                        if (es.x> IMGWID || es.x<0 || es.y> IMGHGT || es.y<0) ;
                        else
                                imagedata2[(int)es.x][(int)es.y]= t;
                }

                if (1) for (int i= 0; i< nspacecrafts*nspacecrafts; i++){

                        COLORREF t = spacecraft[i].c;

                        vector es= (spacecraft[i].pos-origin+movevecone) * ((IMGHGT/2)/sft1) + vector
((IMGWID/2), (IMGHGT/2));
                        vector &les = spacecraft[i].lastes1;

                        if (es.x > IMGWID && les.x > IMGWID) ;
                        else if (es.x < 0 && les.x < 0) ;
                        else if (es.y > IMGHGT && les.y > IMGHGT) ;
                        else if (es.y < 0 && les.y < 0) ;
                        else if (spacecraft[i].lastes1.x != 0)
```

```
                        LineWrite(imagedata2, IMGWID, IMGHGT, spacecraft[i].lastes1.x,
spacecraft[i].lastes1.y, es.x, es.y, t, 1, 2);
/*                      for (int i = 0; i < 10; i++) {
                                vector pes = spacecraft[i].lastes*(9.-i)/9. + es*i/9.;
                                if (pes.x> IMGWID || pes.x<0 || pes.y> IMGHGT || pes.y<0) ;
                                else
                                        imagedata2[(int)pes.x][(int)pes.y]= t;
                        }
*/
                        spacecraft[i].lastes1 = es;
/*                      vector es= (spacecraft[i].pos-body[sun].pos+movevecone) * ((IMGHGT/2)/sft1) +
vector ((IMGWID/2), (IMGHGT/2));

                        if (es.x> IMGWID || es.x<0 || es.y> IMGHGT || es.y<0) ;
                        else
                                imagedata2[(int)es.x][(int)es.y]= t;
*/
                }

/************
sun in middle but with rotation, takes sf1
************/

                if (1) for (int i= 0; i< nbody; i++){

                        COLORREF t = body[i].c;

                        vector es= (body[i].pos-origin+movevecone) * ((IMGHGT/2)/sft2) + vector
((IMGWID/2), (IMGHGT/2));
                                double diam = body[i].diam * ((IMGHGT/2)/sft2);

                        if (es.x> IMGWID || es.x<0 || es.y> IMGHGT || es.y<0) ;
                        else
                                imagedata3[(int)es.x][(int)es.y]= t;
                }

                if (1) for (int i= 0; i< nspacecrafts*nspacecrafts; i++){

                        COLORREF t = spacecraft[i].c;

                        vector es= (spacecraft[i].pos-origin+movevecone) * ((IMGHGT/2)/sft2) + vector
((IMGWID/2), (IMGHGT/2));
                        vector &les = spacecraft[i].lastes2;

                        if (es.x > IMGWID && les.x > IMGWID) ;
                        else if (es.x < 0 && les.x < 0) ;
                        else if (es.y > IMGHGT && les.y > IMGHGT) ;
                        else if (es.y < 0 && les.y < 0) ;
                        else if (spacecraft[i].lastes2.x != 0)
```

```
                        LineWrite(imagedata3, IMGWID, IMGHGT, spacecraft[i].lastes2.x,
spacecraft[i].lastes2.y, es.x, es.y, t, 1, 2);

                        spacecraft[i].lastes2 = es;

                }

/***********
earth in middle but with rotation, sf2
***********/
                for (int i= 0; i< nbody; i++){

                        COLORREF t = body[i].c;

                        vector es= (body[i].pos-body[earth].pos) * ((IMGHGT/2)/sf2) + vector
((IMGWID/2), (IMGHGT/2));
                        double diam = body[i].diam * ((IMGHGT/2)/sf2);

                        if (es.x> IMGWID || es.x<0 || es.y> IMGHGT || es.y<0) ;
                        else
                                imagedata4[(int)es.x][(int)es.y]= t;
                }

                for (int r= 0; r< nspacecrafts*nspacecrafts; r++){

                        COLORREF t = spacecraft[r].c;

