Practical Numerical Simulations: Assignment 2

Kevin Grainger - 20333346

November 2024

Problem 1

x(t) obeys the second-order ordinary differential equation:

$$\frac{d^2x}{dt^2} = \frac{-tx(t+2)}{2+t^2x^2} \tag{1}$$

with boundary conditions x(0)=3/4, x(10)=-1 . Write software to solve this equation numerically using the fourth-order Runge-Kutta scheme and determine the values of x'(t=0) to 6sf. Plot any solutions you find.

1 Solution

The main aims we need to achieve in this solution:

- Find an expression for x'(t) & x'(t=0)
- Use this to solve the ODE numerically
- Plot the ODE and the evolution of x'(t=0)

1.1 Theory

We are presented with a 2nd order ODE, we can break this into a system of 1st order equations:

$$\frac{dx}{dt} = y$$
$$\frac{dy}{dt} = \frac{-tx(t+2)}{2+t^2x^2}$$

The issue in this case we are given only boundary conditions rather than initial conditions (ie. $x(t_0)=0$). We cannot use an iterative method to reproduce our ODE as we have no 'trajectory'/slope to use. The solution to this is the Shooting method.

1.1.1 Shooting Method

The Shooting method works by taking a guess at the solution to the initial value problem. We see our generalized 2nd order system can be written as such:

$$x''(t) = f(t, x(t), x'(t)) \ , \ x(t_0) = 3/4 \tag{2}$$

Letting:

$$x'(t_0) = a \tag{3}$$

We have now turned this problem into an initial value problem, as we can solve the system of equations that represent our 2nd order ODE. By using this guessed x'(t_0) we can find x_1 and so on, this is done using the iterative Runge-Kutta method. If our constructed x(t) satisfies the boundary condition x(10)=-1 we have found a solution. Representing our solution as such:

$$x(t) = x(t, x'(t)) \tag{4}$$

$$x(t_1, a) = x_1 \tag{5}$$

$$x(10, a) = -1 \tag{6}$$

Our solutions will correspond to the roots of the equation:

$$F(a) = x(t_1; a) - x_1 \tag{7}$$

We now need a root finding method. This guess can thus be altered until we reach the correct (within tolerance) values of our given boundary conditions. But what informs our the alteration we make on our initial x'(t) guess?

1.1.2 Bisection Method

The Bisection Method is based on the theorem: Given f(x) a real continuous function has at least one root between x_a and x_b if $f(x_a)f(x_b) < 0$. (Meaning the function has crossed the x-axis and changed sign). We start this process by taking 2 guesses for the root of our function, call these x_a and x_b. We want to take a wide guess to ensure we contain the root, or more specifically choose x_a and x_b such that $f(x_a)f(x_b) < 0$. We then make another estimate for the root location x_c:

$$x_c = \frac{x_a + x_b}{2} \tag{8}$$

To inform our next guess, we calculate the following and reassign variables as such:

$$f(x_a)f(x_c) < 0 \quad root \in [x_a, x_c] \ \ (\textbf{assign } \mathbf{x_b = x_c}) \tag{9}$$

$$f(x_a)f(x_c) > 0 \quad root \in [x_b, x_c] \ \ (\textbf{assign } \mathbf{x_a = x_c}) \tag{10}$$

$$f(x_a)f(x_c) = 0 \quad root = x_c \ \ (\textbf{end}) \tag{11}$$

We then repeat the process with our x variables reassigned so as to search in a smaller range. In practice we won't land on exactly the root ($f(x_a)f(x_b)=0$) so we need to define an acceptable error, upon which to stop the process.

$$|\epsilon| = |\frac{x_c^{current} - x_c^{previous}}{x_c^{current}}| \tag{12}$$

Meaning the distance from the actual root will be a maximum of $\epsilon/2$.

1.1.3 Runge-Kutta Method

We have broken the 2nd Order ODE into 2 1st order ODE's, as such:

$$\frac{dx}{dt} = y$$
$$\frac{dy}{dt} = \frac{-tx(t+2)}{2+t^2x^2}$$

These can be solved numerically using the standard fourth order method for a second order ODE broken into 2 first order ODE's:

$$k_1 = hz(x_0) = hz_0, \qquad l_1 = hf(x, y, z)$$
$$k_2 = h\left(z(x_0) + \frac{l_1}{2}\right), \qquad l_2 = hf\left(x_0 + \frac{h}{2}, y_0 + k_1, z_0 + \frac{l_1}{2}\right)$$
$$k_3 = h\left(z(x_0) + \frac{l_2}{2}\right), \qquad l_3 = hf\left(x_0 + \frac{h}{2}, y_0 + \frac{k_2}{2}, z_0 + \frac{l_2}{2}\right)$$
$$k_4 = h(z_0 + l_3), \qquad l_4 = hf\left(x_0 + h, y_0 + k_3, z_0 + l_3\right)$$

In my case I use a 2D vector form of the standard RK4 equations (below), with each element recording the progress of each equation in our system of ODE's.

$K_1 = hf(t_n, x_n)$ (Slope at the beginning of the timestep.)

$K_2 = hf\left(t_n + \frac{1}{2}h, x_n + \frac{1}{2}K_1\right)$ (Used K_1 to step halfway into timestep, then get midpoint slope.)

