A Simulation of an N-Body System Acting under Gravity

Samuel Crook, 38245361

2023-05-19

Abstract

The purpose of this project is to model the development of the Solar System using key properties of the planets in the initial frame, such as their position and velocity. This was achieved by calculating the effect of each individual planet on each other at a given time, letting these effects play out over a small time interval and iterating this system so that the forces between each planet was constantly updated to be more accurate to the current layout of the system. There were many different algorithms to choose from to iterate this system, such as the Euler method or the Euler-Cromer algorithm, each with their own uses and error margins. These systems were evaluated by investigating the order of their errors and how well they conserve other quantities such as total energy. For the given state of the solar system, it was found that the Euler-Richardson method was the most stable and accurate simulation according to the Jet Propulsion Laboratory's own simulation and results with a root mean square percent error of approximately 13.5%

Setting the Situation

To start, it is important to understand how particles interact in a stationary frame. According to Newton's law of gravitiation, the force exhibited on a particle with mass m_1 by a particle with mass m_2 is

$$\mathbf{F_g} = -\frac{Gm_1m_2}{r^2}\hat{\mathbf{e_r}},$$

where r is the distance from m_2 to m_1 and G is the gravitational constant. By Newton's second law $F = ma$, one can obtain the acceleration a of m_1 due to m_2's gravity:

$$\mathbf{a} = -\frac{Gm_2}{r^2}\hat{\mathbf{e_r}}$$

With this, one can calculate the resultant acceleration of a particle m_1 due to multiple particles $m_2, m_3, ..., m_n$ to be the sum of each individual acceleration.

$$\mathbf{g_1} = \sum_{i=2,3,\ldots,n} -\frac{Gm_i}{|r_{1,i}|^2}\mathbf{e}_{\hat{\mathbf{r_{1,i}}}}$$

where $r_{1,i}$ is the distance from m_i to m_1.

```r
# Calculates the effect particle 2 has on particle 1
grav_2_on_1 = function(m2, p1, p2, constant) {
    r = p1 - p2
    mag_r = norm(r, type = "2")
    unit_vec = r/mag_r
    g = -constant * m2 * unit_vec/(mag_r^2)
    return(g)
}
```

```
# Test of gravity on Earth's surface, (-9.81, 0, 0) is
# expected
grav_2_on_1(ME, c(RE, 0, 0), c(0, 0, 0), G)
```

```
## [1] -9.820138  0.000000  0.000000
```

```
# Takes the vector mass and positions of a set of particles
# and returns a vector of the resultant acceleration of
# each particles.  We will mostly be using dim=3
# (3-dimensional) and constant=G.
gravity_of_all = function(m, p, dim, constant) {
    l = length(m)
    gxyz = rep(0, l * dim)
    for (i in 1:l) {
        m_without = m[-i]
        p_without = p[-(((dim * i) - (dim - 1)):(dim * i))]
        p_cur = p[((dim * i) - (dim - 1)):(dim * i)]
        for (j in 1:(l - 1)) {
            gxyz[((dim * i) - (dim - 1)):(dim * i)] = gxyz[((dim *
                i) - (dim - 1)):(dim * i)] + grav_2_on_1(m_without[j],
                p_cur, p_without[((dim * j) - (dim - 1)):(dim *
                  j)], constant)
        }
    }
    return(gxyz)
}

testM = c(ME, 70)  #A test vector with the mass of the Earth and the mass of an average human
testP = c(0, 0, 0, RE, 0, 0)  #A test vector setting Earth at the origin and a human
# on the surface of Earth

# Earth should have a very low acceleration in the x-axis
# and the human should have an acceleration of -g in the
# x-axis
gravity_of_all(testM, testP, 3, G)
```

```
## [1]  1.151035e-22  0.000000e+00  0.000000e+00 -9.820138e+00  0.000000e+00
## [6]  0.000000e+00
```

```
# Returns the acceleration of each planet due to the other
# planets at the initial frame of reference
gravity_of_all(SSMass, SSP22, 3, G)
```

```
##  [1]  2.292295e-07 -8.886597e-08 -5.464666e-09  4.050257e-02 -3.335450e-03
##  [6] -3.987722e-03 -3.824756e-03  1.051323e-02  3.650289e-04  4.388804e-03
## [11]  3.853339e-03 -2.199400e-07 -1.709595e-03  2.498236e-03  9.429642e-05
## [16] -2.361952e-04  4.208833e-05  5.109358e-06 -4.529914e-05  4.041631e-05
## [21]  1.101802e-06 -1.091150e-05 -1.071605e-05  1.015532e-07 -6.585694e-06
## [26]  8.227313e-07  1.348223e-07
```

Now that the acceleration of each particle can be calculated, one can use this to calculate how this affects the particles' velocities and displacements. By defintion:

$$\mathbf{v}(t) = \mathbf{v_0} + \int \mathbf{a}(t)dt$$

$$\mathbf{x}(t) = \mathbf{x_0} + \int \mathbf{v}(t)dt$$

Since our calculation for acceleration is only at a discrete time and not over a continuous interval, the integrals will have to be approximated. By Taylor expansions, these variables can be found at a time $t + \Delta t$:

$$\mathbf{v}(t + \Delta t) = \mathbf{v}(t) + \mathbf{a}(t)\Delta t + \mathcal{O}((\Delta t)^2)$$

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \mathbf{v}(t)\Delta t + \mathcal{O}((\Delta t)^2)$$

By taking small enough time intervals Δt, these expressions can be approximated to simply

$$\mathbf{v}(t + \Delta t) \approx \mathbf{v}(t) + \mathbf{a}(t)\Delta t$$

$$\mathbf{x}(t + \Delta t) \approx \mathbf{x}(t) + \mathbf{v}(t)\Delta t$$

This is known as the Euler method and is the most simple of approximations. The error in this approximation is given by the term which was ignored $\mathcal{O}(\Delta t)^2$. If this process was to be repeated a total of n times over a constant timespan $T = N\Delta t$, then the cumulative error would be $\mathcal{O}(n(\Delta t)^2) = \mathcal{O}(\Delta t)$ An alternative approximation is the Euler-Cromer method, which considers the velocity at the end of the time interval for the calculation of displacement instead of at the start.

$$\mathbf{v}(t + \Delta t) \approx \mathbf{v}(t) + \mathbf{a}(t)\Delta t$$

$$\mathbf{x}(t + \Delta t) \approx \mathbf{x}(t) + \mathbf{v}(t + \Delta t)\Delta t$$

Similarly, this method has a cumulative error of $\mathcal{O}(\Delta t)$ due to the same reasons as the Euler method. However, this method tends to be much more accurate when modelling oscillating systems, including orbits, due to its ability to conserve total energy, unlike the Euler method. The proof of this will be given here[2]:

Taking the initial displacement and velocity of a particle to be x_0 and v_0 respectively, then by iterating the system using the equations described above total of n times over time intervals Δt, we get

$$v_n = (F_0 + F_1 + ... + F_{n-1})\Delta t = S_{n-1}$$

$$x_{n+1} = x_n + vn + 1\Delta t = x_0 + S_n \Delta t,$$

where $S_n = \Delta t \sum_{j=0}^{n} F_j$ and F is the force acting on the particle. Note that, if the regular Euler method was to be used, this last term would be S_{n-1} instead. The change in kinetic energy between $t_0 = 0$ and $t_n = n\Delta t$ is

$$\Delta K_n = K_n - K_0 = \frac{1}{2} S_{n-1}^2$$

The change in potential energy is given by

$$\Delta U_n = -\int_{x_0}^{x_n} F(x)dx$$

By splitting this integral into n different integrals each of width Δt, we can use the trapezoidal rule to obtain:

$$\Delta U_n = -\frac{1}{2} \sum_{i=0}^{n-1} (F_i + F_{i+1})(x_{i+1} - x_i)$$

$$= -\frac{1}{2} \Delta t \sum_{i=0}^{n-1} (F_i + F_{i+1}) S_i$$

$$= -\frac{1}{2} (\Delta t)^2 \sum_{i=0}^{n-1} \sum_{j=0}^{i} (F_i + F_{i+1}) F_j$$

$$= -\frac{1}{2}(\Delta t)^2(\sum_{i=0}^{n-1} F_i^2 + \sum_{i=0}^{n-1}\sum_{j=0}^{i-1} F_i F_j + \sum_{i=1}^{n}\sum_{j=0}^{i-1}(F_i F_j$$

