Practical Numerical Simulations: Assignment 1

Kevin Grainger [20333346]

October 2024

Problem 1

Solving Analytically the ODE

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

Integrating both sides gives:

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

$$\ln(x+1) = \int (t^2 - 4t + 4) dt \tag{3}$$

$$\ln(x+1) = \frac{t^3}{3} - 2t^2 + 4t + C \tag{4}$$

Exponentiating both sides results in:

$$x = e^{\frac{t^3}{3} - 2t^2 + 4t + C} - 1 \tag{5}$$

Applying the boundary condition $x(0) = 0$:

$$e^C - 1 = 0 \implies C = 0 \tag{6}$$

Thus, we have:

$$x(t) = e^{\frac{t^3}{3} - 2t^2 + 4t} - 1 \tag{7}$$

Writing a C++ Program Using Fourth Order Runge-Kutta Method

To construct an approximate solution for $t \in [0, 1]$.

Theory

The 4th order Runge-Kutta method operates by taking a step forward from a given x_n by taking a weighted average of some intermediate slope values $f(t, x)$. Our step is defined as:

$$x_{n+1} = x_n + \sum_{m=1}^{4} b_m K_m \tag{8}$$

The step forward is composed of 4 slope calculations, starting in similar form to the Euler method, but each step is weighted to the previous steps. A final weighted average is taken using the b_m coefficients.

Firstly, at a time t_n (finding the slope at the beginning of the time step):

$$K_1 = hf(t_n, x_n) \tag{9}$$

Then evaluating the slope at a 'c1' step forward into our timestep and using the slope K_1 to project into the timestep:

$$K_2 = hf\left(t_n + c_2 h, x_n + a_{21}K_1\right) \tag{10}$$

Where a_{21} refers to the weighting given to the previous step K_1. We are looking for a point x, at which to evaluate our next slope within the time step; some methods use different locations, for example, the midpoint method will have $a_{21} = \frac{1}{2}$. It is written as such since these coefficients are written in a matrix form called a Butcher's tableau.

We take another measure of slope within the timestep, but this time using the K_2 to step forward:

$$K_3 = hf\left(t_n + c_3 h, x_n + (a_{31}K_1 + a_{32}K_2)\right) \tag{11}$$

This gives us two values for the slope at the 'midpoint', which when averaged will give a much more accurate result than the K_1 measurement at the start of the timestep.

Continuing the process:

$$K_4 = hf\left(t_n + c_4 h, x_n + (a_{41}K_1 + a_{42}K_2 + a_{43}K_3)\right) \tag{12}$$

To work out the coefficients, we need to use the Taylor expansion with a timestep and x-step:

$$f(t+h, x+k) = \sum_{m=0}^{\infty} \frac{1}{m!}\left(h\frac{\partial}{\partial t} + k\frac{\partial}{\partial x}\right)^m f(t,x) \tag{13}$$

Given that the RK4 method has 4 steps, this is a long expansion. To give the first steps:

$$x(t+h) = x(t) + w_1K_1 + w_2K_2 + w_3K_3 + w_4K_4 \tag{14}$$

Matching with the equivalent Taylor expansion:

$$x(t+h) = x(t) + h\dot{x}(t) + \frac{1}{2!}h^2\ddot{x}(t) + \frac{1}{3!}h^3\dddot{x}(t) + \frac{1}{4!}h^4\ddddot{x}(t) + \ldots \tag{15}$$

To match these terms, it is required that all the compound functions from our RK4 method are expanded, i.e. $w_2\left(hf(t_n + c_2h, x_n + a_{21}hf(t,x)\right)$. This is a long process but results in the ability to find simple solutions for our coefficients: (X and time-step coefficients respectively)

$$a_{21} = \frac{1}{2} \qquad c_2 = \frac{1}{2} \quad \text{(used in } K_2\text{)}$$
$$a_{31} = 0 \qquad c_3 = \frac{1}{2} \quad \text{(used in } K_3\text{)}$$
$$a_{32} = \frac{1}{2} \qquad c_4 = 1 \quad \text{(used in } K_4\text{)}$$
$$a_{41} = 0$$
$$a_{42} = 0$$
$$a_{43} = 1$$

Equations

Leaving these standard equations:

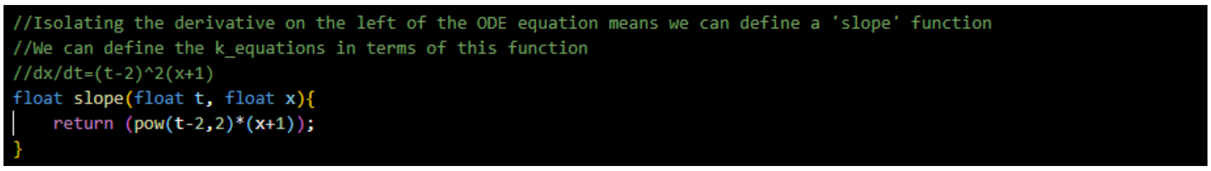
$$K_1 = hf(t_n, x_n) \quad \text{(Slope at the beginning of the timestep.)}$$
$$K_2 = hf\left(t_n + \frac{1}{2}h, x_n + \frac{1}{2}K_1\right) \quad \text{(Used } K_1 \text{ to step halfway into timestep, then get midpoint slope.)}$$
$$K_3 = hf\left(t_n + \frac{1}{2}h, x_n + \frac{1}{2}K_2\right) \quad \text{(Used } K_2 \text{ to step halfway into the timestep, another slope estimate t}$$
$$K_4 = hf(t_n + h, x_n + K_3) \quad \text{(Used } K_3 \text{ to step a full timestep, last slope estimate.)}$$

Now we can take a weighted average of these slopes and define a step forward:

$$x_{n+1} = x_n + \frac{1}{6}(K_1 + 2K_2 + 2K_3 + K_4) \tag{16}$$

Code Theory

To create this program we need to use our ODE in it's original form to create a function for slope. This means we can express our k-formulae in terms of this function.