                        vector es= (spacecraft[r].pos-body[earth].pos) * ((IMGHGT/2)/sf2) + vector
((IMGWID/2), (IMGHGT/2));

                        if (es.x> IMGWID || es.x<0 || es.y> IMGHGT || es.y<0) ;
                        else
                                imagedata4[(int)es.x][(int)es.y]= t;
                }

                if (1) for (int i= 0; i< nlagrange; i++){                       //sf1 center sun, for planets

                COLORREF tc;
                        if (i == 0) tc = RGB(204,204,24); //earth-moon lagrange L1
                        else if (i==1) tc= RGB(204,204,24); //earth-moon Lagrange L2
                        else if (i==2) tc= RGB(204,204,24); //sun-earth Lagrange L1
                        else if (i==3) tc= RGB(204,204,24); //sun-earth Lagrange L2
                        else if (i==4) tc= RGB(255,0,0); //to incluste the else if for more points later

                        vector es= (lagrange[i].pos-body[earth].pos) * ((IMGHGT/2)/sf2) + vector
((IMGWID/2), (IMGHGT/2));

                        if (es.x> IMGWID || es.x<0 || es.y> IMGHGT || es.y<0) ;
                        else
```

```
                    imagedata4[(int)es.x][(int)es.y]= tc;

        }


/***********
drawing the un-rotated version between the earth/moon. earth in middle, takes sf2
***********/
        if (1) for (int i= 0; i< nbody; i++){ //sf1 center earth
        vector draw = body[earth].pos-body[sun].pos;
        double theta= atan2(draw.y,draw.x);
        vector draw2 = body[i].pos- body[earth].pos;
        vector draw3= draw2.rotate(theta);

                COLORREF t = body[i].c;

                vector es= (draw3) * ((IMGHGT/2)/sf2) + vector ((IMGWID/2), (IMGHGT/2));
                double diam = body[i].diam * ((IMGHGT/2)/sf2);

                if (es.x> IMGWID || es.x<0 || es.y> IMGHGT || es.y<0) ;
                else
                        imagedata5[(int)es.x][(int)es.y]= t;
        }

        if (1) for (int i= 0; i< nspacecrafts*nspacecrafts; i++){ //sf1 center earth
        vector draw = body[earth].pos-body[moon].pos;
        double theta= atan2(draw.y,draw.x);
        vector draw2 = spacecraft[i].pos- body[earth].pos;
        vector draw3= draw2.rotate(theta);

                COLORREF t = spacecraft[i].c;

                vector es= (draw3) * ((IMGHGT/2)/sf2) + vector ((IMGWID/2), (IMGHGT/2));

                if (es.x> IMGWID || es.x<0 || es.y> IMGHGT || es.y<0) ;
                else
                        imagedata5[(int)es.x][(int)es.y]= t;
        }

/***********
drawing mars-centric picture, takes sf1
***********/
        if (1) for (int i= 0; i< nbody; i++) { //sf1 center earth
                vector draw2 = body[i].pos- body[sun].pos;

                COLORREF t = body[i].c;

                vector es= (draw2) * ((IMGHGT/2)/sf3) + vector ((IMGWID/2), (IMGHGT/2));
                double diam = body[i].diam * ((IMGHGT/2)/sf3);
```

```
                if (es.x> IMGWID || es.x<0 || es.y> IMGHGT || es.y<0) ;
                else
                        imagedata6[(int)es.x][(int)es.y]= t;
        }

        if (1) for (int i= 0; i< nspacecrafts*nspacecrafts; i++) { //sf1 center earth
                vector draw2 = spacecraft[i].pos- body[sun].pos;

                COLORREF t = spacecraft[i].c;

                vector es= (draw2) * ((IMGHGT/2)/sf3) + vector ((IMGWID/2), (IMGHGT/2));

                if (es.x> IMGWID || es.x<0 || es.y> IMGHGT || es.y<0) ;
                else
                        imagedata6[(int)es.x][(int)es.y]= t;
        }

/***********
drawing mars-centric picture, takes sf1
***********/
        if (1) for (int i= 0; i< nbody; i++) { //sf1 center earth
                vector draw2 = body[i].pos- body[sun].pos;