$$= -\frac{1}{2}(\Delta t)^2(\sum_{i=0}^{n-1} F_i^2 + 2\sum_{i=0}^{n-1}\sum_{j=0}^{i-1} F_i F_j + F_n\sum_{j=0}^{i-1}(F_i F_j \tag{1}$$

$$= -\frac{1}{2}S_{n-1}^2 - \frac{1}{2}F_n S_{n-1}\Delta t$$

Thus:

$$\Delta E_n = \Delta K_n + \Delta U_n = -\frac{1}{2}F_n S_{n-1}\Delta t$$

$$= -\frac{1}{2}F_n v_n \Delta t$$

Since $v = 0$ when x reaches its maximum and minimum and $a = 0$ when the particle reaches the equilibrium point, there are four points in each oscillation that $\Delta E_n = 0$. Also, since F and v are bounded, ΔE_n must also be bounded. One can also calculate the average of ΔE_n over a half cycle to be

$$\langle \Delta E_n \rangle = \frac{D^2}{T}\sum_{n=0}^{\frac{T}{2D}} F_n v_n = \frac{D}{T}\int_0^{\frac{T}{2}} F v dt$$

$$= -\frac{D}{T}[U(\frac{T}{2}) - U(0)]$$

since U has the same value when the particle is at each turning point. Therefore, on average, the Euler-Cromer method conserves energy when considering oscillatory motion. It should be noted that, when the Euler method is used, the $\sum_{i=0}^{n-1} F_i^2$ term from 1 disappears, which means that

$$\Delta E_n = -\frac{1}{2}F_n v_n \Delta t - \sum_{i=0}^{n-1} F_i^2$$

This shows that there will always be some energy loss when a particle is affected by a force when using the Euler method.

Another method of approximation is the Euler-Richardson method, which calculates each iteration as follows

$$\mathbf{v}(t + \frac{1}{2}\Delta t) \approx \mathbf{v}(t) + \frac{1}{2}\mathbf{a}(t)\Delta t$$

$$\mathbf{x}(t + \frac{1}{2}\Delta t) \approx \mathbf{x}(t) + \frac{1}{2}\mathbf{v}(t)\Delta t$$

$$\mathbf{a}(t + \frac{1}{2}\Delta t) \approx \frac{\mathbf{F}(\mathbf{x}(t + \frac{1}{2}\Delta t), \mathbf{v}(t + \frac{1}{2}\Delta t), t + \frac{1}{2}\Delta t)}{m}$$

such that

$$\mathbf{v}(t + \Delta t) \approx \mathbf{v}(t) + \mathbf{a}(t + \frac{1}{2}\Delta t)\Delta t$$

$$\mathbf{x}(t + \Delta t) \approx \mathbf{x}(t) + \mathbf{v}(t + \frac{1}{2}\Delta t)\Delta t$$

This method can be derived by splitting the time interval into two[3]:

$$\mathbf{x_1}(t + \Delta t) = \mathbf{x}(t) + \mathbf{v}(t)\Delta t + \frac{1}{2}\mathbf{a}(t)(\Delta t)^2 + \mathcal{O}((\Delta t)^3) \tag{2}$$

Note that the subscript for x_1 is just notation which will become useful later on.

$$\mathbf{x}(t + \frac{1}{2}\Delta t) = \mathbf{x}(t) + \mathbf{v}(t)\frac{\Delta t}{2} + \frac{1}{2}\mathbf{a}(t)(\frac{\Delta t}{2})^2 + \mathcal{O}((\Delta t)^3)$$

$$\mathbf{x}(t+\Delta t) = \mathbf{x}(t+\frac{1}{2}\Delta t) + \mathbf{v}(t+\frac{1}{2}\Delta t)\frac{\Delta t}{2} + \frac{1}{2}\mathbf{a}(t+\frac{1}{2}\Delta t)(\frac{\Delta t}{2})^2 + \mathcal{O}((\Delta t)^3)$$

$$\mathbf{x}(t+\Delta t) = \mathbf{x}(t) + (\mathbf{v}(t+\frac{1}{2}\Delta t) + \mathbf{v}(t))\frac{\Delta t}{2} + \frac{1}{2}(\mathbf{a}(t+\frac{1}{2}\Delta t) + \mathbf{a}(t))(\frac{\Delta t}{2})^2 + \mathcal{O}((\Delta t)^3)$$

We have $\mathbf{a}(t+\frac{1}{2}\Delta t) \approx \mathbf{a} + a'(t)\frac{\Delta t}{2}$, so one can write

$$\mathbf{x_2}(t+\Delta t) = \mathbf{x}(t) + (\mathbf{v}(t+\frac{1}{2}\Delta t) + \mathbf{v}(t))\frac{\Delta t}{2} + \frac{1}{2}(2\mathbf{a}(t))(\frac{\Delta t}{2})^2 + \mathcal{O}((\Delta t)^3) \tag{3}$$

By combining 2 and 3, it is possible to show:

$$\mathbf{x}(t+\Delta t) = 2\mathbf{x_2}(t+\Delta t) - \mathbf{x_1}(t+\Delta t)$$

$$= 2[\mathbf{x}(t) + (\mathbf{v}(t+\frac{1}{2}\Delta t) + \mathbf{v}(t))\frac{\Delta t}{2} + \frac{1}{2}(2\mathbf{a}(t))(\frac{\Delta t}{2})^2] - [\mathbf{x}(t) + \mathbf{v}(t)\Delta t + \frac{1}{2}\mathbf{a}(t)(\Delta t)^2] + \mathcal{O}((\Delta t)^3)$$

$$= \mathbf{x}(t)[2-1] + \mathbf{v}(t)[\Delta t - \Delta t] + \mathbf{v}(t+\frac{1}{2}\Delta t)\Delta t + \mathbf{a}(t)[\frac{(\Delta t)^2}{2} - \frac{(\Delta t)^2}{2}] + \mathcal{O}((\Delta t)^3)$$

$$= \mathbf{x}(t) + \mathbf{v}(t+\frac{1}{2}\Delta t)\Delta t + \mathcal{O}((\Delta t)^3)$$

One can use a similar method to derive

$$\mathbf{v}(t+\frac{1}{2}\Delta t) = \mathbf{v}(t) + \mathbf{a}(t+\frac{1}{2}\Delta t)\Delta t + \mathcal{O}((\Delta t)^3)$$

