Project in Computational Mathematics

The University of Leeds

Author: Kathryn Irish 200855216
Supervisor: Professor Stephen Griffiths
Module: MATH3001 Project in Mathematics
March 22, 2018

Contents

1 Introduction

In this project I explore the application of computational methods to various problems in the field of applied mathematics. Frequently, in both academia and industry, it is not possible to find explicit solutions to mathematical problems and in these cases computational solutions become necessary. In this document I look primarily at problems that do have known explicit solutions - this allows for a comparison between approximation and explicit solution. By taking this approach we can explore some of the limitations of numerical approximations and also the limitations of software.

I used the programming language Python with the software package Anaconda - both free to use, readily available, and open source. Python is designed to be easily readable and the syntax allows for concepts to be expressed in relatively few lines of code. The Python libraries NumPy and Matplotlib, both automatically installed in Anaconda, provide support for various mathematical functions including matrix algebra and easy graph plotting respectively. These aspects make Python a sensible choice for this project.

1.1 Time Stepping Schemes

I'll begin by giving a general overview of time stepping schemes, used in all 3 sections of this project. Additional information on time stepping schemes can be found in Epperson (2002), which provided guidance for this section. Time stepping can be used to approximate the solutions to differential equations, this is especially useful for equations which cannot be solved analytically.

Say we have differential equation

$$\dot{\phi} = f(t, \phi), \quad \phi(t_0) = y_0$$

Then the scheme is implemented as such, where Δt is the size of the time step. Different schemes use different methods to estimate the value of the integral.

$$\phi(t + \Delta t) = \phi(t) + \int_t^{t+\Delta t} \dot{\phi} dt$$

The simplest of these time stepping schemes is the first order Euler scheme. This works by approximating the integral as a rectangle.

$$\phi(t + \Delta t) = \phi(t) + \Delta t f(t, \phi(t))$$

The Runge-Kutta 4th order scheme is very commonly used, and tends to be significantly more accurate than the Euler scheme. This scheme uses the weighted average of 4 increments to approximate the integral.

$$\phi(t + \Delta t) = \phi(t) + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

$k_1 = f(t, \phi(t)),\ \ k_2 = f(t+\frac{\Delta t}{2}, \phi(t)+\Delta t\frac{k_1}{2}),\ \ k_3 = f(t+\frac{\Delta t}{2}, \phi(t)+\Delta t\frac{k_2}{2}),\ \ k_4 = f(t+\Delta t, \phi(t)+\Delta t k_3)$

The "order" of a scheme refers to the proportion of the error to the size of the time step as shown below. n is the order of the error.

$$E \propto \Delta t^n$$

2 Determining the period of a pendulum

A frictionless pendulum supported by a massless rigid rod of length l moves under the influence of a downward gravitational acceleration g. At time τ, the angle from the downward vertical is denoted by θ, and can be shown to satisfy equation (2.1). Figure 1 is a diagram of this set up.

$$\frac{d^2\theta}{d\tau^2} = -(g/l)\sin\theta \tag{2.1}$$

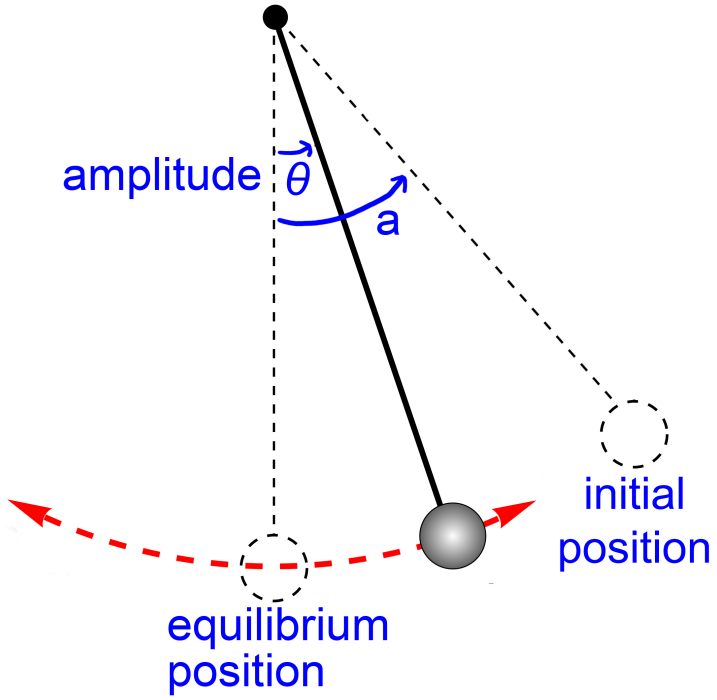


Figure 1: diagram showing pendulum set up

2.1 Deriving and Manipulating the Equations

Derivation
Let $t = \tau/k$ for a suitable constant k, with units of time.

We can rewrite equation 1.1 as equation 1.2 where $\dot{\theta}$ refers to differentiation by t. Here a is the initial amplitude.

$$\ddot{\theta} = -\sin\theta, \quad \theta(0) = a, \quad \dot{\theta}(0) = 0 \tag{2.2}$$

To derive equation (1.2) from (1.1) first define k.

$$k = \sqrt{l/g} \Rightarrow t = \tau\sqrt{g/l}$$

Then using the chain rule we get this, which is substituted back into (1.1) to get (1.2).

$$\frac{d^2}{d\tau^2} = (g/l)\frac{d^2}{dt^2}$$

Linear approximation
For very small values of $a \ll 1$ we can also approximate (1.2) as (1.3)

$$\ddot{\theta} = -\theta, \quad \theta(0) = a, \quad \dot{\theta}(0) = 0. \tag{2.3}$$

This comes from the Taylor series $\sin\theta = \theta - \frac{\theta^3}{3!} + \frac{\theta^5}{5!} - \frac{\theta^7}{7!}\ldots$

if a is very small θ is also very small and the rest of the series after the first term is negligible. Hence $\sin\theta$ can be approximated by just θ

Rewriting as a system of first order equations
second order equations can be rewritten as a system of 2 first order equations. The non-linear case (1.2) can be rewritten

$$\dot{\theta}_0 = \theta_1, \quad \theta_0(0) = a. \quad \dot{\theta}_1 = -\sin\theta_0, \quad \theta_1(0) = 0, \tag{2.4}$$

and the linear scheme (1.3) can be rewritten

$$\dot{\theta}_0 = \theta_1, \quad \theta_0(0) = a. \quad \dot{\theta}_1 = -\theta_0, \quad \theta_1(0) = 0. \tag{2.5}$$

2.2 Time-Stepping Schemes

We implement the Euler time step scheme as such for the linear and non-linear schemes

$$\theta_0(t+\Delta t) \approx \theta_0(t) + \Delta t \cdot \theta_1(t), \quad \theta_1(t+\Delta t) \approx \theta_1(t) - \Delta t \cdot \theta_0(t).$$

$$\theta_0(t+\Delta t) \approx \theta_0(t) + \Delta t \cdot \theta_1(t), \quad \theta_1(t+\Delta t) \approx \theta_1(t) - \Delta t \cdot \sin(\theta_0(t)).$$