                COLORREF t = body[i].c;

                vector es= (draw2) * ((IMGHGT/2)/sf4) + vector ((IMGWID/2), (IMGHGT/2));
                double diam = body[moon].diam * ((IMGHGT/2)/sf4);

                if (es.x> IMGWID || es.x<0 || es.y> IMGHGT || es.y<0) ;
                else
                        imagedata7[(int)es.x][(int)es.y]= t;
        }

        if (1) for (int i= 0; i< nspacecrafts*nspacecrafts; i++) { //sf1 center earth
                vector draw2 = spacecraft[i].pos- body[sun].pos;

                COLORREF t = spacecraft[i].c;

                vector es= (draw2) * ((IMGHGT/2)/sf4) + vector ((IMGWID/2), (IMGHGT/2));

                if (es.x> IMGWID || es.x<0 || es.y> IMGHGT || es.y<0) ;
                else
                        imagedata7[(int)es.x][(int)es.y]= t;
        }
}

vector totalclosestpos;
int winningprobenum=0;
```

```
	if (1) for (int i= 0; i< (nspacecrafts*nspacecrafts); i++){
		if (norm(spacecraft[i].closestpos)>norm(totalclosestpos)){
			winningprobenum=i;
			totalclosestpos=spacecraft[i].closestpos;
		}
	}

	printf ("Best probe # %d with %f %f away \n", winningprobenum, totalclosestpos.x,
totalclosestpos.y);
	double jupiterpos= norm(body[sun].pos);
	double probepos= norm(spacecraft[0].pos);
	double probevec= norm(spacecraft[0].v);
	printf("jupiterpos= %f, probepos= %f, probevec= %f\n", jupiterpos, probepos, probevec);
	printf("probe velo x= %f, probe velo y %f\n", spacecraft[0].v.x, spacecraft[0].v.y);
	if (1) for (int i=0; i<(nspacecrafts*nspacecrafts); i++){
		printf ("method # %d with %f miss \n", spacecraft[i].method, spacecraft[i].miss);
	}

#define MAG 1
#if BMP != 0
			fprintf(index, "<img src=%s width=%d height=%d><p>\n",
ImageStoreAdd(IMGWID, IMGHGT, imagedata1), IMGWID*MAG, IMGHGT*MAG);
			fprintf(index, "<img src=%s width=%d height=%d><p>\n",
ImageStoreAdd(IMGWID, IMGHGT, imagedata2), IMGWID*MAG, IMGHGT*MAG);
			fprintf(index, "<img src=%s width=%d height=%d><p>\n",
ImageStoreAdd(IMGWID, IMGHGT, imagedata3), IMGWID*MAG, IMGHGT*MAG);
			fprintf(index, "<img src=%s width=%d height=%d><p>\n",
ImageStoreAdd(IMGWID, IMGHGT, imagedata4), IMGWID*MAG, IMGHGT*MAG);
			fprintf(index, "<img src=%s width=%d height=%d><p>\n",
ImageStoreAdd(IMGWID, IMGHGT, imagedata5), IMGWID*MAG, IMGHGT*MAG);
			fprintf(index, "<img src=%s width=%d height=%d><p>\n",
ImageStoreAdd(IMGWID, IMGHGT, imagedata6), IMGWID*MAG, IMGHGT*MAG);
			fprintf(index, "<img src=%s width=%d height=%d><p>\n",
ImageStoreAdd(IMGWID, IMGHGT, imagedata7), IMGWID*MAG, IMGHGT*MAG);
#endif
#if GIF != 0
//	want to put back in?		fprintf(index, "<img src=%s width=%d height=%d><br>\n",
GifImageStoreAdd(IMGWID, IMGHGT, imagedata1, imagedata2), IMGWID*2, IMGHGT*2);
//			fprintf(index, "<img src=%s width=%d height=%d><br>\n",
GifImageStoreAdd(IMGWID, IMGHGT, imagedata1, 0), IMGWID*2, IMGHGT*2);
//			fprintf(index, "<img src=%s width=%d height=%d><br>\n",
GifImageStoreAdd(IMGWID, IMGHGT, imagedata2, 0), IMGWID*2, IMGHGT*2);
#endif
			// end experimental code
			printf("\n");
//			free(img);