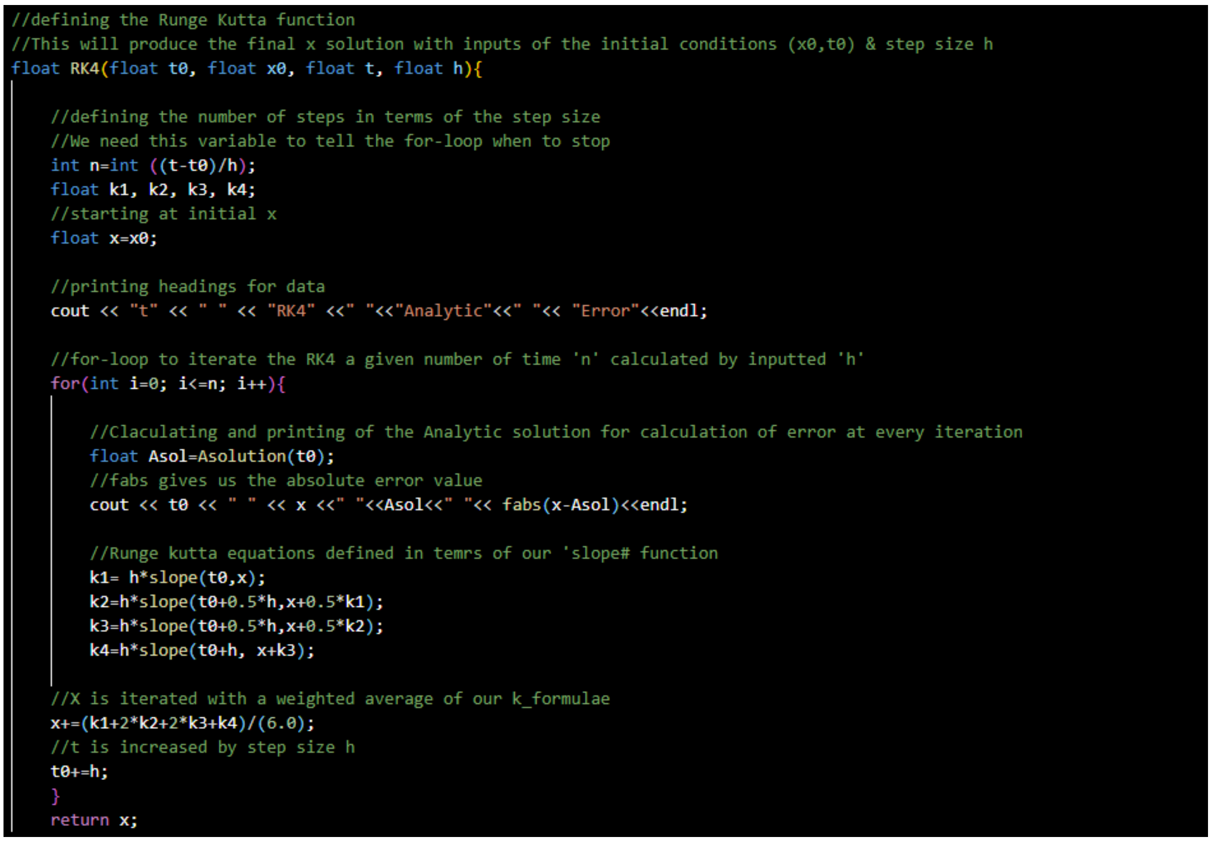


```
//Isolating the derivative on the left of the ODE equation means we can define a 'slope' function
//We can define the k_equations in terms of this function
//dx/dt=(t-2)^2(x+1)
float slope(float t, float x){
    return (pow(t-2,2)*(x+1));
}
```

Figure 1: 'Slope' function

We can then create a Runge Kutta function with the derived K-equations. These equations will be calculated from our 'slope' function (by imputing required step size), then inserted into the weighted average and finally added to the starting x0. This is put in a for-loop to be calculated and added to x repeatedly, until reaching a maximum 'n' steps (calculated by the value of step size 'h' inputted). We then increase the value of t by h (step). We print the Analytic solution at each run of the for-loop in order to calculate the error.

We then call this RK4 function with the initial conditions inserted as our starting variables.



```
//defining the Runge Kutta function
//This will produce the final x solution with inputs of the initial conditions (x0,t0) & step size h
float RK4(float t0, float x0, float t, float h){

    //defining the number of steps in terms of the step size
    //We need this variable to tell the for-loop when to stop
    int n=int ((t-t0)/h);
    float k1, k2, k3, k4;
    //starting at initial x
    float x=x0;

    //printing headings for data
    cout << "t" << " " << "RK4" <<" "<<"Analytic"<<" "<< "Error"<<endl;

    //for-loop to iterate the RK4 a given number of time 'n' calculated by inputted 'h'
    for(int i=0; i<=n; i++){

        //Claculating and printing of the Analytic solution for calculation of error at every iteration
        float Asol=Asolution(t0);
        //fabs gives us the absolute error value
        cout << t0 << " " << x <<" "<<Asol<<" "<< fabs(x-Asol)<<endl;

        //Runge kutta equations defined in temrs of our 'slope# function
        k1= h*slope(t0,x);
        k2=h*slope(t0+0.5*h,x+0.5*k1);
        k3=h*slope(t0+0.5*h,x+0.5*k2);
        k4=h*slope(t0+h, x+k3);

    //X is iterated with a weighted average of our k_formulae
    x+=(k1+2*k2+2*k3+k4)/(6.0);
    //t is increased by step size h
    t0+=h;
    }
    return x;
```