We also use the Runge Kutta schemes RK2, RK3, and RK4. The general RK2 method is outlined by Burden and Faires (2001). We implement RK2 for the linear system as follows

$$\begin{aligned} k_1 &= \Delta t \cdot \theta_1(t) & m_1 &= -\Delta t \cdot \theta_0(t) \\ k_2 &= \Delta t(\theta_1(t) + \tfrac{k_1}{2}) & m_2 &= -\Delta t(\theta_0(t) + \tfrac{k_1}{2}) \\ \theta_0(t+\Delta t) &= \theta_0(t) + k_2 & \theta_1(t+\Delta t) &= \theta_1(t) + m_2. \end{aligned}$$

RK4 is implemented for the linear system as follows.

$$\begin{aligned} k_1 &= \Delta t \cdot \theta_1(t) & m_1 &= -\Delta t \cdot \theta_0(t) \\ k_2 &= \Delta t(\theta_1(t) + \tfrac{k_1}{2}) & m_2 &= -\Delta t(\theta_0(t) + \tfrac{m_1}{2}) \\ k_3 &= \Delta t(\theta_1(t) + \tfrac{k_2}{2}) & m_3 &= -\Delta t(\theta_0(t) + \tfrac{m_2}{2}) \\ k_4 &= \Delta t(\theta_1(t) + k_3) & m_4 &= -\Delta t(\theta_0(t) + m_3) \\ \theta_0(t+\Delta t) &= \theta_0(t) + \tfrac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) & \theta_1(t+\Delta t) &= \theta_1(t) + \tfrac{1}{6}(m_1 + 2m_2 + 2m_3 + m_4) \end{aligned}$$

I implemented Euler, RK2, RK3, and RK4 time stepping schemes in python for both the non-linear and linear systems of equations. These schemes have orders of error 1, 2, 3, and 4 respectively.

We can demonstrate the theoretical orders of these schemes in Python by calculating the error of the approximation $E = |\cos(t_{max}) - \hat{\theta}(t_{max})|$ (I used $t_{max} = 5$) for a number of Δt values and then plotting $\log$ error against $\log \Delta t$. I did this for $\Delta t = [0.004, 0.008, 0.016, ..., 0.128]$. Figure 2 shows the result, the gradients of the lines are 0.91 for RK1, 2.01 for RK2, 2.87 for RK3 and 4.01 for RK4. This approximately corresponds to the theoretical orders.

2.3 Directly Measuring the Period

Our main topic of interest within this project is determining the period $T(a)$, the length of time it takes for the pendulum to do a full forward and back swing. We can estimate $\frac{T}{4}$ by approximating the value of t where $\theta(t) = 0$. We do this by interpolating a finite set of points from our time step scheme $(t_n, \theta(t_n)), (t_{n+1}, \theta(t_{n+1})) \dots (t_m, \theta(t_m))$ for which $0 \in [\theta(t_m), \theta(t_n)]$.

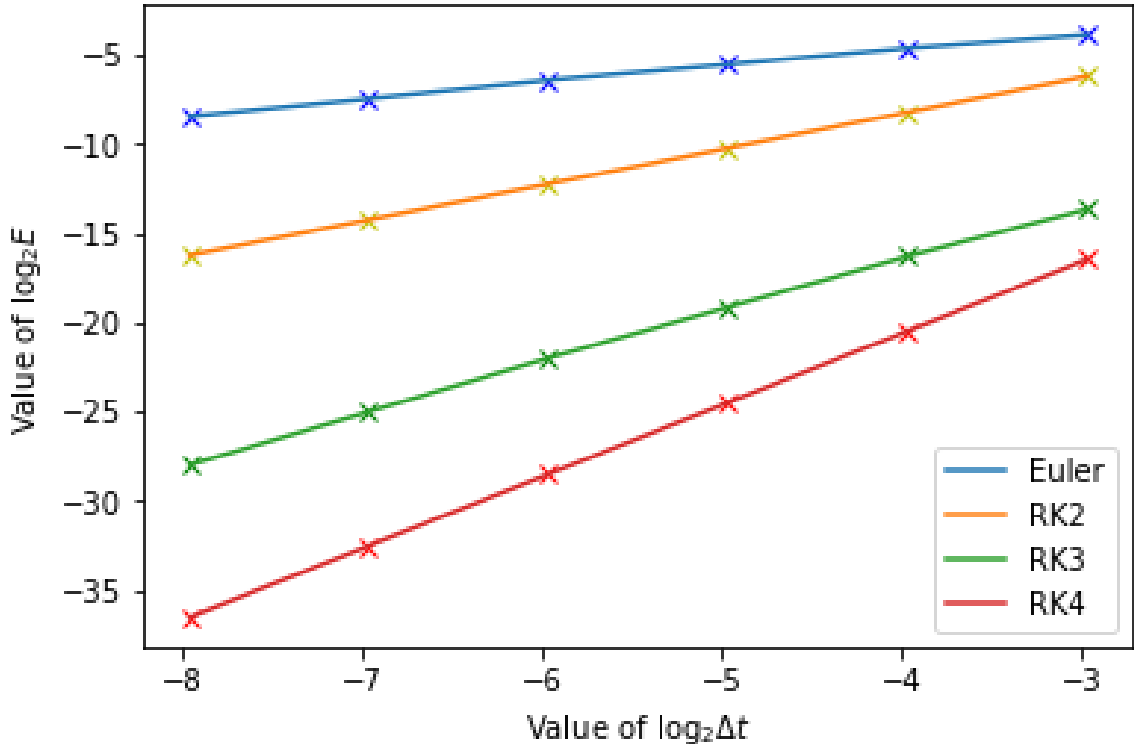


Figure 2: log plot of error of time step estimates of θ against Δt

Lagrange interpolation

The simplest form of interpolation is a straight line between two points. For this we need two points $(t_n, \theta(t_n))$ and $(t_{n+1}, \theta(t_{n+1}))$ where $0 \in [\theta(t_n), \theta(t_{n+1})]$. The formula for linear interpolation is

$$\theta(t) - \theta(t_n) = \frac{\theta(t_{n+1}) - \theta(t_n)}{t_{n+1} - t_n}(t - t_n)$$

By setting $\theta(t) = 0$, and noting that for this case $t = \frac{T}{4}$, and that $t_{n+1} - t_n = \Delta t$ we can rearrange to get

$$\frac{T}{4} \approx t_n - \frac{\theta(t_n) \cdot \Delta t}{\theta(t_{n+1}) - \theta(t_n)}$$

We can measure the error of the approximated period when we know the value of $T(a)$. There are 2 cases (i) the linear case where $T(a) = 2\pi$ for all values of a, (ii) the non-linear case where the solution can only be found exactly in a few cases, one of which is when $a = \frac{\pi}{2}$. The solution in this case is below, where the symbol Γ refers to the gamma function which is defined as $\Gamma(z) = \int_0^\infty x^{z-1}e^{-x}dx$.

$$T(\pi/2) = \frac{\Gamma(1/4)^2}{\sqrt{\pi}} \approx 7.42$$

I did this for a linear interpolation with the RK2 time step scheme, figure 3 shows the log plot for this interpolation. The gradient of the line in figure 3 is 2.03 corresponding to second order error.