Hence the Euler-Richardson method has an error of order $\mathcal{O}((\Delta t)^3)$. This suggests that this method should have less error than the other two methods, however it is important to note that, since this algorithm calculates acceleration twice for each time interval, this method will take approximately twice as long to compute since calculating the acceleration is the most time-heavy operation in these methods.

```r
# These next 3 blocks of code use their respective
# approximation to output a matrix with each position in
# the first row and each velocity in the second, for
# example: x1 y1 z1 x2 y2 z2 vx1 vy1 vz1 vx2 vy2 vz2
Euler_Method = function(m, x, v, delta, constant, dim) {
    g = gravity_of_all(m, x, dim, constant)
    x1 = x + (v * delta)
    v1 = v + (g * delta)
    return(matrix(c(x1, v1), nrow = 2, byrow = TRUE))
}
Euler_Cromer_Method = function(m, x, v, delta, constant, dim) {
    g = gravity_of_all(m, x, dim, constant)
    v1 = v + (g * delta)
    x1 = x + (v1 * delta)
    return(matrix(c(x1, v1), nrow = 2, byrow = TRUE))
}
Euler_Richardson_Method = function(m, x, v, delta, constant,
    dim) {
    g = gravity_of_all(m, x, dim, constant)
    vmid = v + (0.5 * g * delta)
    xmid = x + (0.5 * v * delta)
    gmid = gravity_of_all(m, xmid, dim, constant)
    x1 = x + (vmid * delta)
    v1 = v + (gmid * delta)
    return(matrix(c(x1, v1), nrow = 2, byrow = TRUE))
}
```

```r
# This code takes an input of the initial conditions of the
# system, the timespan to simulate for and the
# approximation to use. It will return a data frame
# consisting of the position and velocity of each particle
# at each time interval
TimeEvolutionSystem = function(m, x0, v0, t, n, constant, process,
    names) {
    delta = t/n
    l = length(m)
    dim = length(x0)/l
    X = V = matrix(rep(0, dim * l * (n + 1)), nrow = (n + 1))
    X[1, ] = x0
    V[1, ] = v0
    for (i in 1:n) {
        xv = process(m, X[i, ], V[i, ], delta, constant, dim)
        X[(i + 1), ] = xv[1, ]
        V[(i + 1), ] = xv[2, ]
    }
    Xdata = as.data.frame(X[, (1:dim)])
    Vdata = as.data.frame(V[, (1:dim)])
    for (i in 1:dim) {

        names(Xdata)[i] = paste(names[1], "x", i)
        names(Vdata)[i] = paste(names[1], "v", i)
    }
    XV = cbind(Xdata, Vdata)
    for (j in 2:l) {
        Xdata = as.data.frame(X[, (((dim * j) - (dim - 1)):(dim *
            j))])
        Vdata = as.data.frame(V[, (((dim * j) - (dim - 1)):(dim *
            j))])
        for (i in 1:dim) {
            names(Xdata)[i] = paste(names[j], "x", i)
            names(Vdata)[i] = paste(names[j], "v", i)
        }
        XV = cbind(XV, Xdata, Vdata)
    }
    return(XV)
}

# A simple test of a human skydiving on Earth from 4000m.
# This will not be too accurate as the simulation cannot
# account for resistance or terminal velocity, and the
# human would also fall through the surface of the Earth.
# However, it should be expected that the human accelerates
# towards the Earth at -g, whilst the Earth barely moves.
TimeEvolutionSystem(c(ME, 60), c(0, 0, 0, RE + 4000, 0, 0), c(0,
    0, 0, 0, 0, 0), 10, 10, G, Euler_Method, c("Earth", "Human"))
```

```
##       Earth x 1 Earth x 2 Earth x 3    Earth v 1 Earth v 2 Earth v 3 Human x 1
## 1  0.000000e+00         0         0 0.000000e+00         0         0   6375000
## 2  0.000000e+00         0         0 9.853638e-23         0         0   6375000
## 3  9.853638e-23         0         0 1.970728e-22         0         0   6374990
## 4  2.956091e-22         0         0 2.956094e-22         0         0   6374971
## 5  5.912186e-22         0         0 3.941467e-22         0         0   6374941
## 6  9.853653e-22         0         0 4.926849e-22         0         0   6374902
## 7  1.478050e-21         0         0 5.912243e-22         0         0   6374853
## 8  2.069275e-21         0         0 6.897653e-22         0         0   6374794
## 9  2.759040e-21         0         0 7.883080e-22         0         0   6374725
## 10 3.547348e-21         0         0 8.868529e-22         0         0   6374647
## 11 4.434201e-21         0         0 9.854002e-22         0         0   6374559
##    Human x 2 Human x 3  Human v 1 Human v 2 Human v 3
## 1          0         0   0.000000         0         0
## 2          0         0  -9.807818         0         0
## 3          0         0 -19.615637         0         0
## 4          0         0 -29.423485         0         0
## 5          0         0 -39.231394         0         0
## 6          0         0 -49.039394         0         0
## 7          0         0 -58.847514         0         0
## 8          0         0 -68.655785         0         0
## 9          0         0 -78.464237         0         0
## 10         0         0 -88.272901         0         0
## 11         0         0 -98.081806         0         0
```

Here, we see that the bodies accelerate at the expected rates, but the human does not remain at a constant terminal velocity of roughly $55ms^-1$[4]. A future development of this simulation may account for resistances and collisions, which would have occured if this timespan was increased. To test the algorithms in a more complex system, a simulation of the solar system from 02/05/22 to 02/05/23 was performed. A timespan of one Earth year was chosen as this would allow some diversity in the orbits of each planet, ranging from Mercury performing several orbits, to Neptune barely starting one. The system was intialised using data from the Jet Propulsion Laboratory Horizons page[1]. The code written to convert this information into usable vectors can be found in the Appendix. It is to be expected that the Euler approximation will be the least accurate of the three methods due to its error of order $\mathcal{O}((\Delta t)^2)$. The Euler-Richardson method should be more accurate than the Euler-Cromer method due to this reason as well, though the energy-conserving nature of the latter may result in it being more suitable for this simulation.

```
E_S_Time = Sys.time()
E_DF = TimeEvolutionSystem(SSMass, SSP22, SSV22, Year, 2000,
    G, Euler_Method, PlanetNames)
E_F_Time = Sys.time()
E_F_Time - E_S_Time  #Time taken to run the Euler method
```

```
## Time difference of 5.262274 secs
```

```
EC_S_Time = Sys.time()
EC_DF = TimeEvolutionSystem(SSMass, SSP22, SSV22, Year, 2000,
    G, Euler_Cromer_Method, PlanetNames)
EC_F_Time = Sys.time()
EC_F_Time - EC_S_Time  #Time taken to run the Euler-Cromer method
```

```
## Time difference of 5.166363 secs
```

```
ER_S_Time = Sys.time()
ER_DF = TimeEvolutionSystem(SSMass, SSP22, SSV22, Year, 2000,
    G, Euler_Richardson_Method, PlanetNames)
ER_F_Time = Sys.time()
ER_F_Time - ER_S_Time  #Time taken to run the Euler-Richardson method
```

```
## Time difference of 10.20405 secs
```

Before the data is evaluated, it is worth examining how long the computer took to run each algorithm. The Euler and Euler-Cromer algorithms both took roughly the same amount of time to compute, with the latter being slightly faster. However, as predicted, the Euler-Richardson took roughly twice as long to compute the same number of time intervals. This isn't too bad on the scale that this report is looking at, taking only an extra five seconds, but could add up if someone was to model a system with more particles or more time intervals. In order to fairly compare these algorithms, we will only consider the Euler-Richardson approximation with half the time intervals. This means that it will be possible to observe which algorithm works the most efficiently over the same time running.

```
ER_half_S_Time = Sys.time()
ER_DF = TimeEvolutionSystem(SSMass, SSP22, SSV22, Year, 1000,
    G, Euler_Richardson_Method, PlanetNames)
ER_half_F_Time = Sys.time()

# Time taken to run the Euler-Richardson method with half
# the intervals
ER_half_F_Time - ER_half_S_Time
```

```
## Time difference of 5.125399 secs
```

Evaluating the Data

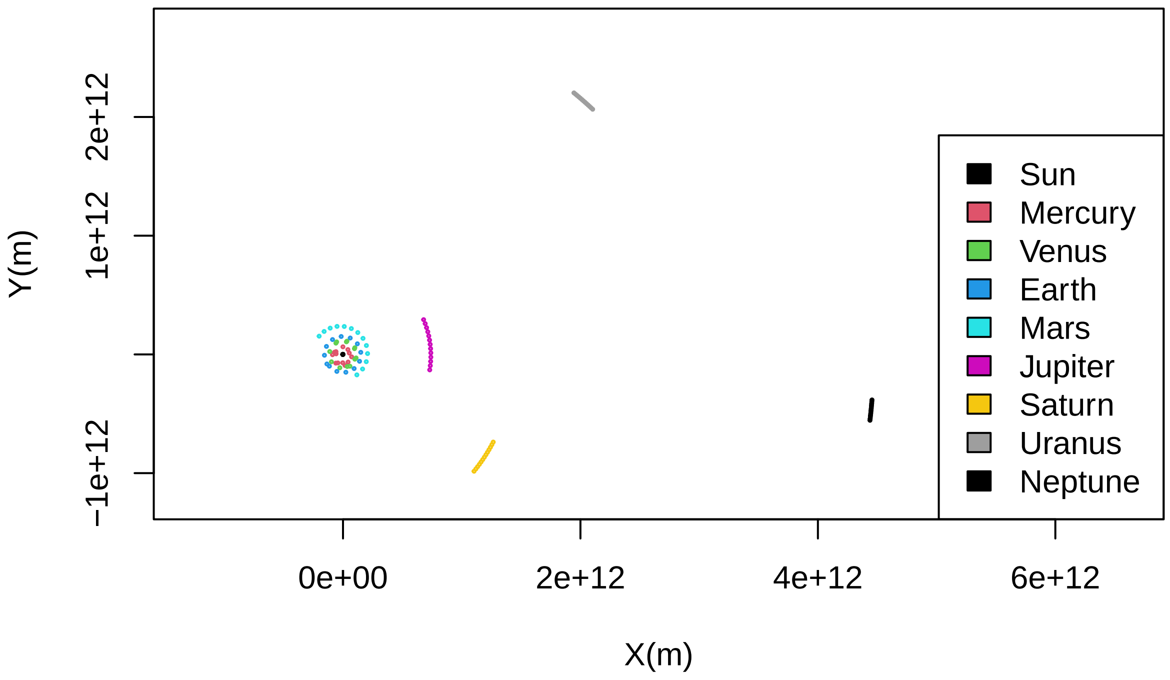
The outputs of the time evolution system are very large in size and may take a lot of time to evaluate, so some code to reduce these data frames down will come in handy. This will also help isolate the inner planets of the Solar System as these are the ones whose respective error will be greater since their velocity changes much more rapidly.

```
# This code can reduce a data frame about an entire system
# to only include data about specific particles involved.
# This way, it is possible to evaluate data about specific
# particles whilst including the effect caused by other
# particles in the system
LimitParticlesDF = function(DF, dim, limit) {
    veclimit = rep(0, (dim * 2 * length(limit)))
    for (i in 1:length(limit)) {
        veclimit[((1 + (i * 2 * dim) - (2 * dim)):(i * 2 * dim))] = (2 *
            dim * limit[i]) - seq((2 * dim) - 1, 0, by = -1)
    }
    NewDF = subset(DF, select = veclimit)
    return(NewDF)
}

# These data frames reduce the data frame to just the inner
# Solar System (Sun to Mars)
I_E_DF = LimitParticlesDF(E_DF, 3, (1:5))
I_EC_DF = LimitParticlesDF(EC_DF, 3, (1:5))
I_ER_DF = LimitParticlesDF(ER_DF, 3, (1:5))

# This code reduces the data frame to a select amount of
# evenly-spaced time intervals
LimitIterationsDF = function(DF, iterations) {
    interval = (nrow(DF) - 1)/iterations
    intseq = seq(1, by = interval, length = iterations + 1)
    intseq = round(intseq)
    NewDF = DF[intseq, ]
    return(NewDF)
}

# Here, the time intervals are reduced to being one month
# apart
R_E_DF = LimitIterationsDF(E_DF, 12)
R_EC_DF = LimitIterationsDF(EC_DF, 12)
R_ER_DF = LimitIterationsDF(ER_DF, 12)
```