		fprintf(index, "</CENTER></CODE></body></html>\n");
		rewind(index);
```

```
                char buf[1234];
                int i;
                do {
                        i = fread(buf, 1, sizeof buf, index);
                        if (i > 0) send(theClient, buf, i, 0);
                } while (i == sizeof buf);

                fclose(index);
        }
```

Osim.h

```
void orbit(SOCKET theClient);
```

data.cpp
- calculation of ellipse

```
/**************************************************************************
 *   Copyright (C) 2007 by Brian Lott                                     *
 *   linuxcoder1@gmail.com                                                *
 *                                                                        *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   (at your option) any later version.                                  *
 *                                                                        *
 *   This program is distributed in the hope that it will be useful,       *
 *   but WITHOUT ANY WARRANTY; without even the implied warranty of        *
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the         *
 *   GNU General Public License for more details.                         *
 *                                                                        *
 *   You should have received a copy of the GNU General Public License     *
 *   along with this program; if not, write to the                        *
 *   Free Software Foundation, Inc.,                                      *
 *   59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.            *
 **************************************************************************/

/*  Under NO circumstances may the author be held responsable for his lame spelling
 *
 *  the equations in
 *  FindM, FindNu, Findr, Findx, Findy, Findz, FindL, FindMoonM, FindF, FindLAMBDA,
 *   FindBETA, FindDELTA, FindMoonX, FindMoonY, and FindMoonZ came from:
 *  http://www.astro.uu.nl/~strous/AA/en/reken/hemelpositie.html
 *   and
 *  http://www.astro.uu.nl/~strous/AA/en/reken/kepler.html
 *  and are owned by their original authors as is applicable
 */
```

```
//File: data.cpp
//Date: 1/22/2007
//Author: Brian Lott

#include <iostream>
#include <cstdlib>
#include <cmath>

using namespace std;

#include "data.h"

double PI = 3.141592653;  //for some odd reason, I can't use the library definition

/************************
data::data()
Passed nothing
Returns nothing

Used to define all the values we will need.  You want to define a value, do it in here.

note: the mass defined in here is not really any value.  It is merely a relational value.
************************/
data::data()
{
  dSubo = 36526.5;        //this is the only time we can change this
  NumberSteps = 0;        //Start our counter off

  Sun.x = 0;  //shouldn't change
  Sun.y = 0;  //touch this, and we all die (but it would be fun, wouldn't it)

  Mercury.a = 0.38710;
  Mercury.e = 0.20563;
  Mercury.i = 7.005;
  Mercury.omega = 29.125;
  Mercury.OMEGA = 48.331;
  Mercury.Mo = 174.795;
  Mercury.n = 4.092317;

  Venus.a = 0.72333;
  Venus.e = 0.00677;
  Venus.i = 3.395;
  Venus.omega = 54.884;
  Venus.OMEGA = 76.680;
  Venus.Mo = 50.416;
  Venus.n = 1.602136;

  Earth.a = 1.00000;
```

```
Earth.e = 0.01671;
Earth.i = 0.000;
Earth.omega = 288.064;
Earth.OMEGA = 174.873;
Earth.Mo = 357.529;
Earth.n = 0.985608;

Moon.L0 = 218.316;
Moon.L1 = 13.176396;
Moon.M0 = 134.963;
Moon.M1 = 13.064993;
Moon.F0 = 93.272;
Moon.F1 = 13.229350;

Mars.a = 1.52368;
Mars.e = 0.09340;
Mars.i = 1.850;
Mars.omega = 286.502;
Mars.OMEGA = 49.558;
Mars.Mo = 19.373;
Mars.n = 0.524039;

Jupiter.a = 5.20260;
Jupiter.e = 0.04849;
Jupiter.i = 1.303;
Jupiter.omega = 273.867;
Jupiter.OMEGA = 100.464;
Jupiter.Mo = 20.020;
Jupiter.n = 0.083056;