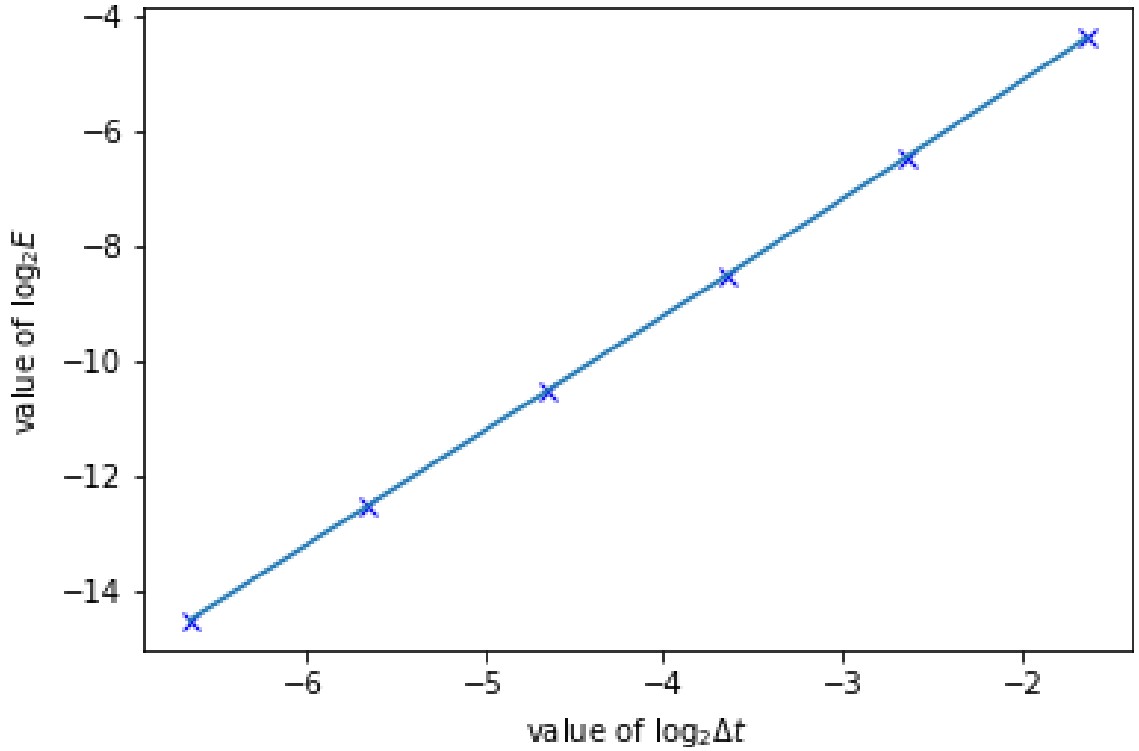


Figure 3: log plot of error of period estimate against dt for RK2 with linear interpolation

I repeated this for all 4 time stepping schemes using 3 orders of interpolation - linear, quadratic, and cubic. When interpolating I used Lagrange interpolation. This is a method for interpolating n points of the form (t_i, θ_i) to get a polynomial with order $n-1$.

The Lagrange interpolating polynomial is defined below (Burden and Faires, 2008, p.109). This functions so that for all the given points $L(x_i) = \theta_i$

$$L(t) = \sum_{k=0}^{n} \theta_k \cdot l_k(t)$$

Where

$$l_k(t) = \prod_{i=0,\, i\neq k}^{n} \frac{t - t_i}{t_k - t_i}, \quad l_k(t_j) = \begin{cases} 1, i = k \\ 0, i \neq k \end{cases}$$

In order to code this interpolation with a view to approximating the value of t for which $\theta(t) = 0$. I determined the coefficients of the interpolating polynomials for $1, t, \ldots, t^{n-1}$. I was then able to use an inbuilt function for finding the roots of polynomials given a set of coefficients and simply had to determine which root fell within the correct range. I did this for the quadratic and cubic interpolations. One could theoretically do this up to any order of polynomial but using a higher order risks creating erratic polynomials with large error spikes so in this case I kept the order of interpolation relatively low.

I then combined all 4 types of time step with my 3 types of interpolation in order to explore how the order of error convergence differed for different combinations. Table 1 shows the

results of this exploration. For all my interpolations I used a set of points symmetric about 0. For example with $n = 6$ points I have used a set of points where

$$\theta_i \begin{cases} > 0, i = 1, 2, 3 \\ < 0, i = 4, 5, 6 \end{cases}$$

Table 1: Convergence of error - Lagrange interpolation

	Euler	RK2	RK3	RK4
Linear	1st Order	2nd Order	Unclear	Unclear
Quadratic	1st Order	2nd Order	3rd Order	Unclear
Cubic	1st Order	2nd Order	3rd Order	4th Order

From table 1 we can see that the order of error convergence correlates with the order of error of the time stepping scheme. However when using a higher order time step with a lower order interpolation the log plots behaved erratically, unlike the straight line graph in figure 3. An example of this erratic behaviour can be seen in figure 4.

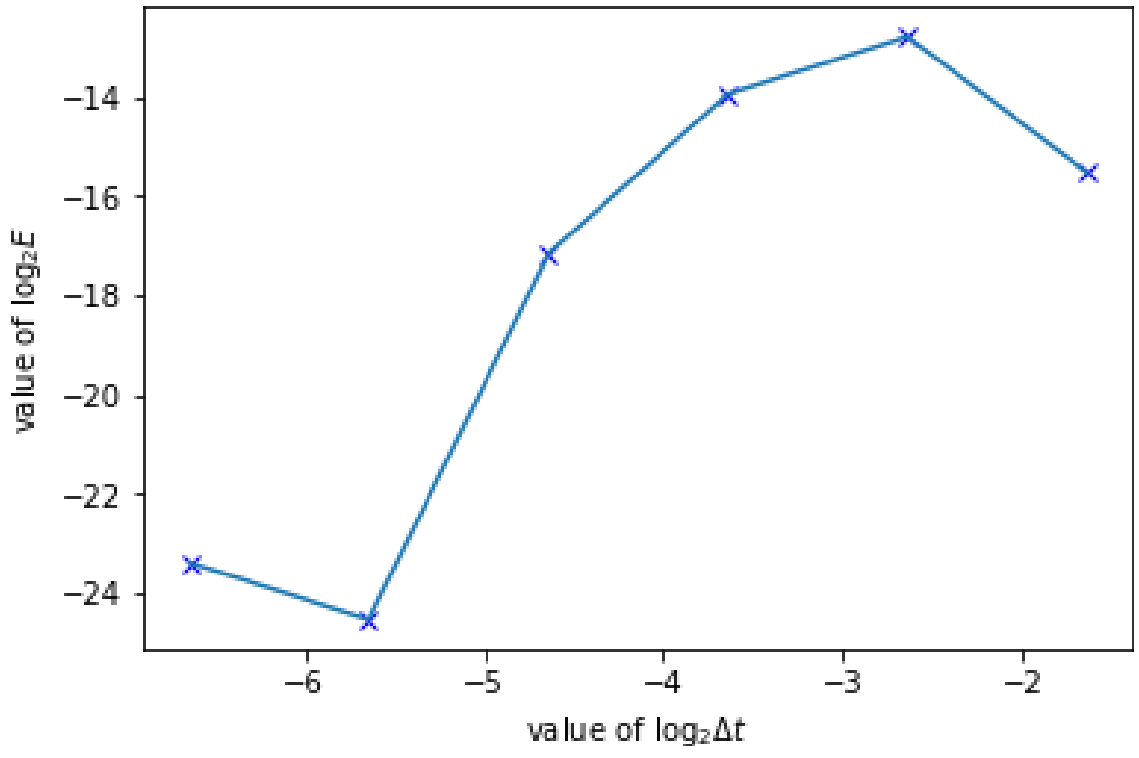


Figure 4: log plot of error of period estimate using RK3 with linear interpolation against dt

Using polyfit to interpolate

I repeated this procedure using the inbuilt python interpolation function polyfit in order to compare with my own Lagrange interpolation functions. This function uses the least

squares method - minimizing the sum of the squared residuals to find the best fitting polynomial for the data. I used this function in such a way it can be directly compared with the Lagrange interpolations, using an $n-1^{th}$ order polynomial to interpolate n points. I also included 5th and 6th order interpolation (doing this with polyfit required no extra coding whereas doing this with Lagrange interpolation would have been quite laborious.)