Since the data frames hold the positions of each planet at each time interval, it is possible to plot the path of each planet on a graph. Since the majority of displacement occurs in the x-y plane, it is best to plot this and leave out the z-axis for now.

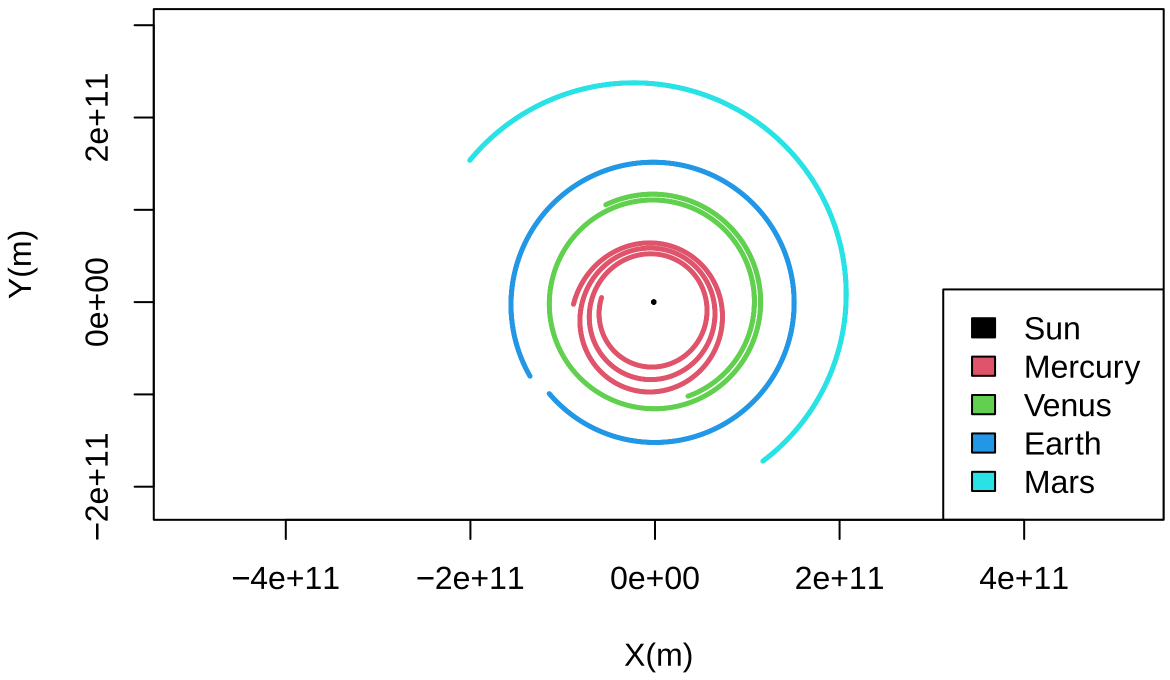
```
# This code plots a data frame with the desired
# coordinates. We will use dimplot = c(1,2) to plot the
# first coordinate (x) against the second (y)
DataFramePlot = function(DF, dim, dimplot, title, xlabel, ylabel) {
    nparticle = ncol(DF)/(2 * dim)  #How many particles in the system
    maxx = minx = DF[1, dimplot[1]]  #Helps to set the size of the graph
    maxy = miny = DF[1, dimplot[2]]
    particlenames = "null"
    for (i in 1:nparticle) {
        maxx = max(maxx, DF[, (i * 2 * dim) - ((2 * dim) - dimplot[1])])
        maxy = max(maxy, DF[, (i * 2 * dim) - ((2 * dim) - dimplot[2])])
        minx = min(minx, DF[, (i * 2 * dim) - ((2 * dim) - dimplot[1])])
        miny = min(miny, DF[, (i * 2 * dim) - ((2 * dim) - dimplot[2])])
        name = gsub("x 1", "", names(DF)[(i * 2 * dim) - ((2 *
            dim) - dimplot[1])])
        particlenames = c(particlenames, name)
    }
    particlenames = particlenames[-1]
    plot(DF[, dimplot[1]], DF[, dimplot[2]], type = "b", xlim = c((1.25 *
        minx), (1.25 * maxx)), ylim = c((1.25 * miny), (1.25 *
        maxy)), main = title, xlab = xlabel, ylab = ylabel, asp = 1,
        cex = 0.2, cex.main = 0.75)
    for (i in 2:nparticle) {
        lines(DF[, (i * 2 * dim) - ((2 * dim) - dimplot[1])],
            DF[, (i * 2 * dim) - ((2 * dim) - dimplot[2])], type = "b",
            col = i, cex = 0.2)
    }

    legend("bottomright", legend = particlenames, fill = (1:nparticle))
}
DataFramePlot(R_E_DF, 3, c(1, 2), "Simulation of the Solar System over an Earth year using the Euler me
    "X(m)", "Y(m)")
```

Simulation of the Solar System over an Earth year using the Euler method



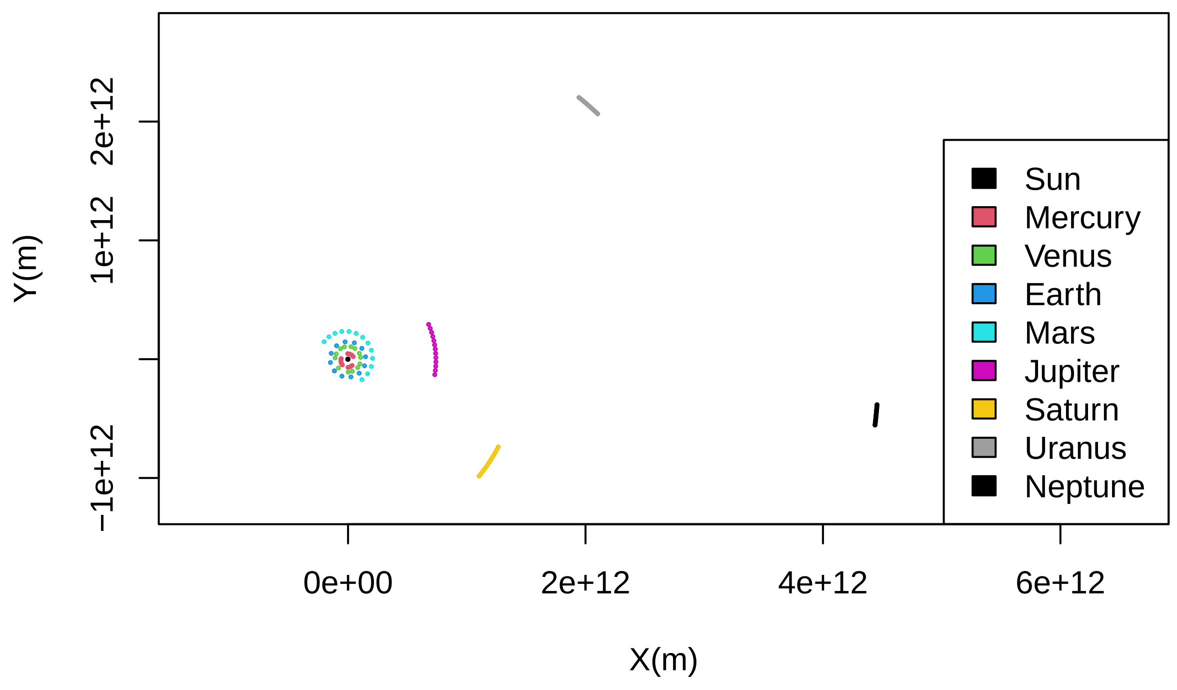
```
DataFramePlot(I_E_DF, 3, c(1, 2), "Simulation of the Inner Solar System over an Earth year using the Eul
    "X(m)", "Y(m)")
```