Saturn.a = 9.55491;
Saturn.e = 0.05551;
Saturn.i = 2.489;
Saturn.omega = 339.391;
Saturn.OMEGA = 113.666;
Saturn.Mo = 317.021;
Saturn.n = 0.033371;

Uranus.a = 19.21845;
Uranus.e = 0.04630;
Uranus.i = 0.773;
Uranus.omega = 98.999;
Uranus.OMEGA = 74.006;
Uranus.Mo = 141.050;
Uranus.n = 0.011698;

Neptune.a = 30.11039;
Neptune.e = 0.00899;
Neptune.i = 1.770;
```

```
  Neptune.omega = 276.340;
  Neptune.OMEGA = 131.784;
  Neptune.Mo = 256.225;
  Neptune.n = 0.005965;
}


/************************
data::calcData()
Passed nothing
Returns nothing

Used to calculate all the values.  If you want to calculate a value, then create a function
to do the math, and call it from here.  Neater that way.
************************/
void data::calcData()
{

  Mercury.M = FindM( Mercury.Mo, Mercury.n );
  Mercury.Nu = FindNu( Mercury.e, Mercury.M );
  Mercury.r = Findr( Mercury.a, Mercury.e, Mercury.Nu );
  Mercury.x = Findx( Mercury.r, Mercury.OMEGA, Mercury.omega, Mercury.Nu, Mercury.i );
  Mercury.y = Findy( Mercury.r, Mercury.OMEGA, Mercury.omega, Mercury.Nu, Mercury.i );

  Venus.M = FindM( Venus.Mo, Venus.n );
  Venus.Nu = FindNu( Venus.e, Venus.M );
  Venus.r = Findr( Venus.a, Venus.e, Venus.Nu );
  Venus.x = Findx( Venus.r, Venus.OMEGA, Venus.omega, Venus.Nu, Venus.i );
  Venus.y = Findy( Venus.r, Venus.OMEGA, Venus.omega, Venus.Nu, Venus.i );

  Earth.M = FindM( Earth.Mo, Earth.n );
  Earth.Nu = FindNu( Earth.e, Earth.M );
  Earth.r = Findr( Earth.a, Earth.e, Earth.Nu );
  Earth.x = Findx( Earth.r, Earth.OMEGA, Earth.omega, Earth.Nu, Earth.i );
  Earth.y = Findy( Earth.r, Earth.OMEGA, Earth.omega, Earth.Nu, Earth.i );

  Moon.L = FindL( Moon.L0, Moon.L1 );
  Moon.M = FindMoonM( Moon.M0, Moon.M1 );
  Moon.F = FindF( Moon.F0, Moon.F1);
  Moon.LAMBDA = FindLAMBDA( Moon.L, Moon.M );
  Moon.BETA = FindBETA( Moon.F );
  Moon.DELTA = FindDELTA( Moon.M );
  Moon.x = FindMoonX( Moon.DELTA, Moon.LAMBDA, Moon.BETA) + Earth.x;
  Moon.y = FindMoonY( Moon.DELTA, Moon.LAMBDA, Moon.BETA) + Earth.y;

  Mars.M = FindM( Mars.Mo, Mars.n );
  Mars.Nu = FindNu( Mars.e, Mars.M );
  Mars.r = Findr( Mars.a, Mars.e, Mars.Nu );
  Mars.x = Findx( Mars.r, Mars.OMEGA, Mars.omega, Mars.Nu, Mars.i );
```

```
  Mars.y = Findy( Mars.r, Mars.OMEGA, Mars.omega, Mars.Nu, Mars.i );

  Jupiter.M = FindM( Jupiter.Mo, Jupiter.n );
  Jupiter.Nu = FindNu( Jupiter.e, Jupiter.M );
  Jupiter.r = Findr( Jupiter.a, Jupiter.e, Jupiter.Nu );
  Jupiter.x = Findx( Jupiter.r, Jupiter.OMEGA, Jupiter.omega, Jupiter.Nu, Jupiter.i );
  Jupiter.y = Findy( Jupiter.r, Jupiter.OMEGA, Jupiter.omega, Jupiter.Nu, Jupiter.i );