Table 2: Convergence of error - polyfit

	Euler	RK2	RK3	RK4
Linear	1st Order	2nd Order	Unclear	Unclear
Quadratic	1st Order	2nd Order	3rd Order	Unclear
Cubic	1st Order	2nd Order	3rd Order	4th Order
4th Order Polynomial	1st Order	2nd Order	3rd Order	4th Order
5th Order Polynomial	1st Order	2nd Order	3rd Order	4th Order

I also did a direct comparison of the values of error for Lagrange vs polyfit to compare the accuracy of the two methods. Polyfit was slightly more accurate but only after 10 decimal places so this difference is negligible.

Estimates of the period for nonlinear case
The aim of this exercise is to find a way to accurately approximate the period in the non-linear case for the values of angle a for which we don't have a solution. Below are some estimates for the period at key values of a. Figure 5 shows the period estimate against a/π with lower limit of 2π. Figure 6 is the plot for $T(a) - 2\pi$. These were made using RK4, cubic interpolation, and time step 0.001. The error for the value $a = \pi/2$ was $5.37 \cdot 10^{-9}$ which suggests these values are accurate up to 8 decimal places.

I also tested what happened to the estimate of the period as $a \to \pi$. I tested up to the value of $\frac{76799 \cdot \pi}{76800}$, for which the software gives an estimate of $T(a) = 39.5243452$. At least up until this value the estimate of the period increases with larger values of a. When I ran the software with $a = \pi$ the software did not run to completion and presumably never would.

Table 3: Estimates of period for non-linear equation

$\pi/12$	$\pi/6$	$\pi/4$	$\pi/3$	$5\pi/12$	$\pi/2$
6.31020664	6.39256800	6.53434522	6.74300142	7.03062699	7.41629871

$7\pi/12$	$2\pi/3$	$3\pi/4$	$5\pi/6$	$11\pi/12$
7.93070950	8.62606260	9.60037785	11.0722526	13.7315474

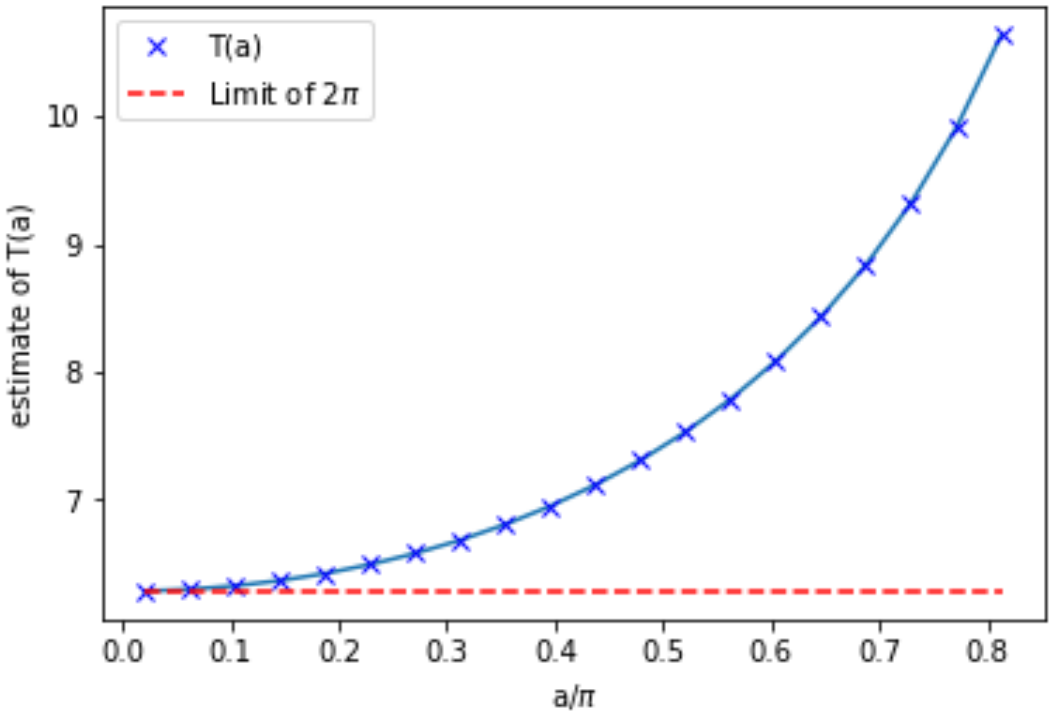


Figure 5: Estimates of T(a) against values of a

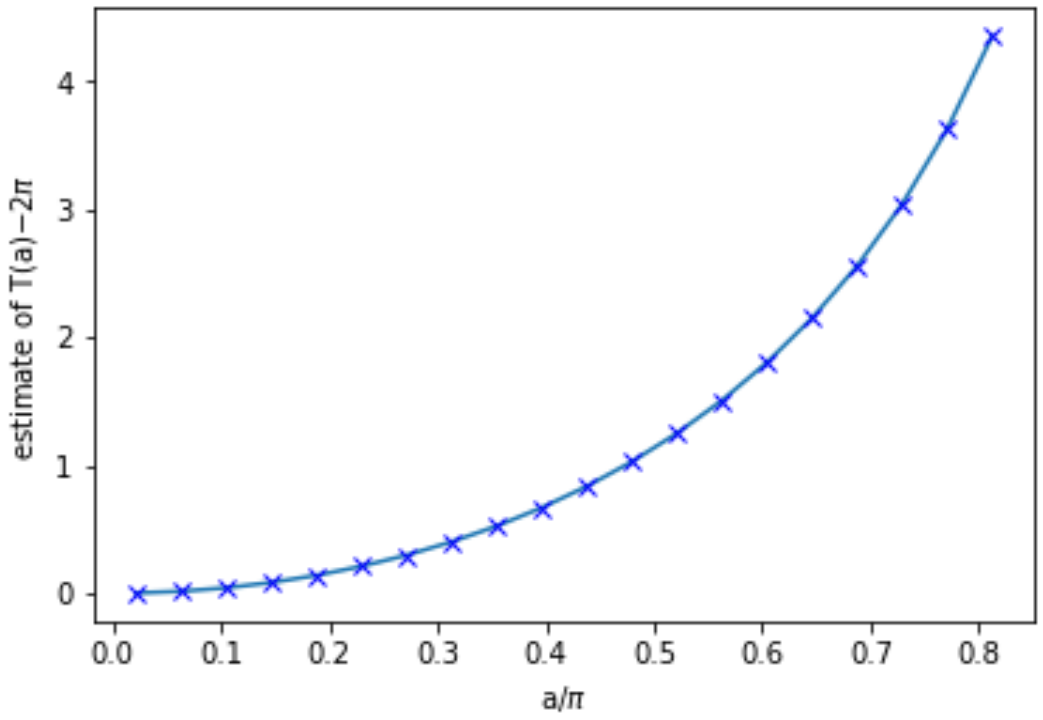


Figure 6: Estimates of T(a)-2π against values of a

3 Solving the heat equation

The heat equation models the conduction of heat down a solid metal bar of length L. We want to model $\theta(x,t)$, the temperature averaged over the cross-section at distance x along the bar.K is the thermal diffusivity constant. The equation is

$$\frac{\partial \theta}{\partial t} = K\frac{\partial^2 \theta}{\partial x^2} \tag{3.1}$$