$K_3 = hf\left(t_n + \frac{1}{2}h, x_n + \frac{1}{2}K_2\right)$ (Used K_2 to step halfway into the timestep, another slope estimate taken.)

$K_4 = hf\left(t_n + h, x_n + K_3\right)$ (Used K_3 to step a full timestep, last slope estimate.)

1.2 Implementation

For our program to work we need to implement the Shooting Method (1.1.1), Bisection Method (1.1.2) and Runge-Kutta(1.1.3). The structure in which we implement these is as such:

- Define the RK4 process as a function **(return Y[0])**
- Define the Shooting Method as a function (Note: Shooting method uses the RK4 function within) **(return computed b.c - actual b.c)**
- Define the Bisection Method as a function **(return ($\mathbf{x_a}$+$\mathbf{x_b}$)/2)**

Driver code:

- Call Shooting Method function for (h=step-size)
- Define the Y[ ] vector with our initial system Y=[x_0,Shooting Method()]
- Call RK4 function to print values of our ODE.

```cpp
int main(){

    double h=0.01;                                  //step size
    //x'(t=0) -> bisect(-1.6,-1.3,h) our bisection function finds the actual value of x'(t=0), we give a wide range [-3,3]
    double dx_0=bisect(5.0,15.0,h);
    vector<double> X0 = {x_0,dx_0};                 //Our X0 vector is now the given initial condition x_0 and the calculated x'(t=0) [using bisection method]
    int n = (t1-t0)/h ;                             //Defining our number of iterations
    RK4_Save_Print(X0,t0,t1,n);                     //We use the now complete initial condition 'X0' vector in the Rk4 method to solve the ODE!
    cout<< "Final x'(0)="<<dx_0<<endl;              //Printing the value of x'(t)

    return 0;
}
```

Figure 1: Driver code: To show the order of function calling

1.2.1 RK4 Function

The code used for this section is the exact same as the previous Assignment. It looks as such: We define a vector operation as such, in order to take the RK steps using the derivative in vector form. The k-equations are also composed of vectors to simultaneously solve our 2D system. We print the time, x(t) and x'(t) values at each iteration. So we can later graph these values, once RK4 is called in the 'Shooting Method Function'.

```cpp
//The Vector calculation needed for the RK4 method
vector<double> AddMul(vector<double> U, vector<double> V, double a){
    for(int i=0; i<V.size(); i++){
        V[i] = U[i] + a*V[i];       //We have a for-loop, scaling every value of V[i] and adding it to vector U[i] (Generalized calculation needed in RK4)
    }
    return V;
}


//RK4 returns final value of X0 array at time tstop. Saves intermediate values in csv file to be plotted
vector<double> RK4_Save_Print(vector<double> X0, double t0, double tstop, int n){
    ofstream ofs;                                   //We are storing the results in a csv file as there were many data points
    ofs.open("X0_array.csv");

    double h=(tstop-t0)/(double(n));
    vector<double> k1, k2, k3, k4;                  //Our k-formulae become 2D vectors, as we are solving both ODE's in our system
    vector<double> X_temp;                          //Needed so as to not overwrite our original X0[]

    ofs<<"time,"<<"x(t),"<<"x'(t)"<<endl;           //Saves headers to csv
    cout<<"time,"<<"  x(t),"<<"x'(t),"<<endl;       //Prints headers
    for(int i=1; i<=n; i++){
        X_temp = X0;                                //sets X_temp to the intial conditions
        k1=derivative_array(t0,X_temp);             //k1 calculated in terms of derivative_array function with imputed initial conditions (note this is a 2-d vector)

        X_temp = AddMul(X0, k1, 0.5*h);             //X_temp is advanced by half step with AddMul function. (X_temp[] is altered, X0[] intact)
        k2=derivative_array(t0+0.5*h,X_temp);       //k2 uses the altered X_temp, ie. a half step added

        X_temp = AddMul(X0, k2, 0.5*h);             //process continues.
        k3=derivative_array(t0+0.5*h,X_temp);

        X_temp = AddMul(X0, k3, 1.0*h);
        k4=derivative_array(t0+h, X_temp);

        for (int i = 0; i < X0.size(); ++i) {
            X0[i] = X0[i] + (h / 6.0) * (k1[i] + 2.0 * k2[i] + 2.0 * k3[i] + k4[i]);    //We need to average the results in the standard Runge Kutta weighting, using for-loop to access every element
        }
        t0+=h;
        ofs<<t0<<","<<X0[0]<<","<<X0[1]<<endl;          //storing our csv data
        cout<<t0<<","<<X0[0]<<","<<X0[1] <<","<<endl;   //printing
    }
    ofs.close();
    return X0;                                      //returning our final X0 array
}
```