Simulation of the Inner Solar System over an Earth year using the Euler method



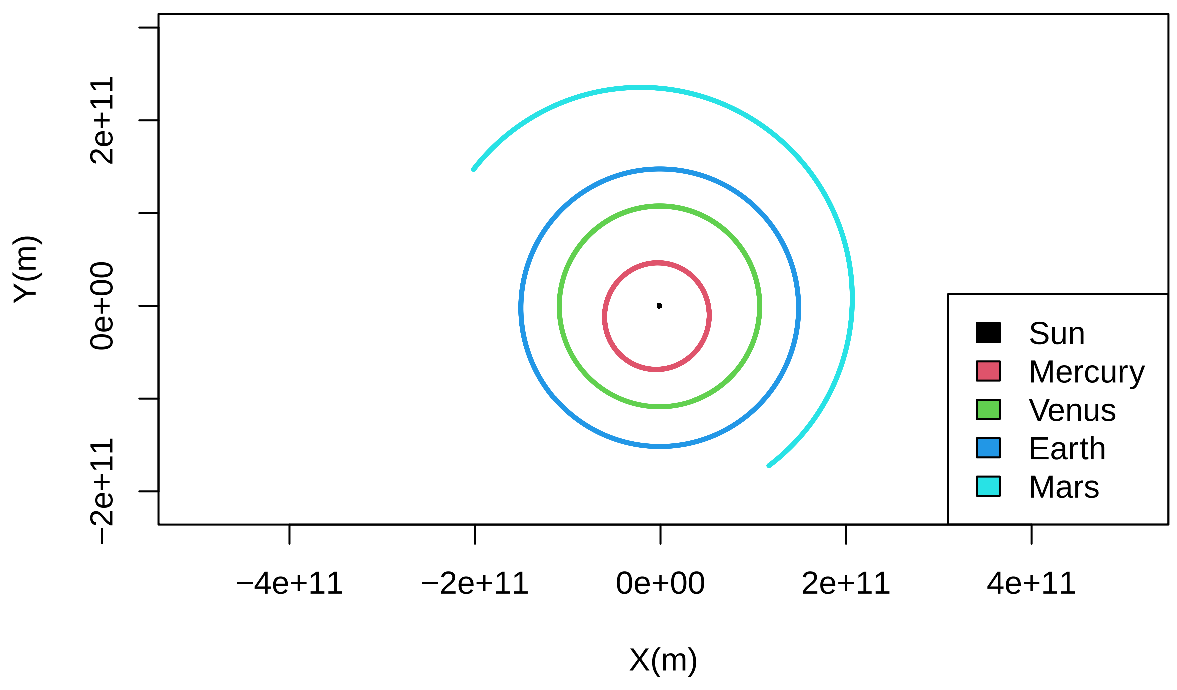
```
DataFramePlot(R_EC_DF, 3, c(1, 2), "Simulation of the Solar System over an Earth year using the Euler-C
    "X(m)", "Y(m)")
```

Simulation of the Solar System over an Earth year using the Euler−Cromer method

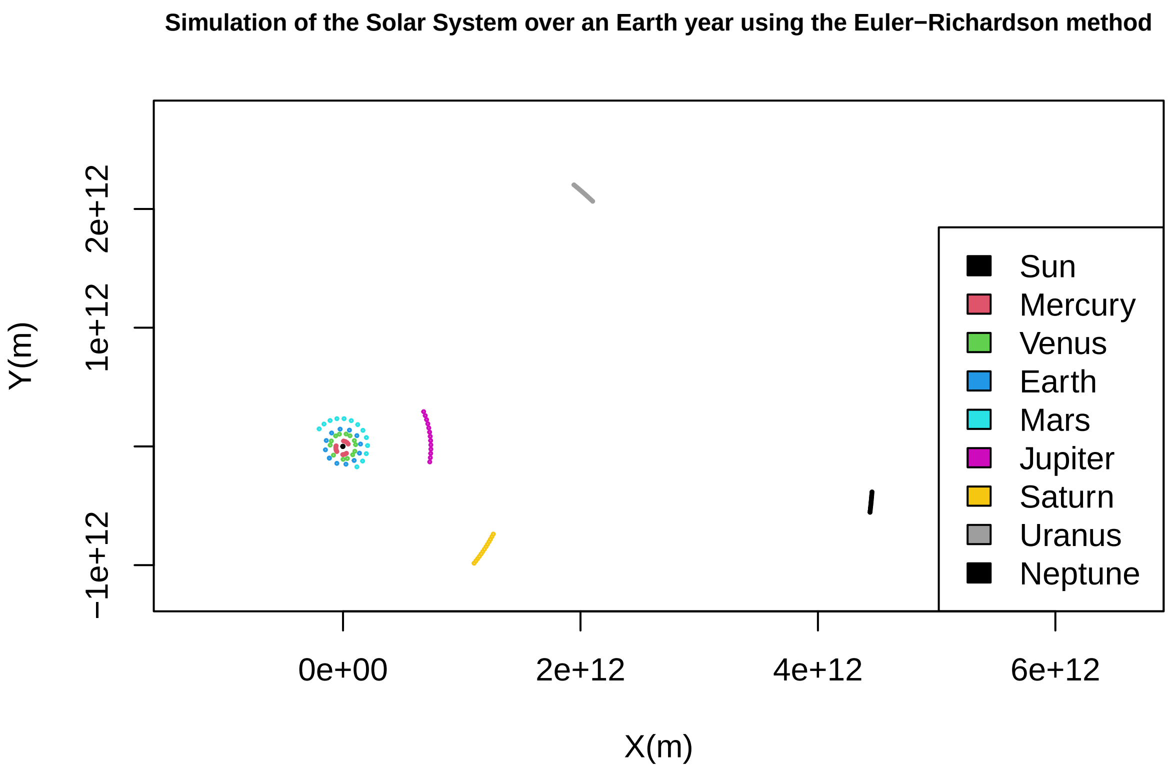


```
DataFramePlot(I_EC_DF, 3, c(1, 2), "Simulation of the Inner Solar System over an Earth year using the Eu
    "X(m)", "Y(m)")
```