The simplest version of this equation is when the ends of the bar are held at a fixed temperature. We set this temperature at 0. Our boundary and initial conditions are

$$\theta(0,t) = \theta(L,t) = 0, \quad \theta(x,0) = f(x)$$

The analytical solution to (3.1) is

$$\theta(x,t) = \sum_{n=1}^{\infty} A_n \sin\left(\frac{n\pi x}{L}\right) \exp\left(-K\left(\frac{n\pi}{L}\right)^2 t\right) \tag{3.2a}$$

Where

$$A_n = \frac{2}{L}\int_0^L f(x)\sin\left(\frac{n\pi x}{L}\right) dx \tag{3.2b}$$

3.1 Deriving the analytic solution

Deriving the general solution
(3.2) is derived by separating the variables and substituting back into (3.1)

$$\theta(x,t) = U(x)G(t) \Rightarrow \frac{G'(t)}{KG(t)} = \frac{U''(x)}{U(x)}$$

The left hand side and right hand side of the resulting equation must be equal to some constant $-\lambda$. We now have 2 differential equations

$$G'(t) = -\lambda K G(t), \;\; U''(x) = -\lambda U(x).$$

The initial conditions are $G(t)U(0) = G(t)U(L) = 0$. Having $G(t) = 0$ would give the trivial solution so let $U(0) = U(L) = 0$. We use this to find a solution for $U(x)$. There is only a non-trivial solution when $\lambda > 0$

$$U(x) = a_1 \sin(\sqrt{\lambda}x) + a_2 \cos(\sqrt{\lambda}x)$$
$$U(0) = a_2 = 0, \quad U(L) = a_1 \sin(\sqrt{\lambda}x) = 0 \Rightarrow \sqrt{\lambda}x = n\pi, \;\; n = 1,2,3\ldots$$

We can thus define λ_n and $U_n(x)$ for this case (a_1 gets absorbed into another constant later)

$$\lambda_n = \left(\frac{n\pi}{L}\right)^2, \;\; U_n(x) = \sin\left(\frac{n\pi x}{L}\right), \;\; n = 1,2,3\ldots$$

The other differential equation has solution

$$G_n(t) = A_n \exp(-K\lambda_n t) = a\exp\left(-K\left(\frac{n\pi}{L}\right)^2 t\right)$$

Thus we have the solution for specific n. These are all solutions for the heat equation so we sum them to get (3.1).

$$\theta_n(x,t) = A_n \sin\left(\frac{n\pi x}{L}\right) \exp\left(-K\left(\frac{n\pi}{L}\right)^2 t\right)$$

Deriving the solution for specific f(x)
We define $f(x) = \sin(\frac{\pi x}{L})$. We find the explicit solution by for this case by substituting $f(x)$ into (2.2b).

$$A_n = \frac{2}{L}\int_0^L \sin\left(\frac{\pi x}{L}\right)\sin\left(\frac{n\pi x}{L}\right)dx$$

$A_n = 0$ except in the case $n = 1$

$A_1 = \frac{1}{L}\int_0^L \sin^2\left(\frac{\pi x}{L}\right)dx = \frac{1}{L}\int_0^L \cos(0) - \cos\left(\frac{2\pi x}{L}\right)dx = \frac{1}{L}\left[x - \frac{L}{2\pi}\sin\left(\frac{2\pi x}{L}\right)\right]_0^L = 1$

Thus the solution is just $\theta_1(x,t)$.

$$\theta(x,t) = \sin\left(\frac{\pi x}{L}\right)\exp\left(-K\left(\frac{\pi}{L}\right)^2 t\right) \tag{3.3}$$

3.2 Numerical solution using finite-difference method

We now set $L = 1$, $K = 1$ and set about solving this problem numerically using a simple finite-difference method, as described by Recktenwald (2011). We do this in the domain $x \in [0,1]$ for a finite set of N points where $\delta x = \frac{1}{N-1}$ and with time step δt. We also define the Courant Number, $C = \frac{K\delta t}{(\delta x)^2}$.

We approximate the time derivative using a first-order forward difference scheme

$$\frac{\partial\theta(x,t)}{\partial t} = \frac{\theta(x,t+\delta t) - \theta(x,t)}{\delta t} + O(\delta t) \tag{3.4}$$

We approximate the space derivative using the, second order, central difference scheme

$$\frac{\partial^2\theta(x,t)}{\delta x^2} = \frac{\theta(x+\delta x,t) - 2\theta(x,t) + \theta(x-\delta x,t)}{(\delta x)^2} + O((\delta x)^2) \tag{3.5}$$

This gives us the numerical scheme (3.6), where $\theta_i^m = \theta(i\delta x, m\delta t)$

$$\theta_i^{m+1} = \theta_i^m + \frac{\delta t}{(\delta x)^2}\left[\theta_{i+1}^m - 2\theta_i^m + \theta_{i+1}^m\right] \tag{3.6}$$

I implemented this scheme in Python for $N = 5 \Rightarrow \delta x = \frac{1}{4}, \delta t = \frac{1}{48}$. I tabulated the values of the approximation, analytic solution, and error below. I have also plotted these values as a scatter graph (fig 7).

$t =$	0.25	0.25	0.25	0.75	0.75	0.75
$x =$	0.25	0.5	0.75	0.25	0.5	0.5
Analytical	0.05997	0.08480	0.05997	0.0002838	0.0004014	0.0002838
Numerical	0.05216	0.07377	0.05216	0.0004314	0.0006099	0.0004313
Error	0.00781	0.0110	0.00781	0.0001474	0.0002085	0.0001474

Table 4: Comparison of numerical and analytical solutions to 4 significant figures