Figure 2: Runge-Kutta function

Plotting the Solution

Plot this solution for step size $h = 0.1$

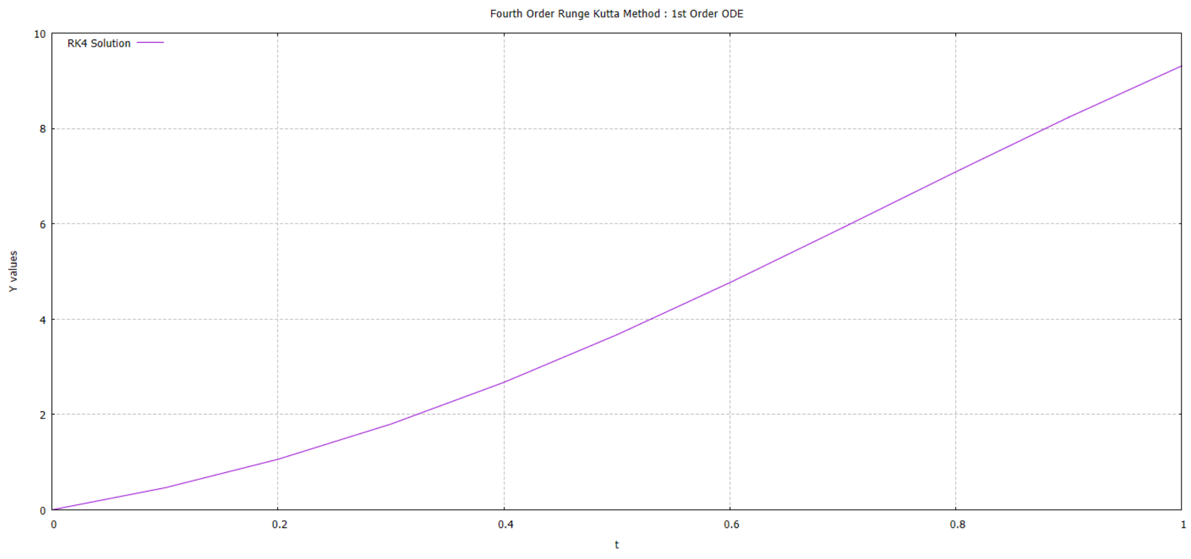


Figure 3: RK4 Solution vs. t

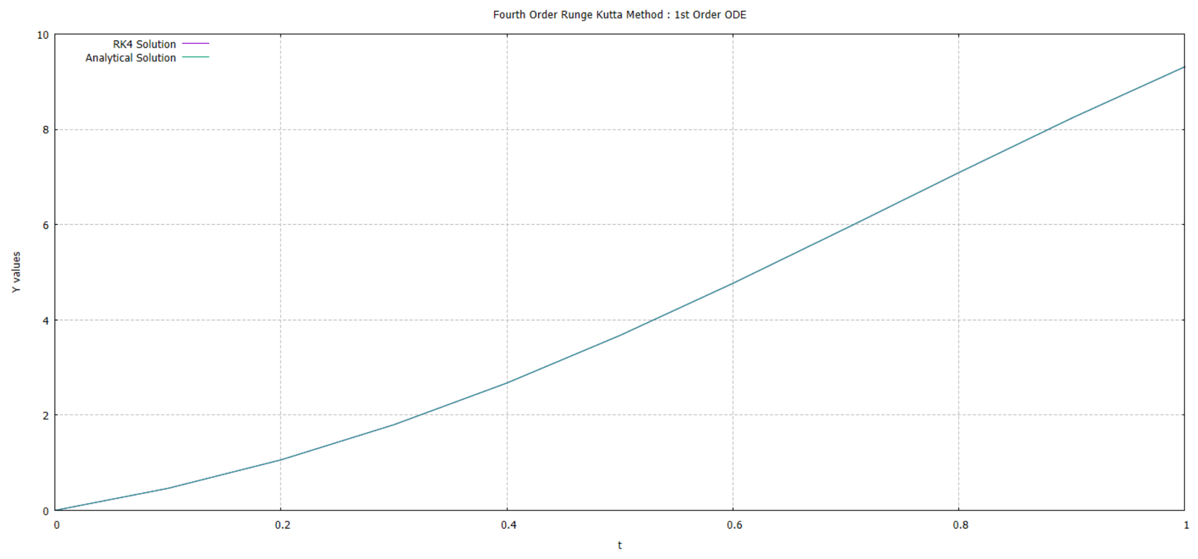


Figure 4: RK4 vs. Analytical

We see from Figure 4 that the RK4 and Analytical solution are nearly exactly equal.We can print the values to show the slight discrepancy.

# t	**RK4**	**Analytical**	**Error**
0	0	0	0
0.1	0.462699	0.462772	7.26581e-05
0.2	1.05975	1.05992	0.000165105
0.3	1.79799	1.79827	0.000275254
0.4	2.67379	2.67419	0.000399828
0.5	3.67184	3.67237	0.000535727
0.6	4.76544	4.76612	0.000679493
0.7	5.9186	5.91943	0.000826836
0.8	7.08933	7.09031	0.000973225
0.9	8.23388	8.235	0.00111485
1	9.31101	9.31226	0.00124645

Table 1: Comparison of RK4, Analytical Solution, and Error values

We can plot the development of the error as the RK4 iterates.