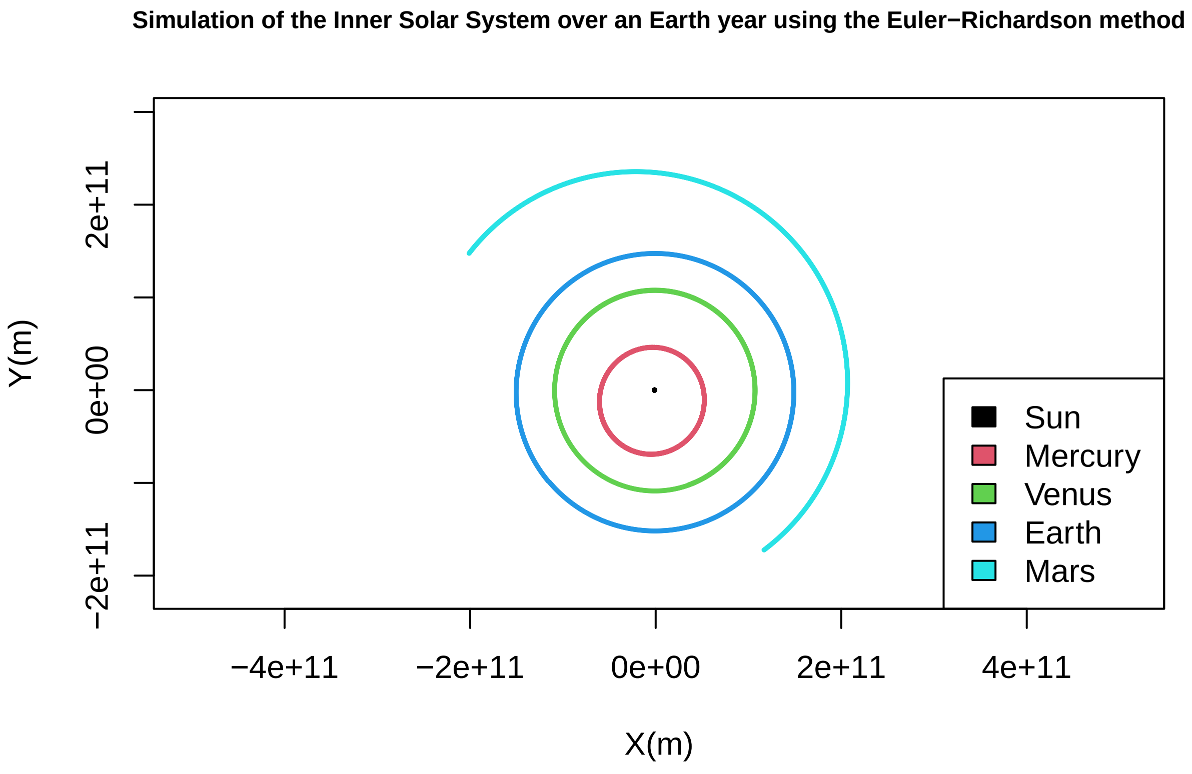
Simulation of the Inner Solar System over an Earth year using the Euler−Cromer method



```
DataFramePlot(R_ER_DF, 3, c(1, 2), "Simulation of the Solar System over an Earth year using the Euler-Ri
    "X(m)", "Y(m)")
```



```
DataFramePlot(I_ER_DF, 3, c(1, 2), "Simulation of the Inner Solar System over an Earth year using the Eu
    "X(m)", "Y(m)")
```



It can be observed from these graphs that both the Euler-Cromer and the Euler-Richardson methods produce stable orbits. However, for the Euler method, the planets seem to spiral, which should not be expected over just one year. To test if the orbits generated follow the laws of physics, one can apply the law of conservation of momentum and the law of conservation of energy. It should be expected that these quantities remain roughly constant throughout the system, though since there are approximations involved then slight deviations should be anticipated.

```
# Checks the total linear momentum in each coordinate
L_Momentum_Conservation = function(DF, m, dim) {
    for (i in 1:dim) {
        sequence = seq(((2 * dim) - (dim - i)), by = (2 * dim),
            length = length(m))
        for (j in 1:nrow(DF)) {
            DF[j, i] = sum(DF[j, sequence] * m)
        }
        names(DF)[i] = paste("Momentum X", i)
    }
    DF = DF[1:dim]
    return(DF)
}

# Total Kinetic Energy of the System
Kinetic_Energy = function(DF, m, dim) {
    K = rep(0, nrow(DF))
    nparticles = ncol(DF)/(2 * dim)
    V2 = rep(0, nparticles)
    for (i in 1:nrow(DF)) {
        for (j in 1:nparticles) {
            V2[j] = sum((DF[i, (((2 * dim * j) - (dim - 1)):(2 *
                dim * j))])^2)
        }
        K[i] = sum(0.5 * m * V2)
    }
    return(K)
}

# Total Potential Energy of the System
Potential_Energy = function(DF, m, dim, constant) {
    U = rep(0, nrow(DF))
    nparticles = ncol(DF)/(2 * dim)
    for (i in 1:nrow(DF)) {
        Ucur = rep(0, dim)
        for (j in (1:nparticles)[-nparticles]) {
            for (k in ((j + 1):nparticles)) {
                r = (DF[i, (((2 * dim * j) - ((2 * dim) - 1)):((2 *
                  dim * j) - (2 * dim)))] - DF[i, (((2 * dim *
                  k) - ((2 * dim) - 1)):((2 * dim * k) - (2 *
                  dim)))])
                Ucur = Ucur + 2 * constant * m[j] * m[k]/sum(abs(r))
            }
        }
        U[i] = sum(Ucur)
    }
    return(U)
}
Total_Energy = function(DF, m, dim, constant) {
    K = Kinetic_Energy(DF, m, dim)
    U = Potential_Energy(DF, m, dim, constant)
    Total = K + U
    return(Total)
}
L_Momentum_Conservation(R_E_DF, SSMass, 3)  #Total linear momentum of E method
```

```
##      Momentum X 1  Momentum X 2 Momentum X 3
## 1    -4.80457e+27 -4.797624e+27 4.338577e+26
## 168  -4.80457e+27 -4.797624e+27 4.338577e+26
## 334  -4.80457e+27 -4.797624e+27 4.338577e+26
## 501  -4.80457e+27 -4.797624e+27 4.338577e+26
## 668  -4.80457e+27 -4.797624e+27 4.338577e+26
## 834  -4.80457e+27 -4.797624e+27 4.338577e+26
## 1001 -4.80457e+27 -4.797624e+27 4.338577e+26
## 1168 -4.80457e+27 -4.797624e+27 4.338577e+26
## 1334 -4.80457e+27 -4.797624e+27 4.338577e+26
## 1501 -4.80457e+27 -4.797624e+27 4.338577e+26
## 1668 -4.80457e+27 -4.797624e+27 4.338577e+26
## 1834 -4.80457e+27 -4.797624e+27 4.338577e+26
## 2001 -4.80457e+27 -4.797624e+27 4.338577e+26
```

```
L_Momentum_Conservation(R_ER_DF, SSMass, 3)  #Total linear momentum of EC method
```

```
##      Momentum X 1  Momentum X 2 Momentum X 3
## 1    -4.80457e+27 -4.797624e+27 4.338577e+26
## 84   -4.80457e+27 -4.797624e+27 4.338577e+26
## 168  -4.80457e+27 -4.797624e+27 4.338577e+26
## 251  -4.80457e+27 -4.797624e+27 4.338577e+26
## 334  -4.80457e+27 -4.797624e+27 4.338577e+26
## 418  -4.80457e+27 -4.797624e+27 4.338577e+26
## 501  -4.80457e+27 -4.797624e+27 4.338577e+26
## 584  -4.80457e+27 -4.797624e+27 4.338577e+26
## 668  -4.80457e+27 -4.797624e+27 4.338577e+26
## 751  -4.80457e+27 -4.797624e+27 4.338577e+26
## 834  -4.80457e+27 -4.797624e+27 4.338577e+26
## 918  -4.80457e+27 -4.797624e+27 4.338577e+26
## 1001 -4.80457e+27 -4.797624e+27 4.338577e+26
```

```
L_Momentum_Conservation(R_ER_DF, SSMass, 3)  #Total linear momentum of ER method
```

```
##      Momentum X 1  Momentum X 2 Momentum X 3
## 1    -4.80457e+27 -4.797624e+27 4.338577e+26
## 84   -4.80457e+27 -4.797624e+27 4.338577e+26
## 168  -4.80457e+27 -4.797624e+27 4.338577e+26
## 251  -4.80457e+27 -4.797624e+27 4.338577e+26
## 334  -4.80457e+27 -4.797624e+27 4.338577e+26
## 418  -4.80457e+27 -4.797624e+27 4.338577e+26
## 501  -4.80457e+27 -4.797624e+27 4.338577e+26
## 584  -4.80457e+27 -4.797624e+27 4.338577e+26
## 668  -4.80457e+27 -4.797624e+27 4.338577e+26
## 751  -4.80457e+27 -4.797624e+27 4.338577e+26
## 834  -4.80457e+27 -4.797624e+27 4.338577e+26
## 918  -4.80457e+27 -4.797624e+27 4.338577e+26
## 1001 -4.80457e+27 -4.797624e+27 4.338577e+26
```

```
Total_Energy(R_E_DF, SSMass, 3, G)  #Evolution of total energy in Euler method
```

```
##  [1] 2.884801e+36 2.976027e+36 2.925756e+36 2.872354e+36 2.795997e+36
##  [6] 2.764156e+36 2.783436e+36 2.936431e+36 3.142181e+36 2.853294e+36
## [11] 2.848926e+36 3.066206e+36 2.953454e+36
```

```
Total_Energy(R_EC_DF, SSMass, 3, G)  #Evolution of total energy in EC method
```

```
##  [1] 2.884801e+36 2.960557e+36 2.926358e+36 2.889267e+36 2.804319e+36
##  [6] 2.768261e+36 2.792231e+36 3.732219e+36 3.019683e+36 2.842323e+36
## [11] 2.879311e+36 3.060078e+36 2.929180e+36
```

```
Total_Energy(R_ER_DF, SSMass, 3, G)  #Evolution of total energy in ER method
```

```
##  [1] 2.884801e+36 2.972980e+36 2.924279e+36 2.884307e+36 2.810839e+36
##  [6] 2.767675e+36 2.791814e+36 3.892776e+36 3.017521e+36 2.842283e+36
## [11] 2.878860e+36 3.001276e+36 2.928746e+36
```