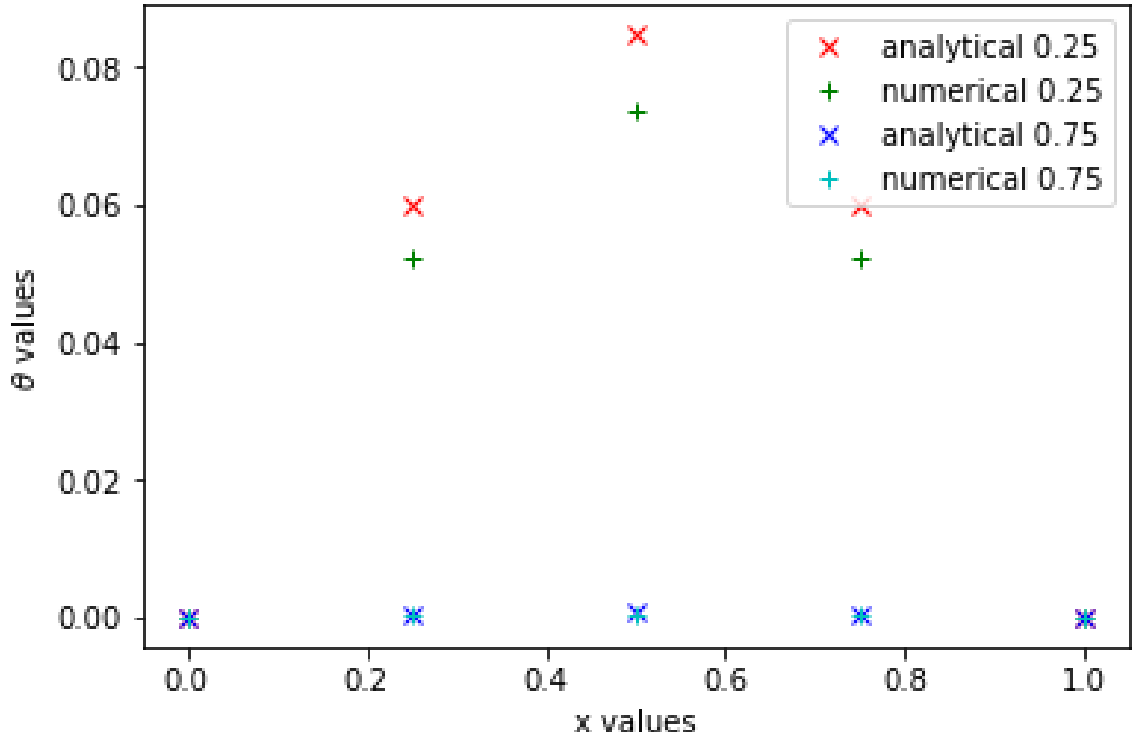


Figure 7: Comparison of the analytic and numerical solutions for $t = 0.25$ and $t = 0.75$

Root-mean-square error We define the root-mean-square error

$$E = \sqrt{\int_0^1 \mid \theta(x,t) - \theta_{an}(x,t) \mid^2 dx} \tag{3.7}$$

Where $\theta(x,t)$ is the approximation and $\theta_{an}(x,t)$ is the analytic solution. To implement this with a set of discrete points, rather than a continuous function, we use

$$E = \frac{1}{NT} \sum_i \sum_j \mid \theta(i\delta x, j\delta t) - \theta_{an}(i\delta x, j\delta t) \mid, \quad T = \frac{t_{max}}{\delta t}.$$

I calculated the root-mean-square error (RMSE), at $t = 0.5$, whilst varying the values of N and δt, keeping $C < \frac{1}{2}$.

Figure 8 shows the RMSE when varying the value of N but keeping $dt = 0.0001$. Figure 9 is the corresponding log plot of δx against RMSE and has gradient 2.29. This corresponds to a theoretical order of 2 for spatial error.

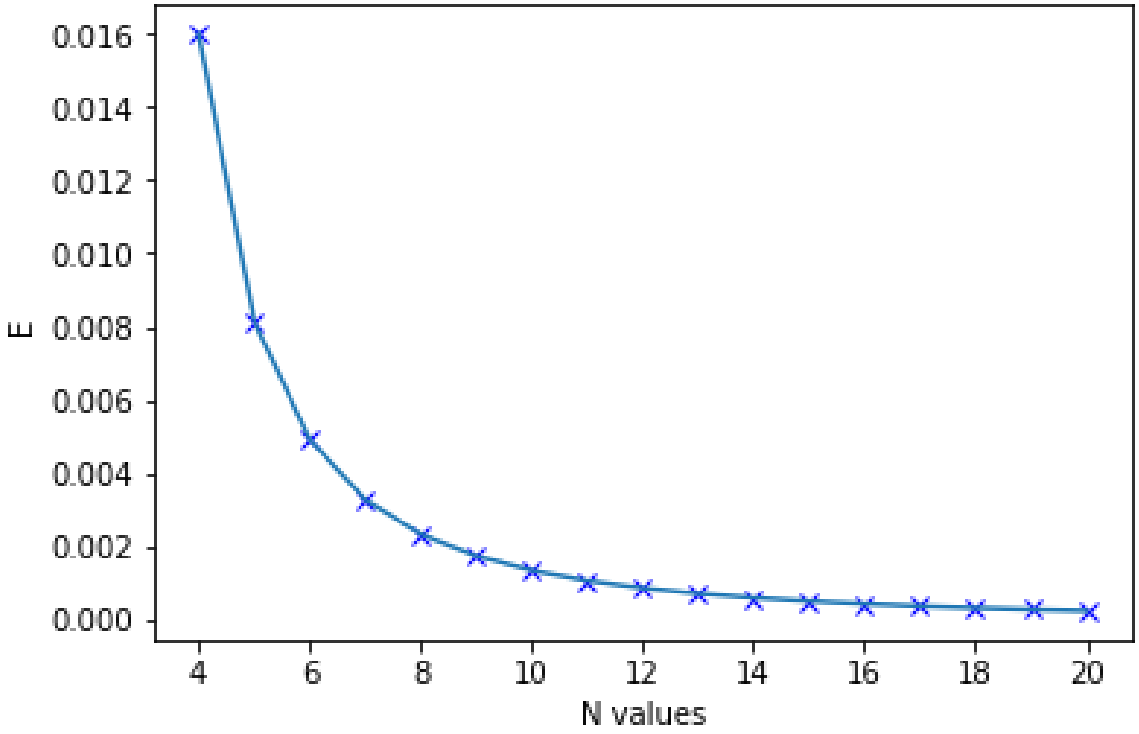


Figure 8: Error compared to varying values for N. ($dt = 0.0001$, tmax=0.5)