Figure 2: RK4 Method Overview

1.2.2 Shooting Method Function

The code needed for the shooting method is simple enough, as it relies in on calling both the RK4 function and the Bisection function. We need to start with our initial conditions vector defined as such:

$$X_0 = \begin{bmatrix} x(t_0) \\ \frac{dx(t_0)}{dt} \end{bmatrix} \tag{13}$$

We start by defining our initial condition vector $X_0[\ ]$ manually. X_0 is made up of our given initial condition for x(0), and our guess for x'(0) (which can be any number).We then create an array named X_guess[], this is simply the Rk4 method running using our guess at the intial conditions. Our shooting method function only has the job of returning a function f(x)=X_guess[0]-x0, the root of which occurs when we have the correct guess of x'(t). This will be found using the Bisection Method.

```
//Shooting Method
double shooting_method(double dxdt_guess, double h) {

    vector<double> X0 = {x_0, dxdt_guess};                                  //Defining the 'initial conditions' vector X0 as: x0, and random guess for x'(t). X0=[x0,x'(guess)]
    vector<double> X_RK4 = RK4_Save_Print(X0, t0, t1, int((t1 - t0) / h));  //We run RK4 with this guess for x'(t) as an input via X0, we are running the Rk4 with our guess at the intial conditions

    return X_RK4[0] - x_10;                                                 //We produce a function whose roots correspond to the correct guess of x'(t)
}
```

Figure 3: Shooting Method

1.2.3 Bisection Method Function

The Bisection Function follows a simple logic. The Shooting method is outputting the function:

$$F(x) = Y_{guess}[0] - X_1 \tag{14}$$

This being the function whose roots we are looking to find. We define f_{xa} and and f_{xb} as the value of our 'Shooting Method' function with the inputted values of x_a and x_b respectively. We then define the x_c value as the midpoint (as discussed in 1.1.2). Now using a while loops which stops once our root search is confined to a very small area (1.0e-6). Within this while loop we assign x_b=x_c if our product is negative, or x_a=x_c if positive. Then the x_c value is becomes the midway point between our new x_a and x_b. The result is a return the x-value for our root. (ie. the correct guess of x'(t))

```
//Bisection Method
//Needed to solve the returned equation from the shooting method (X_RK4[0] -x1)
//Saves revised boundary [x'(0)] for bisection method in csv file
double bisect(double x_a, double x_b,double h){                         //Defining our area boundaries, x_a, x_b , in which we search for a root

    double f_xa = shooting_method(x_a, h);                              //Finding the result of our RK4 with x_a (root guess) as the input
    double f_xb =shooting_method(x_b, h);                               //with x_b as the guess
    double x_c= 0.5*(x_a+x_b);                                          //Finds the middle of these boundaries

    ofstream ofs;
    ofs.open("Guesses.csv");


    while (x_b-x_a >1.0e-6){                        //We set our tolerance for how small our 'section' needs to be

        cout<<"x'(0) estimate "<<x_c<<endl;         //Printing our estimate after each iteration of the bisection method, to see the evolution of x'(t=0)
        ofs<<x_c<<endl;                             //Saving to csv file

        double f_xc = shooting_method(x_c,h);       //plug our middle value x_c into our shooting method funcition, which will use it in the RK4
        if(f_xa*f_xc < 0.0){                        //If the product of our functions (x_a & x_c) is negative it means we have a root in this section, as the function has crossed the axis
            x_b=x_c;                                //We then reassign the boundaries to make the search area smaller
            f_xb = f_xc;                            //Reassign our functions
        }
        else{                                       //In the case the product is positive we know the root is not located in that area, so we reassign the borders in the opposite direciton
            x_a=x_c;
            f_xa = f_xc;
        }
        x_c=0.5*(x_a+x_b);                          //We then take the mid point of our new section and repeat the process
    }
    ofs.close();
    return 0.5*(x_a+x_b);                           //We return the final point x_c, or in other words the average of x_a and x_b (these are very close in value)
}
```

Figure 4: Bisection Method

1.3 Results

We can find multiple solutions to this ODE by changing the range of our bisection function. Using step size h=0.01 .
Solution 1: x'(t=0) = -2.41626 Bisection range = [-7.0,0]

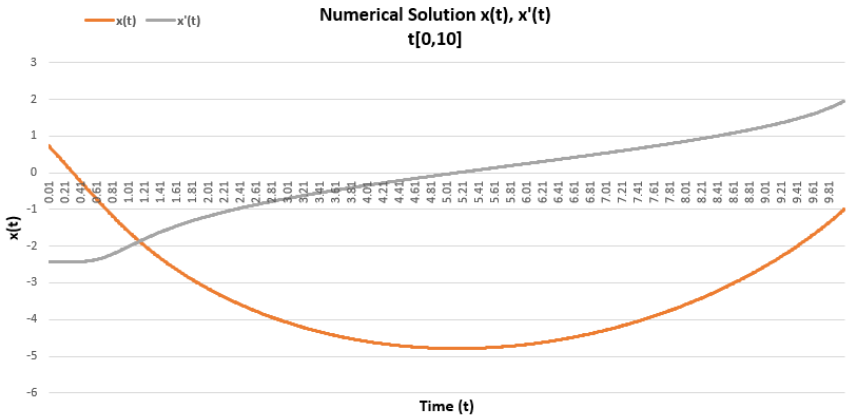

Figure 5: First Solution (1)

Solution 2: x'(0)=1.50568: Bisection Range: [0.0,2.0]