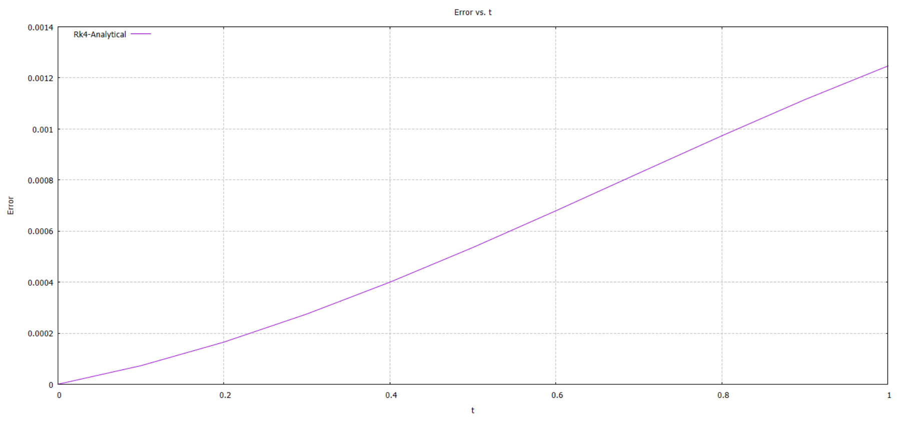


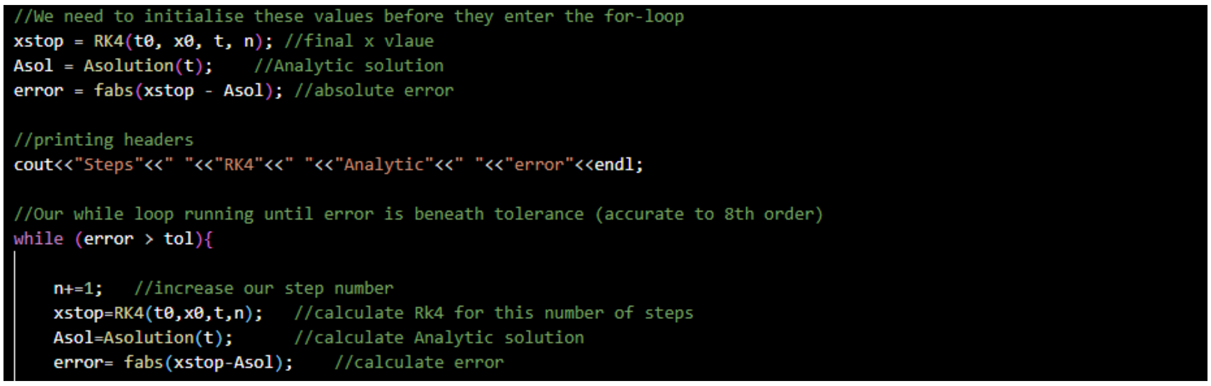
Figure 5: Error vs. t

This is to be expected as the error can be proportional to the value of x(t), dependent on facotrs like step size 'h' as will explored next.

Step Size Requirement

What value of step size is required for a solution accurate to 8 significant figures?

The only change needed to our code in order to find the step size needed is to add a while loop, which slowly increases the number of steps taken by the Runge-Kutta, this in turn decreases the step size gradually. The while-loop will have the condition stating the error value must be greater than the defined tolerance. By setting the tolerance to 1e-8 we can end then loop when we reach 8sf. of accuracy.



```
//We need to initialise these values before they enter the for-loop
xstop = RK4(t0, x0, t, n); //final x vlaue
Asol = Asolution(t);     //Analytic solution
error = fabs(xstop - Asol); //absolute error

//printing headers
cout<<"Steps"<<" "<<"RK4"<<" "<<"Analytic"<<" "<<"error"<<endl;

//Our while loop running until error is beneath tolerance (accurate to 8th order)
while (error > tol){

    n+=1;   //increase our step number
    xstop=RK4(t0,x0,t,n);   //calculate Rk4 for this number of steps
    Asol=Asolution(t);      //calculate Analytic solution
    error= fabs(xstop-Asol);    //calculate error
```

Figure 6: While-Loop to find required 'h'

Initially I had the issue of my error value reaching only 4sf. of accuracy and then diverging as the step size got smaller. This was due to the build up of errors. Such as rounding errors, which compounds as the number of steps increases. I was mistakenly using floats instead of doubles. Lastly I was defining 'n' (total steps) in terms of step size. This meant that if the the step size didn't divide perfectly into the total time step [0,1] the value 'n' had to be rounded as it is defined as an integer. We can solve this issue by flipping the equation, letting n be our independent variable. We then set 'n' to be a double.

$$h = \frac{(t - t0)}{double(n)} \tag{17}$$

0.1 Results

Our program produced 190 data points, so I can not display them here. The step required for 8 significant figures of accuracy is h = 0.005

Analytical Solution	9.31226
RK4 Solution	9.31226
Error	9.87477e-09
Final Number of Steps	200
Final Step Size	0.005

Table 2: Final Results & Step Size for 8sf.

0.2 Plotting Error vs. Step-size 'h'

To illustrate the fourth-order accuracy we show the error proportional to h^4. We see that error grow exponentially with step size from figure 7.