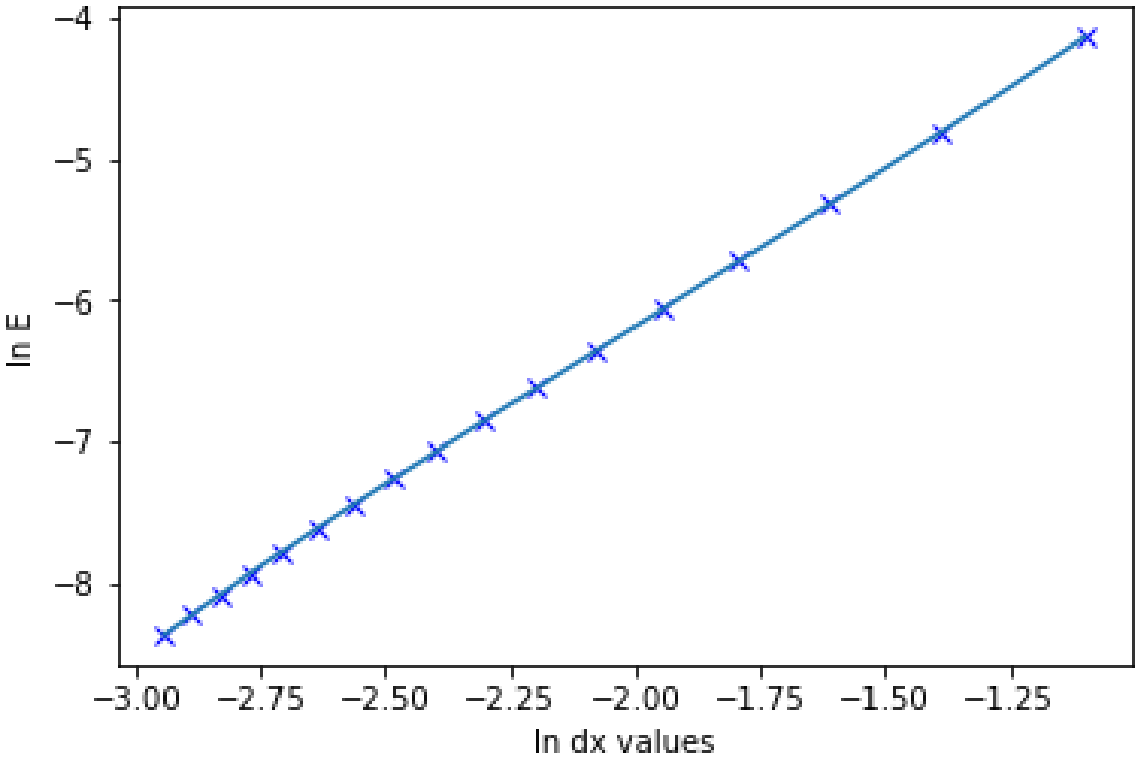


Figure 9: Natural Log plot of error against dx

Figures 10 and 11 show the result when fixing the value of N and varying the value of δt. Theoretically there is first order error for the time step, but it was quite difficult to show this in Python. For carefully chosen values of N and δt it is possible to get something close to the theoretical order of error.

For reasons outlined in the next section we keep $C < \frac{1}{2}$. Ideally we want to fix N at a fairly large value to minimize spatial error. However $C \propto (N-1)^2$. Fixing a large value of N necessitates using smaller time steps, which I believe caused machine error to become significant. Using very small time steps resulted in figure 10 where error gets smaller for larger δt. Goldberg discussed the limitations of floating point arithmetic in his 1991 paper, including the topic of rounding error which I believe is the culprit here.

However by carefully choosing N and δt, I was able to demonstrate something closer to the theoretical order of error when varying dt. This is shown in figure 11 which has a corresponding log plot with gradient 1.45.

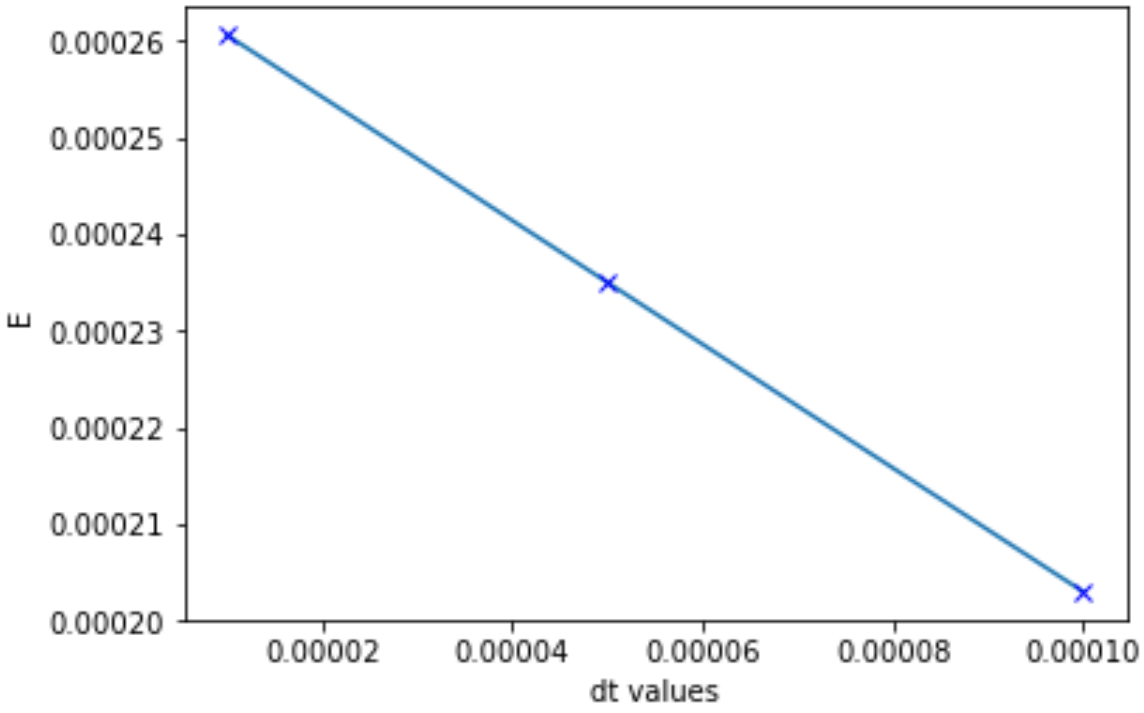


Figure 10: Error against dt - example of machine error problem

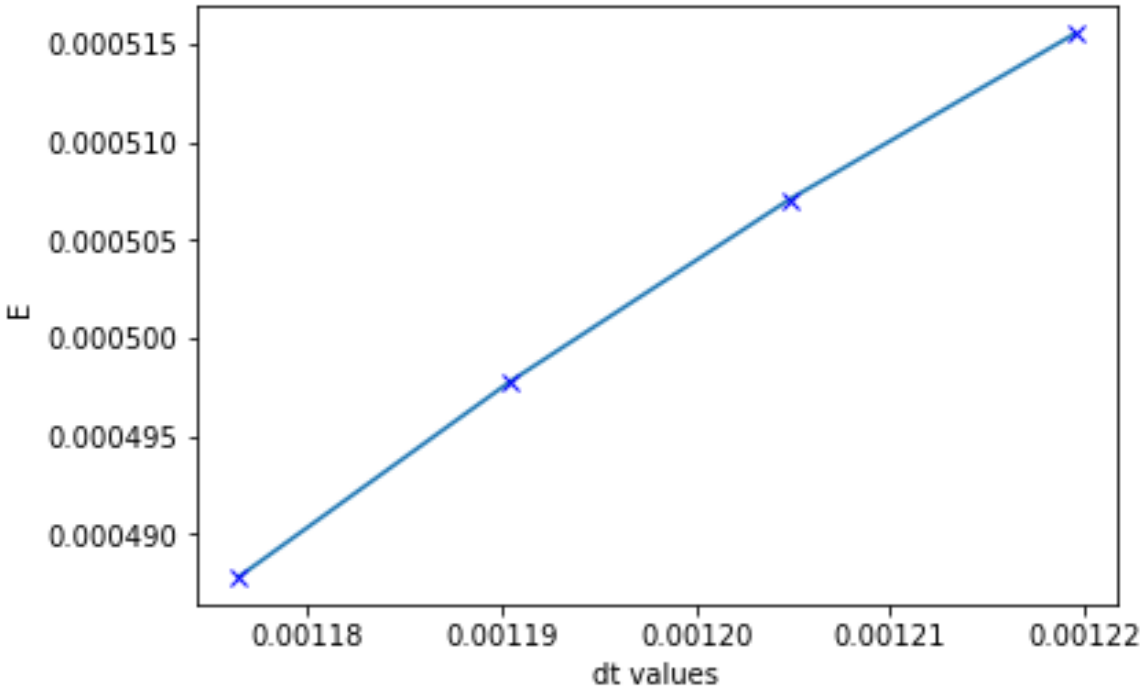


Figure 11: Error against dt for fixed N=21, tmax=0.5

Wildly oscillating numerical approximations For the specific case $N = 10, C = \frac{2}{3}, t_{max=1.0}$ the numerical solution oscillates wildly from positive to negative. This is shown in figure 12, where the θ axis is scaled so that $1 = 10^{22}$.