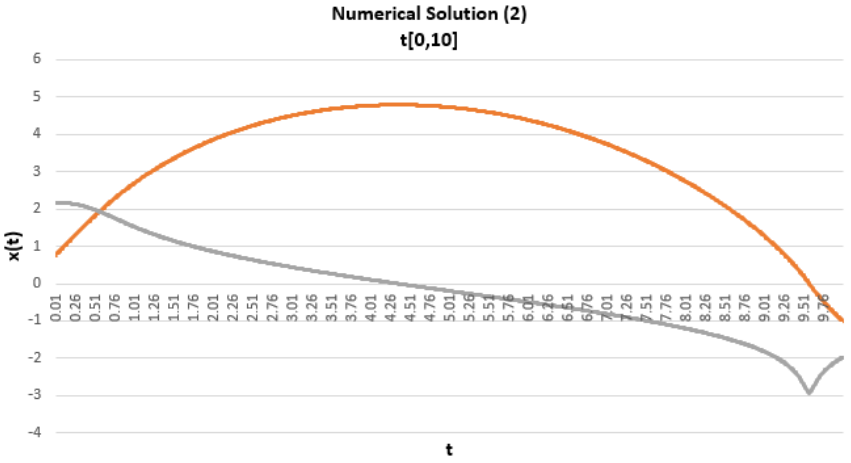

Figure 6: Second Solution (2)

Solution 3: x'(0)= -1.48228: Bisection function range [-3,3]

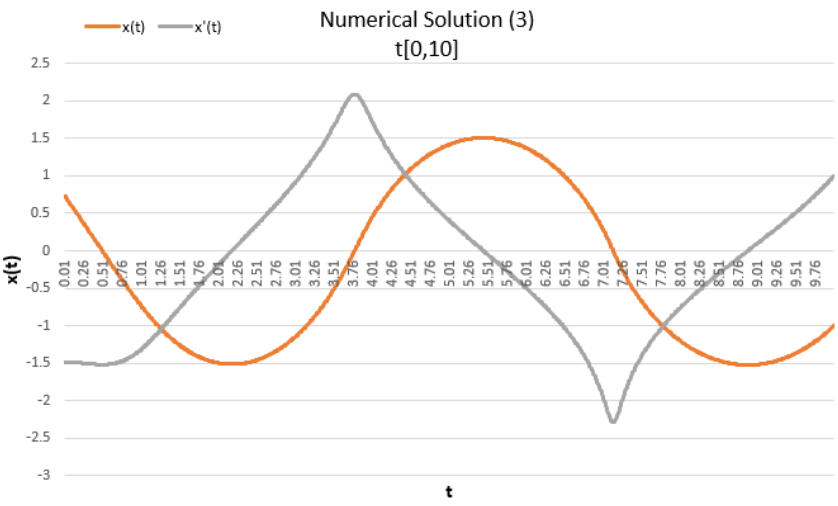

Figure 7: Numerical Solution 3

We can also track the evolution of x'(t=0) as the function iterates, we can get a more detailed results by using step size h=0.001:
For solution 3:

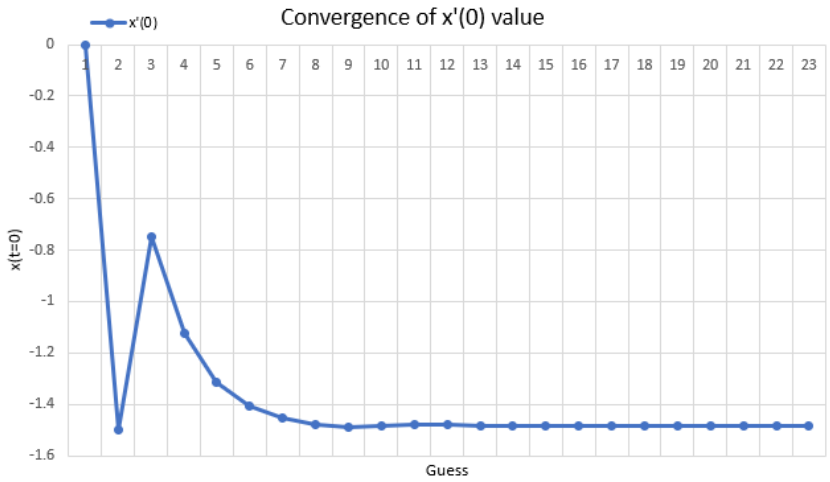

Figure 8: Convergence of x(t=0) for h=0.001

Problem 2

A system of four planets, moving in a two-dimensional plane with masses, initial observed positions and velocities in the plane is given in the table below, in units where Newton's gravitational constant G is unity. They move according to Newton's inverse-square law of gravity, which states the force of attraction exerted on planet i by planet j has magnitude.

$$|F_{ij}| = \frac{Gm_i m_j}{r_{ij}^2} \tag{15}$$

with r_{ij} the distance between the planets. The force is attractive and directed along the line joining i and j.

Planet	**Mass**	**Position** x_0	**Position** x_1	**Velocity** v_0	**Velocity** v_1
0	2.2	-0.50	0.10	-0.84	0.65
1	0.8	-0.60	-0.20	1.86	0.70
2	0.9	0.50	0.10	-0.44	-1.50
3	0.4	0.50	0.40	1.15	-1.60

Find the location of the planets at t = 5, accurate to 4 sf using a leap-frog symplectic integrator.