  Saturn.M = FindM( Saturn.Mo, Saturn.n );
  Saturn.Nu = FindNu( Saturn.e, Saturn.M );
  Saturn.r = Findr( Saturn.a, Saturn.e, Saturn.Nu );
  Saturn.x = Findx( Saturn.r, Saturn.OMEGA, Saturn.omega, Saturn.Nu, Saturn.i );
  Saturn.y = Findy( Saturn.r, Saturn.OMEGA, Saturn.omega, Saturn.Nu, Saturn.i );

  Uranus.M = FindM( Uranus.Mo, Uranus.n );
  Uranus.Nu = FindNu( Uranus.e, Uranus.M );
  Uranus.r = Findr( Uranus.a, Uranus.e, Uranus.Nu );
  Uranus.x = Findx( Uranus.r, Uranus.OMEGA, Uranus.omega, Uranus.Nu, Uranus.i );
  Uranus.y = Findy( Uranus.r, Uranus.OMEGA, Uranus.omega, Uranus.Nu, Uranus.i );

  Neptune.M = FindM( Neptune.Mo, Neptune.n );
  Neptune.Nu = FindNu( Neptune.e, Neptune.M );
  Neptune.r = Findr( Neptune.a, Neptune.e, Neptune.Nu );
  Neptune.x = Findx( Neptune.r, Neptune.OMEGA, Neptune.omega, Neptune.Nu, Neptune.i );
  Neptune.y = Findy( Neptune.r, Neptune.OMEGA, Neptune.omega, Neptune.Nu, Neptune.i );

  ++NumberSteps;
}


/************************
data::outputAll()
Passed: nothing
Returns: nothing

Sends everything to the output stream.  In this case, that's a bunch of files.
************************/
void data::outputBodies( double *SunX, double *SunY,
                         double *MercuryX, double *MercuryY,
                         double *VenusX, double *VenusY,
                         double *EarthX, double *EarthY,
                         double *MoonX, double *MoonY,
                         double *MarsX, double *MarsY,
                         double *JupiterX, double *JupiterY,
                         double *SaturnX, double *SaturnY,
                         double *UranusX, double *UranusY,
                         double *NeptuneX, double *NeptuneY )
{
    *SunX = Sun.x*149597870691.0;                          //au*meters
```

```
   *SunY = Sun.y*149597870691.0;
   *MercuryX = Mercury.x*149597870691.0;
   *MercuryY = Mercury.y*149597870691.0;
   *VenusX = Venus.x*149597870691.0;
   *VenusY = Venus.y*149597870691.0;
   *EarthX = Earth.x*149597870691.0;
   *EarthY = Earth.y*149597870691.0;
   *MoonX = Moon.x*149597870691.0;
   *MoonY = Moon.y*149597870691.0;
   *MarsX = Mars.x*149597870691.0;
   *MarsY = Mars.y*149597870691.0;
   *JupiterX = Jupiter.x*149597870691.0;
   *JupiterY = Jupiter.y*149597870691.0;
   *SaturnX = Saturn.x*149597870691.0;
   *SaturnY = Saturn.y*149597870691.0;
   *UranusX = Uranus.x*149597870691.0;
   *UranusY = Uranus.y*149597870691.0;
   *NeptuneX = Neptune.x*149597870691.0;
   *NeptuneY = Neptune.y*149597870691.0;
}



/////
//Begin: Planet equations
/////

double data::FindM( double Mo, double n )
{
 double M = Mo + n * (d - dSubo);
 return M - 360 * (int)(M / 360);
}

double data::FindNu( double e, double M )
{
 double NuRadians;
 int counter = 1;
 double E[ 21 ];
 E[ 0 ] = M * PI / 180;
 E[ 1 ] = M * PI / 180 + e * sin(E[ 0 ]);

 while(counter <= 20 && fabs(E[ counter ] - E[ counter - 1 ]) > 0.000001)
 {
  ++counter;
  E[ counter ] = E[ 0 ] + e * sin(E[ counter - 1 ]);
 }
        //counter starts at 1 to avoid errors when comparing E[ n ] and E[ n-1 ]
        //limit 20 times through.  E should be stable by then.
```