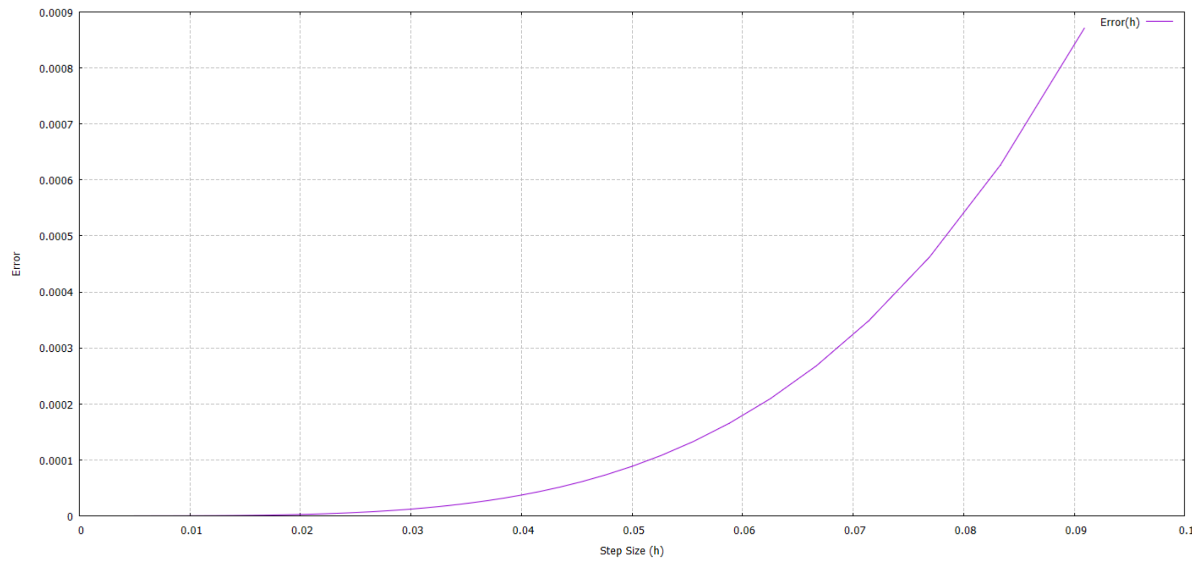


Figure 7: Error vs. Step size (h)

To prove the fourth order accuracy we use figure 8. we can already see our error follows a $x\hat{4}$ like path.

$$h \approx \left(\frac{\text{Error}}{C}\right)^{\frac{1}{4}} \tag{18}$$

Where 'C' is dependent on the initial problem. But we can see the full relationship and prove fourth order accuracy by graphing log(error) = 4log(h) as in Figure 8.

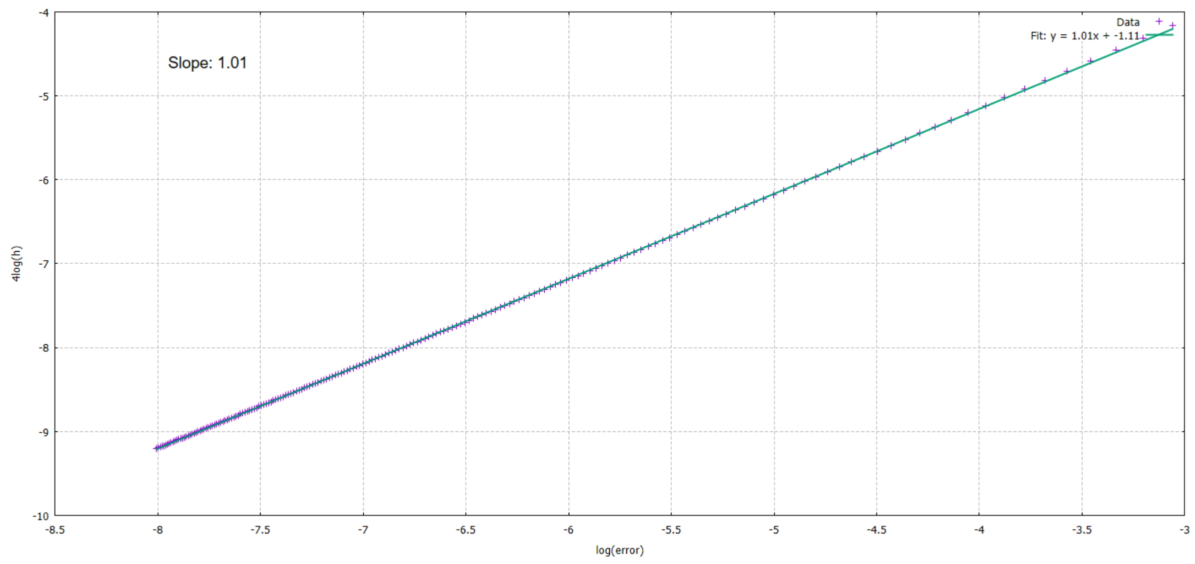


Figure 8: log(error) vs. 4log(h) plot

Slope = 1.01
We can see from Figure 8 that the log plot of our Error value vs. h$\hat{4}$ has very close to a 1:1 relationship. Meaning we have shown fourth order accuracy.

Problem 2

Write code that uses fourth-order Runge-Kutta to evaluate $y(30)$ numerically, accurate to 6 significant figures. Given y obeys:

$$7\frac{d^4y}{dx^4} + 70\frac{d^2y}{dx^2} + xy^3 = 0 \tag{19}$$

Given also the initial conditions:

$$y = 1 \quad \text{and} \quad \frac{dy}{dx} = \frac{d^2y}{dx^2} = \frac{d^3y}{dx^3} = 0 \quad \text{at } x = 0.$$

To solve this equation, you need to break it into a system of 4 lower-order ODEs. This is done by making the following substitutions:

$$\begin{aligned} \frac{d^4y}{dx^4} &= -\frac{xy^3}{7} - 10y_3 = y_4, \\ \frac{d^3y}{dx^3} &= y_3, \\ \frac{d^2y}{dx^2} &= y_2, \\ \frac{dy}{dx} &= y_1. \end{aligned}$$

This can be represented in an array as:

$$Y = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} y \\ \frac{dy}{dx} \\ \frac{d^2y}{dx^2} \\ \frac{d^3y}{dx^3} \end{bmatrix}, \quad Y' = \begin{bmatrix} \frac{dy}{dx} \\ \frac{d^2y}{dx^2} \\ \frac{d^3y}{dx^3} \\ -\frac{xy^3}{7} - 10\frac{d^3y}{dx^3} \end{bmatrix}$$

So our general solution will be Y[0]
For $x = 0$:

$$Y(x=0) = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

This makes it useful to use vectors in our C++ code, as we need to run multiple RK4 schemes, which also become connected since the derivative of y_1 equals y_2. The last equation y_4 has a 'measurable' slope at $x = 0$, serving as a jump-off point for the RK4 calculation.