2 Solution

2.1 Theory

Our system uses the standard connected ODEs of motion:

$$\frac{dx_{ij}}{dt} = v_{ij} \tag{16}$$

$$m\frac{dv_{ij}}{dt} = \frac{Gm_i m_j}{r_{ij}^2} \tag{17}$$

2.1.1 Mechanics

Starting from:

$$|F_{ij}| = \frac{Gm_i m_j}{r_{ij}^2} \tag{18}$$

$$|a_{ij} m_{eff}| = \frac{Gm_i m_j}{r_{ij}^2} \tag{19}$$

Acceleration for each body, where a_i is the sum of acceleration on planet i, caused by 'j' other planets:

$$|a_i| = \sum_j \frac{Gm_j}{r_{ij}^2} \tag{20}$$

In co-ordinates:

$$|a_i| = \sum_j \frac{Gm_j}{(x_i - x_j)^2 + (y_i - y_j)^2} \tag{21}$$

We can also used the fact that these interactions are equal and opposite:

$$m_i a_i = -m_j a_j \tag{22}$$

$$a_i = -a_j \frac{m_j}{m_i} \tag{23}$$

To use these we need to vectorize the expressions:

Vector form:
Starting by decomposing our vector.

$$|F_{x,ij}| = |F_{ij}| Cos\theta \tag{24}$$

$$|F_{y,ij}| = |F_{ij}| Sin\theta \tag{25}$$

We can also say:

$$Cos\theta = \frac{\Delta x_{ij}}{r_{ij}} \tag{26}$$

Meaning our trig functions can be replaced.
$\hat{x}$ direction:

$$|F_{x,ij}| = |F_{ij}| \frac{\Delta x_{ij}}{r_{ij}} \tag{27}$$

$$|F_{x,ij}| = |F_{ij}| \frac{x_j - x_i}{r_{ij}} \tag{28}$$

$\hat{y}$ direction:

$$|F_{y,ij}| = |F_{ij}| \frac{\Delta y_{ij}}{r_{ij}} = \frac{G m_i m_j}{r_{ij}^3} (x_j - x_i) \tag{29}$$

$$|F_{y,ij}| = |F_{ij}| \frac{y_j - y_i}{r_{ij}} = \frac{G m_i m_j}{r_{ij}^3} (y_j - y_i) \tag{30}$$

Finally we can get the acceleration iteration equations we need:

$$a_{x,i} = \sum_j \frac{G m_j}{r_{ij}^3} (x_j - x_i) \tag{31}$$

$$a_{y,i} = \sum_j \frac{G m_j}{r_{ij}^3} (y_j - y_i) \tag{32}$$

$$\tag{33}$$

In the next section we will derive the correct method to make a predictive step forward for both velocities and then positions.

2.1.2 Leap Frog Method

The leap frog method is based off the finite difference method. To better illustrate we start with the Taylor expansion of f(x):
Forward step:

$$f(x+\Delta x) = f(x) + f'(x)\Delta x + \frac{1}{2}f''(x)\Delta x^2 \tag{34}$$

Backward step:

$$f(x-\Delta x) = f(x) - f'(x)\Delta x + \frac{1}{2}f''(x)\Delta x^2 \tag{35}$$

For small Δx and subtracting (18)-(19), then solving for f'(x) we have:

$$f'(x) = \frac{f(x-\Delta x) - f(x+\Delta x)}{2\Delta x} \tag{36}$$

$$v_n = \frac{x_{n-1} - x_{n+1}}{2\Delta x} \tag{37}$$

Halving the time step

$$f'(x) = \frac{f(x-\frac{\Delta x}{2}) - f(x+\frac{\Delta x}{2})}{\Delta x} \tag{38}$$

$$v_n = \frac{x_{n-1/2} - x_{n+1/2}}{\Delta x} \tag{39}$$

And now advancing a step $(1/2\Delta x)$

$$v_{n+\frac{1}{2}} = \frac{x_n - x_{n+1}}{\Delta x} \tag{40}$$

We see from the indices that the 'x' is value is taken at t=0,Δt,2Δt,...,$n\Delta t$. And 'v' is taken at t=1/2Δt,3/2Δt,...,n/2Δt. The 'x' terms appears to 'leap' over the 'v' terms as their intervals are mismatched. Hence the term leapfrog.

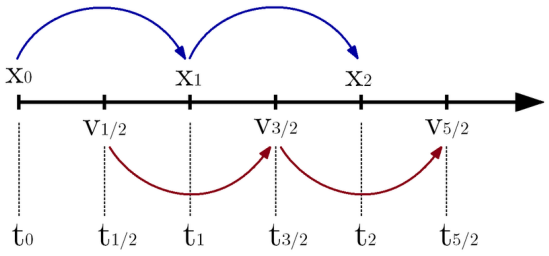

Figure 9: Leap Frog Graphic

In our specific case: $x \to t$ and $\Delta x \to \Delta t$

$$v_{n+\frac{1}{2}} = \frac{x_n - x_{n+1}}{\Delta t} \tag{41}$$

So we can now say our velocity can be approximation taken at the center of the difference. We can use this to start taking iterative steps forward:

$$x_{n+1} = x_n + v_{n+\frac{1}{2}}\Delta t \tag{42}$$

Given our 2D space we need to define x-y directions.