As can be seen, each method conserves linear momentum to great accuracy. However, with the conservation of energy, there appears to be an increase in energy in all systems over time. The Euler method has the greatest discrepancy at an increase of around $6.8653 \times 10^{34} J$ over an Earth year, whereas the Euler-Cromer and Euler-Richardson methods have discrepancies of roughly $4.4379 \times 10^{34} J$ and $4.3945 \times 10^{34} J$. This information suggests that the Euler-Richardson approximation is slightly more accurate than the Euler-Cromer method, despite only using half as many time intervals. With this information, we can now say that the simulation respects the laws of physics to a high-enough degree.

Now that it is important to compare the calculated positions and velocities to the actual positions and velocities.

```
# Finds the relative error between the experimental values
# DF1 and the accepted values DF2
relative_error = function(DF1, DF2) {
    final_result = DF1[nrow(DF1), ]
    error = (final_result - DF2)/DF2
    return(error)
}

# Finds the root mean square percent error for each
# particle
RMSPE_particles = function(error, dim) {
    nparticles = length(error)/(dim * 2)
    mat = matrix(rep(0, nparticles * 2), nrow = 1)
    ans = as.data.frame(mat)
    for (i in 1:(nparticles * 2)) {
        ans[, i] = RMSPE_system(error[(((dim * i) - (dim - 1)):(dim *
            i))])
        name = names(error)[((dim * i) - (dim - 1))]
        name = gsub(" 1", "", name)
        names(ans)[i] = name
    }
    return(ans)
}

# Finds the RMSPE of the whole system
RMSPE_system = function(error) {
    error = error^2
    sum_error = sum(error)
    mean = sum_error/length(error)
    root_mean = 100 * sqrt(mean)
    return(root_mean)
}
```

```
E_RE = relative_error(E_DF, SSPV23)  #Relative error in the Euler Method
RMSPE_system(E_RE)  #Root mean square percentage error in the Euler Method
```

```
## [1] 101.9543
```

```
RMSPE_particles(E_RE, 3)  #RMSPE for each particle in the Euler Method
```

```
##        Sun x    Sun v Mercury x Mercury v  Venus x  Venus v  Earth x Earth v
## 1 0.08272872 6.165936  375.9778  88.12419 82.11797 163.9943 14.98996 54.0859
##     Mars x   Mars v  Jupiter x  Jupiter v   Saturn x   Saturn v    Uranus x
## 1 2.486724 33.87052 0.01213533 0.04632482 0.02817715 0.05374344 0.001502439
##     Uranus v  Neptune x Neptune v
## 1 0.03178501 0.01149674 0.2432707
```

```
EC_RE = relative_error(EC_DF, SSPV23)  #Relative error in the Euler-Cromer Method
RMSPE_system(EC_RE)  #RMSPE in the Euler-Cromer Method
```

```
## [1] 16.18132
```

```
RMSPE_particles(EC_RE, 3)  #RMSPE for each particle in the Euler-Cromer Method
```

```
##       Sun x     Sun v Mercury x Mercury v  Venus x Venus v   Earth x  Earth v
## 1 0.0073188 0.1430134  36.43045  9.095152 1.176827 1.71101 0.7090347 57.39626
##      Mars x   Mars v  Jupiter x Jupiter v   Saturn x   Saturn v    Uranus x
## 1 0.2716816 1.955692 0.01497915 0.0688224 0.02914171 0.05384373 0.001318042
##     Uranus v  Neptune x Neptune v
## 1 0.03177943 0.01145047 0.2432617
```

```
ER_RE = relative_error(ER_DF, SSPV23)  #Relative error in the Euler-Richardson Method
RMSPE_system(ER_RE)  #RMSPE in the Euler-Richardson Method
```

```
## [1] 13.61049
```

```
RMSPE_particles(ER_RE, 3)  #RMSPE for each particle in the Euler-Richardson Method
```

```
##        Sun x      Sun v Mercury x Mercury v    Venus x    Venus v   Earth x
## 1 0.01048434 0.07811972  6.282247  1.363586 0.03905006 0.08784362 0.7466338
##    Earth v     Mars x    Mars v  Jupiter x Jupiter v   Saturn x   Saturn v
## 1 57.37869 0.02849018 0.3717993 0.00847475 0.0589488 0.02874499 0.05439373
##      Uranus x   Uranus v  Neptune x Neptune v
## 1 0.001412721 0.03185588 0.01147346 0.2432337
```

It can be seen that the Euler method has a very high error for the inner planets, though manages to model the outer planets fairly well, even being marginally more accurate than the other methods in some of the measurements. This may be because the timespan chosen results in the outer planets travelling in a relatively straight line, which means that the Euler method is more accurate than the Euler-Cromer method more tended towards oscillations and the Euler-Richardson method which has larger time intervals. The Euler-Cromer method is relatively accurate for most measurements and manages to reduce some of the errors of the Euler method significantly, though Mercury still has a somewhat high error. As can be seen, the Euler-Richardson method has the lowest RMSPE at around 13.5%. However, most of this error originates specifically from the z-velocity of Earth which has a percent error of -99%.

```
100 * ER_RE[24]
```

```
##      Earth v 3
## 1001 -99.37672
```

This error appears in the other methods as well. After ensuring the initial conditions were correct, it was found that after one calendar year, the z-velocity of Earth can vary[1] whereas the other properties tend to perform a full cycle and stay the same, which is what would be expected. On the scale of the Solar System, the z-component of the Earth's velocity is almost negligible, being less than 10,000 times the size of its x-y velocity. However, due to its small size, this can allow for discrepancies in the calculations to cause the z-velocity to change by a large percent, even if the actual value is very small.

Conclusion

A simulation was developed which could model any N-body gravitational system in any dimension. This simulation could plot the path taken by the particles and track the conservation of key quantities such as the total energy of the system. It was found that the system could accurately track the path of bodies in which gravity was the only force, though failed when resistances were involved. The simulation could easily track point-like particles such as planets in the Solar-System with great accuracy. Approximations for position and velocity were evaluated in several different tests in the Solar System model. It was found that the Euler-Richardson method was the most accurate approximation for the Solar System despite only being able to calculate half as many time intervals as the others as it had only a 13.52864% root square percent mean error when compared to the actual traits of the system after one year.

There is plenty of room for development in this code for more complex systems. The system is designed to model particles interacting under gravity, though this is not the only concept it could model. Coulomb's Law states that the force acting on a particle with charge q_1 and mass m_1 by a particle with charge q_2 and mass m_2 is

$$\mathbf{F_E} = \frac{kq_1q_2}{r^2}\hat{\mathbf{e_r}},$$

where k is the constant $\frac{1}{4\pi\epsilon_0}$. By using Newton's second law, it is possible to show the acceleration of the first particle to be

$$\mathbf{F_E} = \frac{kq_1q_2}{m_1r^2}\hat{\mathbf{e_r}},$$

Therefore, it would be possible to write code which calculates this force and acceleration and develop the system described in this report to account for this electric force as well as the gravity between them. This was already partially written into the code This would also allow the particles to repel each other, something which is not possible with gravity alone, allowing for more complex systems.