To illustrate this using a second-order ODE:

$$\frac{d^2y}{dx^2} = f(x, y)$$

Letting z be the derivative:

$$\frac{dz}{dx} = f(x, y, z)$$

Vector form gives:

$$\begin{bmatrix} \frac{dy}{dx} \\ \frac{d^2y}{dx^2} \end{bmatrix} = \begin{bmatrix} z \\ f(x, y, z) \end{bmatrix}$$

Another approach would be to use 16 equations, 4 for each of 4 equations in our system. Solving each of these with RK4, where k_1, k_2, k_3, k_4 and l_1, l_2, l_3, l_4 are the steps for each equation respectively, we notice these have a 'velocity' and 'acceleration' relationship. This means that when executing the RK4 method, the change in slope with each step is simply an addition of the 'acceleration' term. Explicitly:

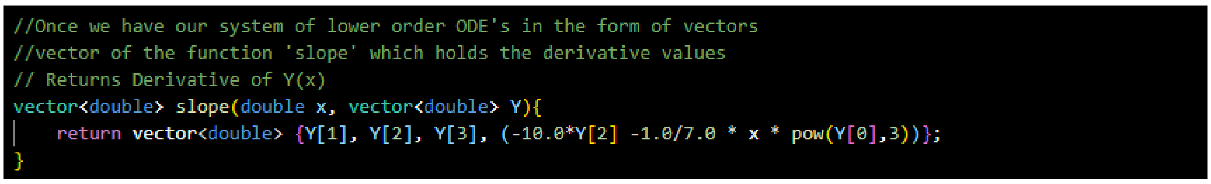
$$\begin{aligned}
k_1 &= hz(x_0) = hz_0, & l_1 &= hf(x, y, z) \\
k_2 &= h\left(z(x_0) + \frac{l_1}{2}\right), & l_2 &= hf\left(x_0 + \frac{h}{2}, y_0 + k_1, z_0 + \frac{l_1}{2}\right) \\
k_3 &= h\left(z(x_0) + \frac{l_2}{2}\right), & l_3 &= hf\left(x_0 + \frac{h}{2}, y_0 + \frac{k_2}{2}, z_0 + \frac{l_2}{2}\right)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

We can extend this for 4 sets of equations. I instead took the approach of using 4D vectors, where each each k_1,2,3,4[i] holds four equations, one for each ODE in the system.

0.3 Code method:

Our code needs to apply each of the RK4 equations on each of the ODEs. This will be achieved with a for loop over a vector composed of our ODEs instead of 4 sets of explicit equations.

Firstly we need to define our vector to represent the system Y[] as above in 'Equations'. Next we define a 'slope function which produces the derivative of Y[]. Meaning it's first entry is dy/dx which is Y[1]. This references the Y[] array which will be assigned as the initial values of Y[] at x=0.

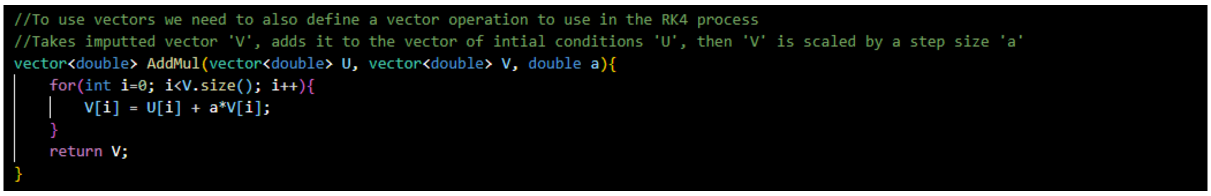


```cpp
//Once we have our system of lower order ODE's in the form of vectors
//vector of the function 'slope' which holds the derivative values
// Returns Derivative of Y(x)
vector<double> slope(double x, vector<double> Y){
    return vector<double> {Y[1], Y[2], Y[3], (-10.0*Y[2] -1.0/7.0 * x * pow(Y[0],3))};
}
```

Figure 9: Vector Function of derivatives

Next we need to define the vector operation involved in the Runge-Kutta. (Some vector operations didn't exist/ were very complex, so I found this easiest option)