$$x_{n+1} = x_n + v_{x,n+\frac{1}{2}}\Delta t \tag{43}$$

$$y_{n+1} = y_n + v_{y,n+\frac{1}{2}}\Delta t \tag{44}$$

Acceleration and velocity have an equivalent relationship. Now we can simply update our velocities (1st order) by a half step using our acceleration (2nd Order).

$$a_{n+\frac{1}{2}} = \frac{v_n - v_{n+1}}{\Delta t} \tag{45}$$

$$v_{n+1} = v_n + a_{n+\frac{1}{2}}\Delta t \tag{46}$$

In our case (as in lectures) we are using a specific simple form of the leap-frog integrator. We are updating our position values by a half step, then the velocity by a full step, then finally the position by the final half step (using updated velocity).

$$x_{n+\frac{1}{2}} = x_n + \frac{\Delta t}{2}v_n \tag{47}$$

$$v_{n+1} = v_n + \Delta t\vec{a}_{n+\frac{1}{2}} \tag{48}$$

$$x_{n+1} = x_{n+\frac{1}{2}} + \frac{\Delta t}{2}v_{n+1} \tag{49}$$

$$\tag{50}$$

2.2 Code Execution

Our code has to follow a certain 'order' while updating our variables. We use initial conditions to calculate a(x_0). The code moves as such, where the right hand side of the arrow depends on the left hand side.

(Acceleration Update) → (Velocity Update) → (Position Update)

The code gets very wordy, so I preferred to group as many processes as possible into separate functions. This way our driver function should only need to (1) initialize our arrays with the starting data and (2) Call our separate functions in the order shown:

Position(dt/2) → Acceleration(dt) & Velocity(dt) → Position(dt/2)

2.2.1 Arrays

I needed to use arrays of larger dimension to store data in an x and y 'column' for each planet. This requires a 4x2 array.

$$array[4][2] = [[x_0, y_0], [x_1, y_1], [x_2, y_2], [x_3, y_3]] \tag{51}$$

Each set of x-y values is assigned to a planet. Storing the x-coordinates in the first entry and y in the second. These are accessed as such: x_i = array[i][0] refers to the x-value of the i^{th} planet. And y_i = array[i][1] is the y-value of the i^{th} planet. I used these to store and update the x-y; position, velocity and acceleration values.

2.2.2 Acceleration Calculation & Nested For-loops

To simulate the system we need a way to calculate and store the mutual gravitational force each planet 'i' and every other planet. This is done with 2 for-loops, one within the other. The first is standard, using values of i, the second for-loop uses the value j=i+1. This means the second loops find the gravitational interaction for each planet other than planet i, without double counting.

[0→1],[0→2],[0→3][1→2],[1→3],[2→3]

```
//We use nested for loops, this means we calculate the interaction between every pair of planets
//While it prevents double counting
//[0,1] [0,2] [0,3] [1,2] [1,3] [2,3]    (list of all calculated planet pairs, noting j=i+1)
for (int i = 0; i < np; ++i) {

    for (int j = i + 1; j < np; ++j) {

        double dx = pos[j][0] - pos[i][0];                    //The distance between planets in cartaesian co-ordinates
        double dy = pos[j][1] - pos[i][1];                    //Note of pos[][0] refers to the x position and pos[][1] to the y position

        double r = sqrt(dx * dx + dy * dy);                   //Radius
        double force_mag = (G * mass[i] * mass[j]) / (r * r);  //Magnitude of gravity force, as given

        double ax_i = (force_mag * dx) / (mass[i] * r);       //Acceleration(s) of planet i
        double ay_i = (force_mag * dy) / (mass[i] * r);       //As derived, when we vectorize of force into x-y directions we need to add factor (dx/r)
        acc[i][0] += ax_i;                                    //These accelerations are added to the 'total acceleration' in each direction
        acc[i][1] += ay_i;                                    //These accelerations only represent one interaction between a pair of planets, not the total acceleration for this location

        double ax_j = (force_mag * dx) / (mass[j] * r);       //Acceleration(s) of planet j
        double ay_j= (force_mag * dy) / (mass[j] * r);
        acc[j][0] -=ax_j;                                     //Forces are in opposite directions (towards each other)
        acc[j][1] -=ay_j;
    }
}
```

Figure 10: Nested For-Loops

2.2.3 Updating Position and Velocity Arrays

These functions are very simple. We update the whole arrays using a for-loop. Using the equations, where $v_{i,n}$ is the n^{th} velocity step of the i^{th} planet:

$$v_{i,n+1} = v_{i,n} + dta_{i,n} \tag{52}$$

Given $v_{i,n}$ is the previous velocity, we can just use += to add on the velocity change.

$$v_{i,n+1} += dta_{i,n} \tag{53}$$

Similarly with position:

$$x_{i,n+1} = x_{i,n} + dtv_{i,n} \tag{54}$$

$$x_{i,n+1} += dtv_{i,n} \tag{55}$$

```cpp
//I have grouped the processes of updating velocity and position into functions for readability in the main ()
//We udate the velocity using our derived formulas
//depending on what form of the leap frog method you wish to use, you can alter the timestep using the function call (eg. update_vel(dt/2))
//- rather than hard coding the factor into the formulae
void update_vel(double vel[np][2], double acc[np][2], double dt) {
    for (int i = 0; i < np; ++i) {

        vel[i][0] += dt * (acc[i][0]);  //x- velocity
        vel[i][1] += dt * (acc[i][1]);  //y- velocity
    }
}

//Position update
void update_pos(double pos[np][2], double vel[np][2], double acc[np][2], double dt) {
    for (int i = 0; i < np; ++i) {

        pos[i][0] += dt * vel[i][0];
        pos[i][1] += dt * vel[i][1];
    }
}
```

Figure 11: Updating Functions for Position and Velocity Arrays

2.2.4 Leap Frog

The leap frog method involves four steps. Unlike in the derivation of the method, with the execution we need to vary the step size, so as to have the velocity step and position step end in the same location.