```
  NuRadians = 2 * atan(sqrt(( 1 + e ) / ( 1 - e )) * tan(E[ counter ] / 2));
  if( NuRadians * 180 / PI < 0 )
    NuRadians = NuRadians * 180 / PI + 360;
  else
    NuRadians = NuRadians * 180 / PI;
    //simple  It's just finding the true anomaly( Nu ).  It gets more fun from here.  ;)
  return NuRadians;
}

double data::Findr( double a, double e, double Nu )
{
  return a * (1 - e * e) / (1 + e * cos((PI / 180) * Nu ));
}

//finds the x for a planet
double data::Findx( double r, double OMEGA, double omega, double Nu, double i )
{
  return r * (cos((PI / 180) * OMEGA) * cos((PI / 180) * (omega + Nu)) - sin((PI / 180) * OMEGA) *
cos((PI / 180) * i) * sin((PI / 180) * (omega + Nu)));
}

//finds the y for a planet
double data::Findy( double r, double OMEGA, double omega, double Nu, double i )
{
  return r * (sin((PI / 180) * OMEGA) * cos((PI / 180) * (omega + Nu)) + cos((PI / 180) * OMEGA) *
cos((PI / 180) * i) * sin((PI / 180) * (omega + Nu)));
}

/////
//End: Planet Eqution
//Start: functions for Moon
/////


double data::FindL( double L0, double L1 )
{
  double L = L0 + L1 * (d - dSubo);
  return L - 360 * (int)(L / 360);
}

double data::FindMoonM( double M0, double M1 )
{
  double M = M0 + M1 * (d - dSubo);
  return M - 360 * (int)(M / 360);
}

double data::FindF( double F0, double F1 )
{
  double F = F0 + F1 * (d - dSubo);
```

```
 return F - 360 * (int)(F / 360);
}

double data::FindLAMBDA( double L, double M )
{
 return L + 6.289 * sin((PI / 180) * M);
}

double data::FindBETA( double F )
{
 return 5.128 * sin((PI / 180) * F);
}

double data::FindDELTA( double M )
{
 return 385001 - 20905 * cos((PI / 180) * M );
}

double data::FindMoonX( double DELTA, double LAMBDA, double BETA )
{
 return DELTA * cos((PI / 180) * LAMBDA) * cos((PI / 180) * BETA) / 149598000;
}

double data::FindMoonY( double DELTA, double LAMBDA, double BETA )
{
 return DELTA * sin((PI / 180) * LAMBDA) * cos((PI / 180) * BETA) / 149598000;
}


/////
//End: functions for Moon
/////
```

data.h

```
/***************************************************************************
 *   Copyright (C) 2007 by Brian Lott                                      *
 *   linuxcoder1@gmail.com                                                 *
 *                                                                         *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                         *
 *   This program is distributed in the hope that it will be useful,       *
 *   but WITHOUT ANY WARRANTY; without even the implied warranty of        *
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the         *
 *   GNU General Public License for more details.                          *
 *                                                                         *
```

```
 *   You should have received a copy of the GNU General Public License     *
 *   along with this program; if not, write to the                          *
 *   Free Software Foundation, Inc.,                                        *
 *   59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.              *
 *************************************************************************/

//File: data.h
//Date: 1/22/2007
//Author: Brian Lott

#include <fstream>

#include "body.h"

#ifndef DATA_H
#define DATA_H

class data
{
  public:
    Body Sun;
    planet Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune;
       //yes, pluto is a planet in my book
    moon Moon;


    data();
    /*
    we will want to change d later.  d is the time.
    */
    double d;
    double dSubo;   //don't even dream of changing this.