One of the assumptions made for the model of the Solar System was that each planet could be treated as a point-like particle. This works well with the sheer size of the Solar System compared to the planets, but does not work too well when considering particles which come close to each other as collisions are not anticipated in this code. This also may be a downside when considering code which deals with charge, as this system would be unable to model wires or charged planes which are commonly looked at in electrodynamics. Investigating these concepts would be a great starting point to develop the code written here.

References

[1] Giorgini, J. D., Yeomans, D. K., Chamberlin, A. B., Chodas, P. W., Jacobson, R. A., Keesey, M. S., Lieske, J. H., Ostro, S. J., Standish, E. M., Wimberly, R. N., "JPL's On-Line Solar System Data Service", Bulletin of the American Astronomical Society, Vol 28, p. 1099, 1997.

[2] Cromer, A., "Stable solutions using the Euler approximation", American Journal of Physics, Vol 49, Issue 5, pp. 455-459, 1 May 1981

[3] Gould H., Tobochnik, J., Christian, W., "An Introduction to Computer Simulation Methods: Applications to Physical Systems", Chapter 3, pp. 78-79, 1988

[4] FAI Skydiving Commission (ISC), https://www.fai.org/page/isc-speed-skydiving

Appendix - Converting JPL data to usable vectors

This is a look at the code which sets some of the variables and constants that were used in the simulation.

```
# 2460066.500000000 = A.D. 2023-May-02 00:00:00.0000 TDB
sun23 = "X =-1.324316846741195E+06 Y =-1.446765512621427E+05 Z = 3.202738040537933E+04
 VX= 4.021534124461124E-03 VY=-1.472929900704240E-02 VZ= 2.948517133131729E-05"
mer23 = "X =-5.140210130150759E+07 Y =-4.381189053847280E+07 Z = 1.056887671139676E+06
 VX= 2.207720296247787E+01 VY=-3.453839718779295E+01 VZ=-4.845914556700924E+00"
ven23 = "X =-9.848082136897813E+07 Y = 4.552713536799312E+07 Z = 6.265256376415793E+06
 VX=-1.504882769689358E+01 VY=-3.187376111911025E+01 VZ= 4.312134646762669E-01"
ear23 = "X =-1.150223276469004E+08 Y =-9.909841274895248E+07 Z = 3.755860068777204E+04
 VX= 1.907614272241964E+01 VY=-2.258721645907844E+01 VZ= 2.018298546438757E-03"
mar23 = "X =-2.010819716042931E+08 Y = 1.476265433762218E+08 Z = 8.029003272268474E+06
 VX=-1.349413909238045E+01 VY=-1.742820367074673E+01 VZ=-3.381872148509668E-02"
jup23 = "X = 6.792848124845287E+08 Y = 2.926757289590889E+08 Z =-1.641170746619791E+07
 VX=-5.318068089405075E+00 VY= 1.261385267190711E+01 VZ= 6.658478320447347E-02"
sat23 = "X = 1.265581460039990E+09 Y =-7.382676489447224E+08 Z =-3.755222856885275E+07
 VX= 4.325577300311080E+00 VY= 8.324810463771845E+00 VZ=-3.167483277652900E-01"
ura23 = "X = 1.945406877724633E+09 Y = 2.202709703800068E+09 Z =-1.702215196928215E+07
 VX=-5.154055480935001E+00 VY= 4.190735053981157E+00 VZ= 8.237035843708407E-02"
nep23 = "X = 4.455662623587698E+09 Y =-3.833067895772491E+08 Z =-9.479194396546717E+07
 VX= 4.297305233379821E-01 VY= 5.447227229797033E+00 VZ=-1.222099521045787E-01"

# 2459701.500000000 = A.D. 2022-May-02 00:00:00.0000 TDB
sun22 = "X =-1.333512273509328E+06 Y = 3.445488642388455E+05 Z = 2.832491561644843E+04
 VX=-3.688954247616953E-03 VY=-1.535801956098796E-02 VZ= 2.095356489194861E-04"
mer22 = "X =-5.787964463498133E+07 Y = 5.001123068678558E+06 Z = 5.595649085666763E+06
 VX=-1.412786024627931E+01 VY=-4.647259111665712E+01 VZ=-2.500663888122642E+00"
ven22 = "X = 3.587612704045174E+07 Y =-1.019279819610429E+08 Z =-3.522707892272152E+06
 VX= 3.267015721969536E+01 VY= 1.184076411660992E+01 VZ=-1.722399554824109E+00"
ear22 = "X =-1.146121503189677E+08 Y =-9.911837382505153E+07 Z = 3.390113022846729E+04
 VX= 1.917565337627882E+01 VY=-2.252086620472453E+01 VZ= 1.293657910466095E-04"
mar22 = "X = 1.168429386885249E+08 Y =-1.723122647793255E+08 Z =-6.489026128990121E+06
 VX= 2.090480902530879E+01 VY= 1.575033363242685E+01 VZ=-1.822586427078310E-01"
jup22 = "X = 7.305933493985869E+08 Y =-1.301920721371184E+08 Z =-1.580507858542991E+07
 VX= 2.137303769034409E+00 VY= 1.347648922277490E+01 VZ=-1.037029936830756E-01"
sat22 = "X = 1.103882677050047E+09 Y =-9.837166526117289E+08 Z =-2.684592972538310E+07
 VX= 5.884999887995806E+00 VY= 7.191307123896053E+00 VZ=-3.588311673691735E-01"
ura22 = "X = 2.102764999640690E+09 Y = 2.064949098017363E+09 Z =-1.957237631821322E+07
 VX=-4.821150907798146E+00 VY= 4.541582713253250E+00 VZ= 7.931778087597818E-02"
nep22 = "X = 4.438820542662540E+09 Y =-5.547741023157761E+08 Z =-9.087265524840316E+07
 VX= 6.385999387923026E-01 VY= 5.425372291426413E+00 VZ=-1.270328159123488E-01"

# This code turns the strings of data into usable vectors
stringtovector = function(string) {
    words = paste(c("Y =", "Z =", "\n VX=", "VY=", "VZ="), collapse = "|")
    string = trimws(gsub(words, "split", string))
    string = gsub("X =", "", string)
```

```r
    string = as.numeric(strsplit(string, "split")[[1]])
    return(string)
}
sun23 = stringtovector(sun23)
mer23 = stringtovector(mer23)
ven23 = stringtovector(ven23)
ear23 = stringtovector(ear23)
mar23 = stringtovector(mar23)
jup23 = stringtovector(jup23)
sat23 = stringtovector(sat23)
ura23 = stringtovector(ura23)
nep23 = stringtovector(nep23)
sun22 = stringtovector(sun22)
mer22 = stringtovector(mer22)
ven22 = stringtovector(ven22)
ear22 = stringtovector(ear22)
mar22 = stringtovector(mar22)
jup22 = stringtovector(jup22)
sat22 = stringtovector(sat22)
ura22 = stringtovector(ura22)
nep22 = stringtovector(nep22)

SSMass = c(19885000, 3.302, 48.685, 59.721, 6.4171, 18981, 5683.4,
    868.13, 1024.1)
SSMass = SSMass * (10^23)

# The PV vectors are of the form x1 y1 z1 vx1 vy1 vz1 x2 y2
# z2...
SSPV23 = c(sun23, mer23, ven23, ear23, mar23, jup23, sat23, ura23,
    nep23)
SSPV23 = SSPV23 * 1000  #Converts from km to m
SSPV22 = c(sun22, mer22, ven22, ear22, mar22, jup22, sat22, ura22,
    nep22)
SSPV22 = SSPV22 * 1000
SSP22 = c(sun22[1:3], mer22[1:3], ven22[1:3], ear22[1:3], mar22[1:3],
    jup22[1:3], sat22[1:3], ura22[1:3], nep22[1:3])
SSV22 = c(sun22[4:6], mer22[4:6], ven22[4:6], ear22[4:6], mar22[4:6],
    jup22[4:6], sat22[4:6], ura22[4:6], nep22[4:6])
SSP22 = SSP22 * 1000
SSV22 = SSV22 * 1000

PlanetNames = c("Sun", "Mercury", "Venus", "Earth", "Mars", "Jupiter",
    "Saturn", "Uranus", "Neptune")
Year = 365 * 24 * 60 * 60  #Approximately one earth year in seconds

G = 6.6743e-11  #Gravitational Constant
ChargeConstant = 8987600000  #The proportionality coefficent in Coulomb's Law (1 / 4*pi*epsilon)

ME = SSMass[4]  #Mass of Earth
RE = 6371000  #Radius of Earth
```