1. Call position update for half time step.
2. Call acceleration calculation for new position
3. Call velocity update for full time step
4. Call position update for half time step using new velocity. (position now at full time step)

This is placed within a while loop, to run this procedure for a given time duration.

```cpp
//Main leapfrog function
//It only calls on all the previous fucntions
void leapfrog(double pos[np][2], double vel[np][2], double acc[np][2], double mass[np], double dt, double t_final) {

    double time = 0.0;                          //Initial time value
    cout << "Time,Planet0_x,Planet1_x,Planet2_x,Planet3_x,Planet0_y,Planet1_y,Planet2_y,Planet3_y" << endl;   //Printing data headers in csv format (x-positions to the left and y-positions to the right)


    while (time < t_final) {
        update_pos(pos, vel, acc, dt/2.0);      //(1) We update position by a half step

        acc_func(pos, acc, mass);               //(2) With this position we can calculate the force and accleration each planet experiences at that point (using acc_func)

        update_vel(vel, acc, dt);               //(3) Using the updated acceleration we can calculate velocity

        update_pos(pos, vel, acc, dt/2.0);      //(4)Using velocity we can update our position by a half step once more [we have now advanced a full time step]

        time += dt;                             //Next time step
    }

    Printing_func(pos, time);             //Printing function can be moved inside the while loop to print all position data
}
```

Figure 12: Leap-Frog Function

2.2.5 Printing Results

As we have using 4x2 arrays, we need to use for-loops to print each element. I needed the x-positions to be printed to the left and the y-position to the right, so I used two for-loops, each looping over pos[i][0] and pos[i][1], respectively. The ”,” is inserted as I was using the csv format.

```cpp
//We need a printing function as we have to cycle through each array element to print it
void Printing_func(double pos[np][2], double time) {

    cout << time;
    for (int i = 0; i < np; ++i) {                    //Printing in csv format , as I used excel to graph the data.
        cout << "," << pos[i][0];}
    for (int i = 0; i < np; ++i) {
        cout << "," << pos[i][1];}
    cout << endl;

}
```

Figure 13: Position Printing function

2.3 Results

Value of the Planets at t= 5

Planet	$\vec{X}(5)$	$\vec{Y}(5)$
0	-0.4819	0.3450
1	-0.5212	0.5294
2	-0.8112	-0.6852
3	-0.5069	-0.6396

Motion of the Planetary System on the x-y plane:

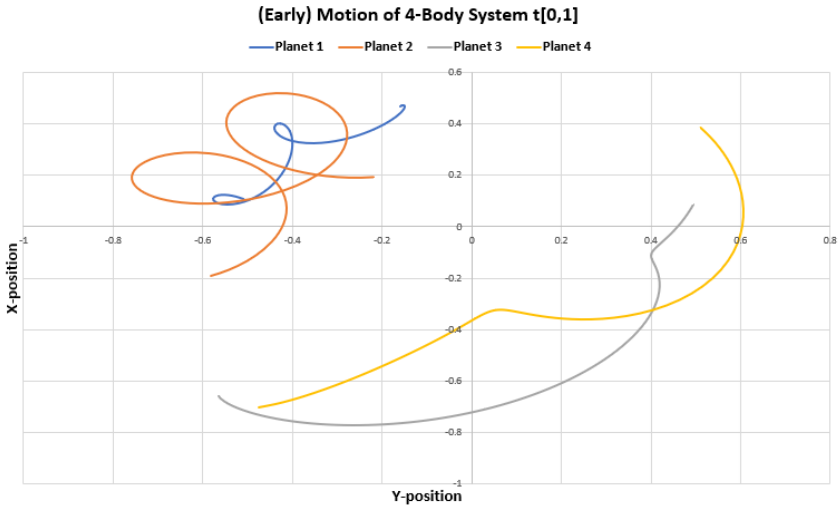

Figure 14: Planetary system motion t[0,1]

Planetary motion t[0,5]