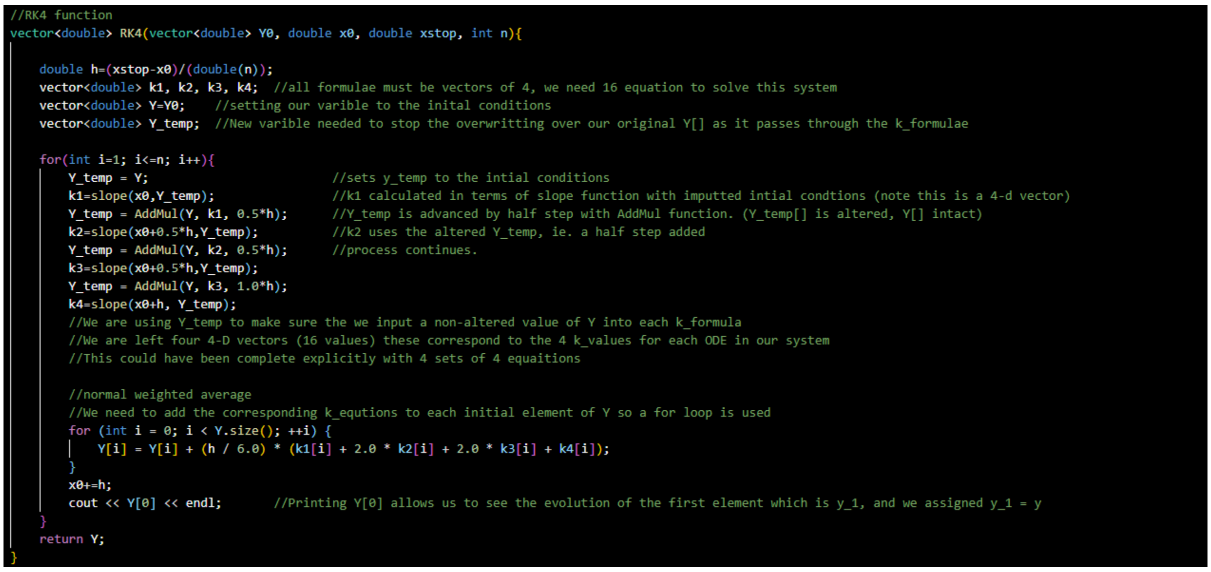
$$y_{n+1} = y_n + \sum_{n=1}^{4} a_n k_n \qquad\qquad V[i] = U[i] + aV[i]... \tag{20}$$



```cpp
//To use vectors we need to also define a vector operation to use in the RK4 process
//Takes imputted vector 'V', adds it to the vector of intial conditions 'U', then 'V' is scaled by a step size 'a'
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];
    }
    return V;
}
```

Figure 10: Vector Sum and Scaling function

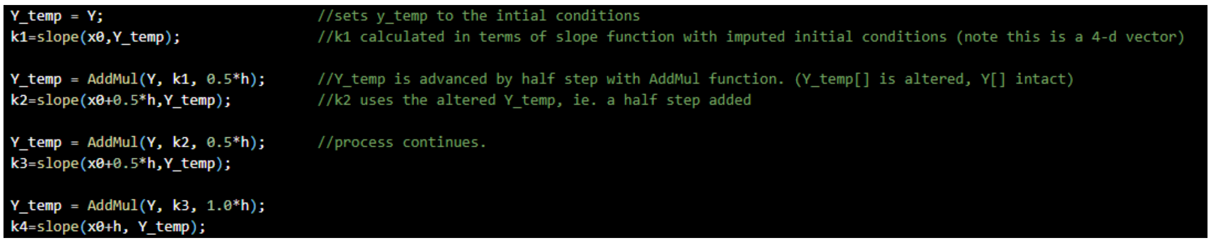
We will define our Runge Kutta function with vectors for all input. We firstly set our Y[ ] array as the Y0[ ] values (values of Y at x=0). Next we need a variable called 'Y_temp' and set it to the initial conditions. We need this variable so that our Y[ ] is not overwritten as it passes through the k-formulae.



```
//RK4 function
vector<double> RK4(vector<double> Y0, double x0, double xstop, int n){

    double h=(xstop-x0)/(double(n));
    vector<double> k1, k2, k3, k4;  //all formulae must be vectors of 4, we need 16 equation to solve this system
    vector<double> Y=Y0;    //setting our varible to the inital conditions
    vector<double> Y_temp;  //New varible needed to stop the overwritting over our original Y[] as it passes through the k_formulae

    for(int i=1; i<=n; i++){
        Y_temp = Y;                         //sets y_temp to the intial conditions
        k1=slope(x0,Y_temp);                //k1 calculated in terms of slope function with imputted intial condtions (note this is a 4-d vector)
        Y_temp = AddMul(Y, k1, 0.5*h);      //Y_temp is advanced by half step with AddMul function. (Y_temp[] is altered, Y[] intact)
        k2=slope(x0+0.5*h,Y_temp);          //k2 uses the altered Y_temp, ie. a half step added
        Y_temp = AddMul(Y, k2, 0.5*h);      //process continues.
        k3=slope(x0+0.5*h,Y_temp);
        Y_temp = AddMul(Y, k3, 1.0*h);
        k4=slope(x0+h, Y_temp);
        //We are using Y_temp to make sure the we input a non-altered value of Y into each k_formula
        //We are left four 4-D vectors (16 values) these correspond to the 4 k_values for each ODE in our system
        //This could have been complete explicitly with 4 sets of 4 equaitions

        //normal weighted average
        //We need to add the corresponding k_equtions to each initial element of Y so a for loop is used
        for (int i = 0; i < Y.size(); ++i) {
            Y[i] = Y[i] + (h / 6.0) * (k1[i] + 2.0 * k2[i] + 2.0 * k3[i] + k4[i]);
        }
        x0+=h;
        cout << Y[0] << endl;       //Printing Y[0] allows us to see the evolution of the first element which is y_1, and we assigned y_1 = y
    }
    return Y;
}
```

Figure 11: Runge-Kutta Function in vector form.