    void calcData();    //calculates the X and Y
    void data::outputBodies( double *SunX, double *SunY,
                        double *MercuryX, double *MercuryY,
                        double *VenusX, double *VenusY,
                        double *EarthX, double *EarthY,
                        double *MoonX, double *MoonY,
                        double *MarsX, double *MarsY,
                        double *JupiterX, double *JupiterY,
                        double *SaturnX, double *SaturnY,
                        double *UranusX, double *UranusY,
                        double *NeptuneX, double *NeptuneY );    //outputs the Planet's X and Y
    void outputCounter();   //outputs the timestep counter

  private:
    //NOTE: I considered tossing most of the equations in the body class, but my original
    //  intention was to keep the body class stictly for storing the data, and the data
```

```
   //  class to handle all of the data, including the computations and the output.

   int NumberSteps;
   //The number of times that we calculate this stuff.
   //NOTE: we can't just use the counter variable we use for the loop in main().
   //      that works, unless we do a fraction of a day as our timestep.
   //      this is only used for the R program to display the current position of the planets

   //Start: Funtions for Planets
   double FindM( double Mo, double n );
   double FindNu( double e, double M );
   double Findr( double a, double e, double Nu );
   double Findx( double r, double OMEGA, double omega, double Nu, double i );
   double Findy( double r, double OMEGA, double omega, double Nu, double i );
   //End: Funtions for Planets

   //Start: Functions for Moon
   double FindL( double L0, double L1 );
   double FindMoonM( double M0, double M1 );
   double FindF( double F0, double F1 );
   double FindLAMBDA( double L, double M );
   double FindBETA( double F );
   double FindDELTA( double M );
   double FindMoonX( double DELTA, double LAMBDA, double BETA );
   double FindMoonY( double DELTA, double LAMBDA, double BETA );
   //End: Funtions for Moon
};

#endif
```

Appendix C – Email From Prof. Hut

From: Piet Hut [piet@ias.edu]
Sent: Saturday, March 31, 2007 7:14 PM
To: hilbert@swcp.com
Cc: makino@yso.mtk.nao.ac.jp; d.c.heggie@ed.ac.uk
Subject: Re: Partitioned Runge-Kutta Algorithms

Dear Kristin,

Thank you for spotting the error in our manuscript.
Yes, you are right. I am glad you found that typo.
I have added an acknowledgment with your name in the preface; it should appear in the next version of ACS, together with the corrected equation.

Actually, I am glad that we have made some typos, since in this way we can find out who are carefully reading our manuscripts! Most of the time, we have no idea who are reading the volues that we have written, and what they think about it.

Do you have any suggestions about improvements in the presentation, of this volume or others in the series?

I am aware of the fact that we never finished the last part of this volume; I keep intending to come back to it, but there have been too many other things catching my attention. I hope to finish it this summer.

By the way, there are two possibilities: either I copied this equation incorrectly from my notes, in which case the following equations should be correct; or I actually made a mistake in my calculations, in which case the equations following the one with a typo may be incorrect as well, based on the incorrect one that you pointed out. If you continue reading this chapter, can you let me know which one is the case?

Also, can you tell us something about the project that you are involved with?

Thank you again,

Piet

------- Start of forwarded message -------
From: "William Cordwell" <hilbert@swcp.com>
Date: Sat, 31 Mar 2007 13:32:23 -0600
Subject: Partitioned Runge-Kutta Algorithms
To: <piet@ias.edu>, <makino@yso.mtk.nao.ac.jp>, <d.c.heggie@ed.ac.uk>

Dear Professors Hut, Makino, and Heggie,

In The Kali Code, Volume 3, Integration Algorithms: Exploring the Runge-Kutta Landscape, Chapter 4, Partitioned Runge-Kutta Algorithms, after about 1.5 pages where

you express p_0 in terms of derivatives of f and v_0, I think that there may be a mistake:

The term inside the curly braces, 2f_0'(v_0)^2, I think it should be 5f_0'(v_0)^2, because you get a 3 from differentiating the middle term of c_0 and another 2 from differentiating the last term of c_0.

Would you please let me know if this is correct or if I'm misunderstanding something?

I am finding your paper very useful for a project I'm doing.

Thank you.

Sincerely,

Kristin Cordwell
------- End of forwarded message -------