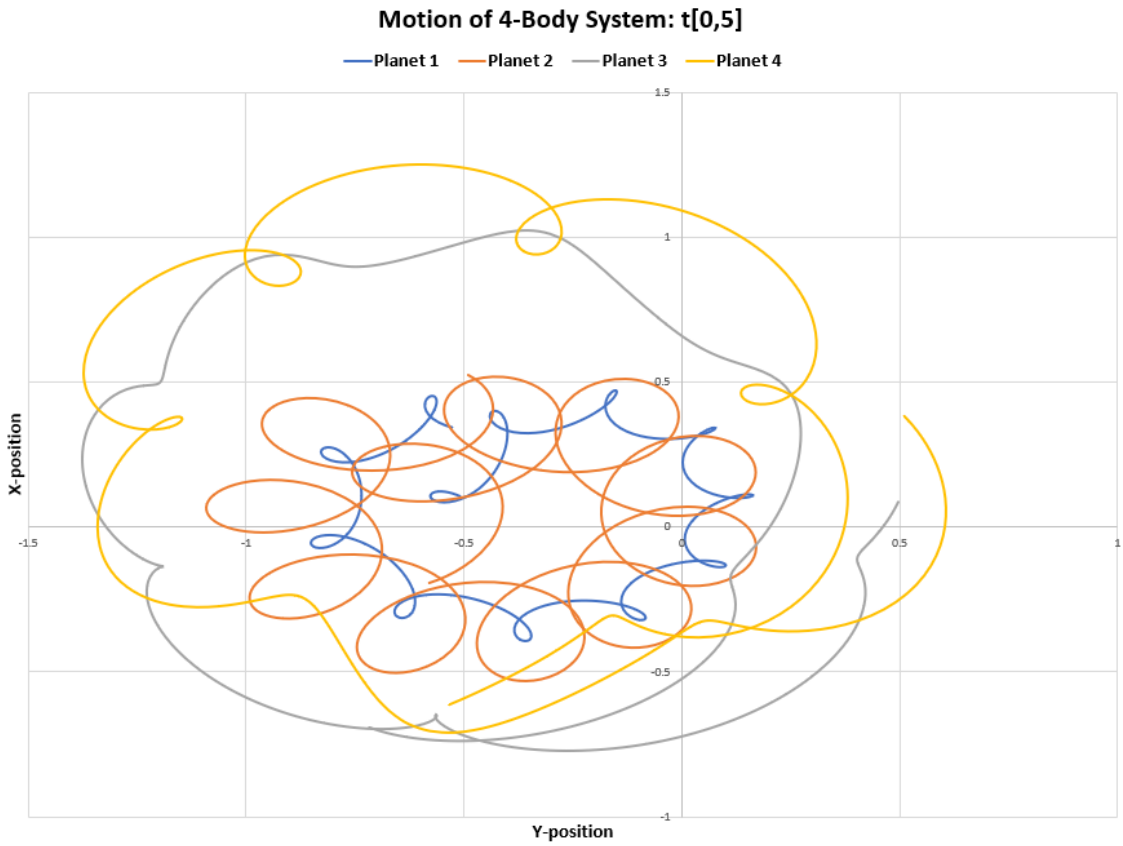

Figure 15: Planetary system motion t[0,5]

2.3.1 Accuracy of Results

The stability condition for our system: It was found by varying the step size that h=0.01 is the maximum step size to find convergence of results.
Conservation of Energy: We can make some assumption to the accuracy of the Leap Frog method if we can see the energy of the system is conserved. I used Excel to make these energy calculations using the printed data, we can then graph the kinetic energy of the system up to t=5. We see it is approximately constant at the value of 6, but begins to decline very slightly. This likely due to the build up of error as the iterations increase.

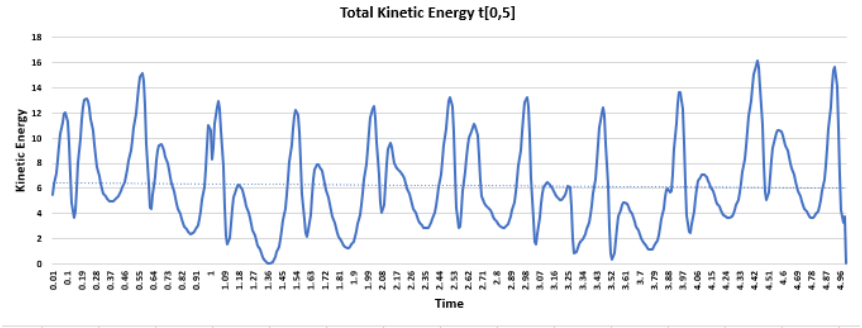

Figure 16: Kinetic Energy of System t[0,5]

Regarding total energy of the system, if our system is accurate, the potential energy should be constant, as the forces are equal and opposite. We can therefore say, the total energy in our system is approximately conserved, especially for smaller values of t. When we include potential energy, we see large spikes. This is most likely due to planets becoming very close and producing large velocities. This could be fixed by adding a minimum distance into the 'r' calculation. Regardless of the outliers we can see that Total Energy is more or less constant.

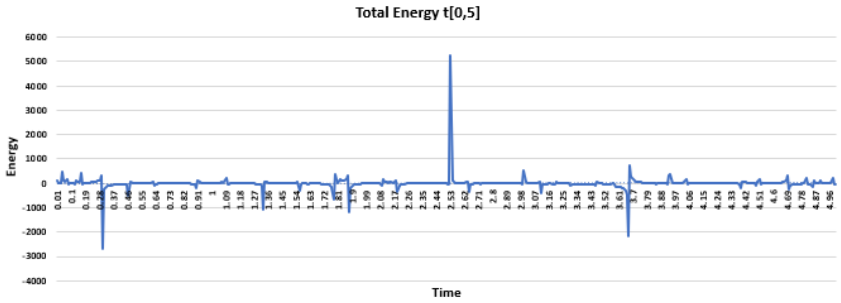

Figure 17: Total Energy of the System t[0,5]