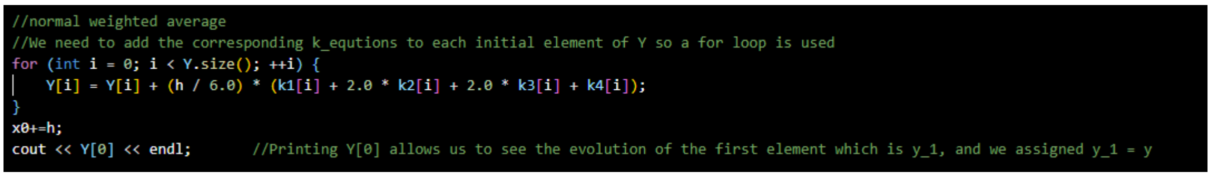
Firstly is k1 calculated in terms of slope function with imputed initial conditions (note this is a 4-d vector),Y_temp is then advanced by half step with the AddMul function to be used to calculate k2, Y_temp[] is altered, Y[] is intact (our vector Y[] holds value of 'y' at 'x', Y_temp holds value of y+k1 etc) This leaves us with four 4D vectors, one for each k-formula, all in the form Y0[i]+h(slope(i)).



```
Y_temp = Y;                         //sets y_temp to the intial conditions
k1=slope(x0,Y_temp);                //k1 calculated in terms of slope function with imputed initial conditions (note this is a 4-d vector)

Y_temp = AddMul(Y, k1, 0.5*h);      //Y_temp is advanced by half step with AddMul function. (Y_temp[] is altered, Y[] intact)
k2=slope(x0+0.5*h,Y_temp);          //k2 uses the altered Y_temp, ie. a half step added

Y_temp = AddMul(Y, k2, 0.5*h);      //process continues.
k3=slope(x0+0.5*h,Y_temp);

Y_temp = AddMul(Y, k3, 1.0*h);
k4=slope(x0+h, Y_temp);
```

Figure 12: RK4 process (Zoomed in)

This all needs to be iterated. Done as such in vector form:



```
//normal weighted average
//We need to add the corresponding k_equtions to each initial element of Y so a for loop is used
for (int i = 0; i < Y.size(); ++i) {
    Y[i] = Y[i] + (h / 6.0) * (k1[i] + 2.0 * k2[i] + 2.0 * k3[i] + k4[i]);
}
x0+=h;
cout << Y[0] << endl;       //Printing Y[0] allows us to see the evolution of the first element which is y_1, and we assigned y_1 = y
```

Figure 13: Iteration of Y[i]

We want to print the value of Y[0] as it is defined as $y_1 = y$ (which equals 1 at x=0), thus we can track the progression of the solution.

0.4 Evaluating the ODE

Our function prints the X and Y[0] values. From this we can plot the solution on the desired range:

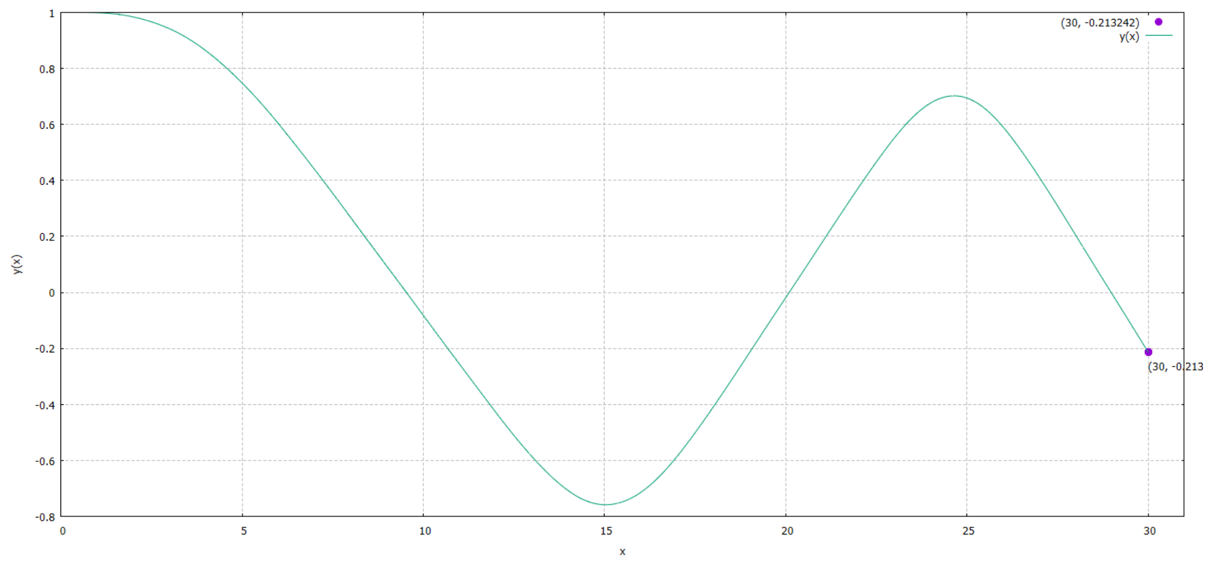


Figure 14: Graphed 4th Order ODE [with y(30) marked]

Solution to 6 sf. y(30)=-0.213242

0.5 References

[1]GeeksforGeeks (2015). Vector in C++ STL. [online] GeeksforGeeks. Available at: `https://www.geeksforgeeks.org/vector-in-cpp-stl/.`

[2]Marcel (2024). Arithmetic operations on vectors. [online] Stack Overflow. Available at: `https://stackoverflow.com/questions/34037634/arithmetic-operations-on-vectors[Accesse`
`2024].`

[3]Runge-Kutta method. (n.d.). Available at: `https://math.okstate.edu/people/yqwang/teaching/math4513\_fall11/Notes/rungekutta.pdf.`

[4]Harold Serrano. (n.d.). Visualizing the Runge-Kutta Method. [online] Available at: `https://www.haroldserrano.com/blog/visualizing-the-runge-kutta-method.`