I thought it would be interesting to explore the different values of N and C which result in positive to negative oscillation. When $C = \frac{2}{3}$ there is no oscillation for $N < 8$. As N increases the error increases very dramatically in scale. I have tabulated the MSRE at $t = 1$ for several N values to demonstrate this.

$N =$	8	9	10	15	20	25
Error	0.0003	25.17	$3.337 \cdot 10^{7}$	$4.639 \cdot 10^{45}$	$3.023 \cdot 10^{100}$	$8.225 \cdot 10^{171}$

Table 5: average error of numerical approximations for $C = 2/3$ with different n values

I also explored what happens for different C values. I was able to get the oscillation effect for $C = 0.5005$ but not for any lower value as lower values of C correlated to larger values for N and it took too long to run the code for $N > 100$ The table below shows different values of C along with corresponding minimum values of N for which the numerical approximation oscillated.

$C =$	0.5005	0.501	0.505	0.51	0.55	0.6	$\frac{2}{3}$
$n =$	99	71	33	24	12	10	8

Table 6: minimum values for N for oscillation

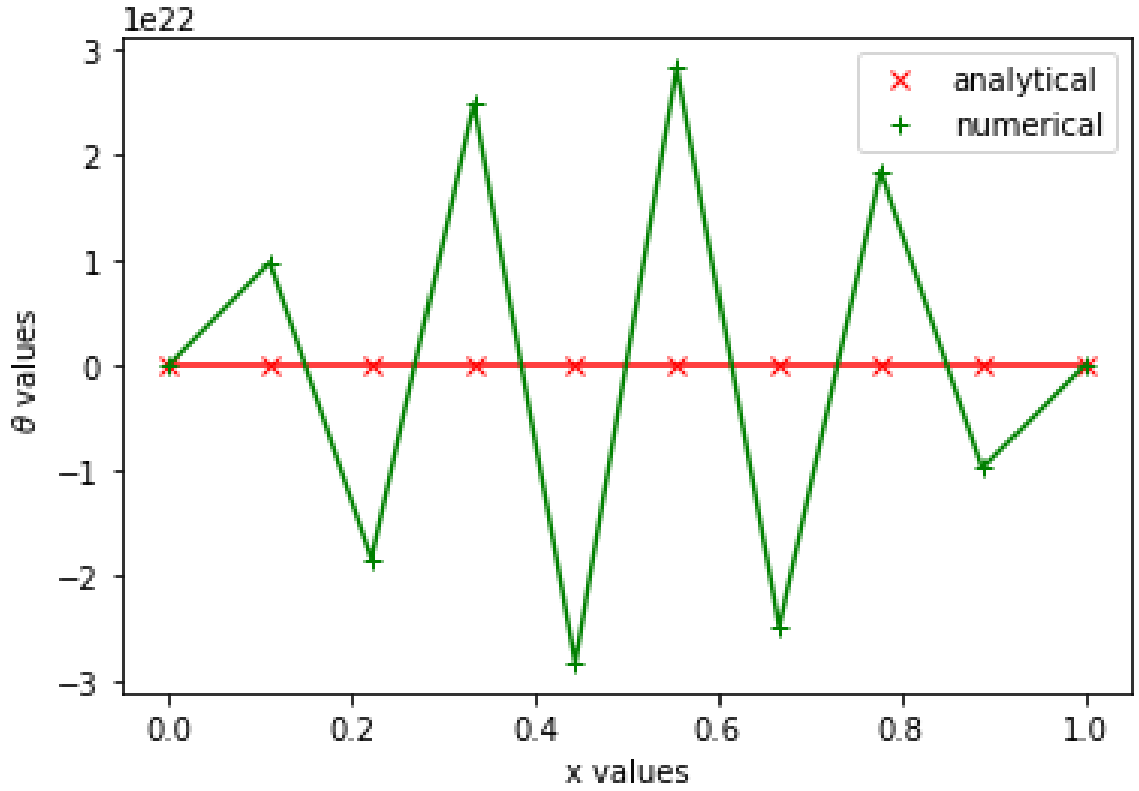


Figure 12: graph of analytical and numerical solutions for $N = 10, C = \frac{2}{3}$ at the point $t = 1$

3.3 Numerical solution using Crank-Nicholson scheme

Next I implemented the more stable and accurate Crank-Nicholson scheme, described in detail by Recktenwald (2011). This scheme approximates the time derivative using

$$\frac{\theta_i^{m+1} - \theta_i^m}{\delta t} = \frac{K}{2(\delta x)^2}\left[\frac{\partial^2 \theta^{m+1}}{\partial x^2} + \frac{\partial^2 \theta^m}{\partial x^2}\right], \tag{3.8}$$

and the spatial derivative is approximated using (3.5).

This is an implicit scheme for which the solution at $m+1$ depends on the solution at $m+1$ as well as the solution at m. The previous numerical scheme implemented was explicit, with the solution at $m+1$ only depending on m. As such when implementing this scheme we must solve a matrix equation.

In the case $N = 5$ We express the values of the interior points (those not at the boundaries for which $\theta_0^m = \theta_4^m = 0$) as the vector $\boldsymbol{\theta^m}$. We also define the spatial derivatives from equation (3.5) as the matrix D.

$$\boldsymbol{\theta^m} = \begin{pmatrix} \theta_1^m \\ \theta_2^m \\ \theta_3^m \end{pmatrix}, \quad D = \begin{pmatrix} -2 & 1 & 0 \\ 1 & -2 & 1 \\ 0 & 1 & -2 \end{pmatrix} \tag{3.9}$$

We can then express this scheme as a matrix equation

$$A_1\boldsymbol{\theta^{m+1}} = A_2\boldsymbol{\theta^m} \tag{3.10}$$

Where for

$$r = \frac{\delta t}{2(\delta x)^2}$$

$$A_1 = I_3 + D = \begin{pmatrix} 1+2r & r & 0 \\ r & 1+2r & r \\ 0 & r & 1+2r \end{pmatrix}, \quad A_2 = I_3 - rD = \begin{pmatrix} 1-2r & -r & 0 \\ -r & 1-2r & -r \\ 0 & -r & 1-2r \end{pmatrix}$$

Crank-Nicholson Root-Mean-Square Error I implemented the Crank-Nicholson scheme and have been able to demonstrate something close to the theoretical order of error $E \propto dt^2$, for the case $N = 21, \quad t_{max} = 0.5, \quad t = [\frac{1}{30}, \frac{1}{28}, \frac{1}{24}, \frac{1}{20}, \frac{1}{16}, \frac{1}{10}]$. Figure 13 shows the natural log plot of the error - it has gradient 1.94.

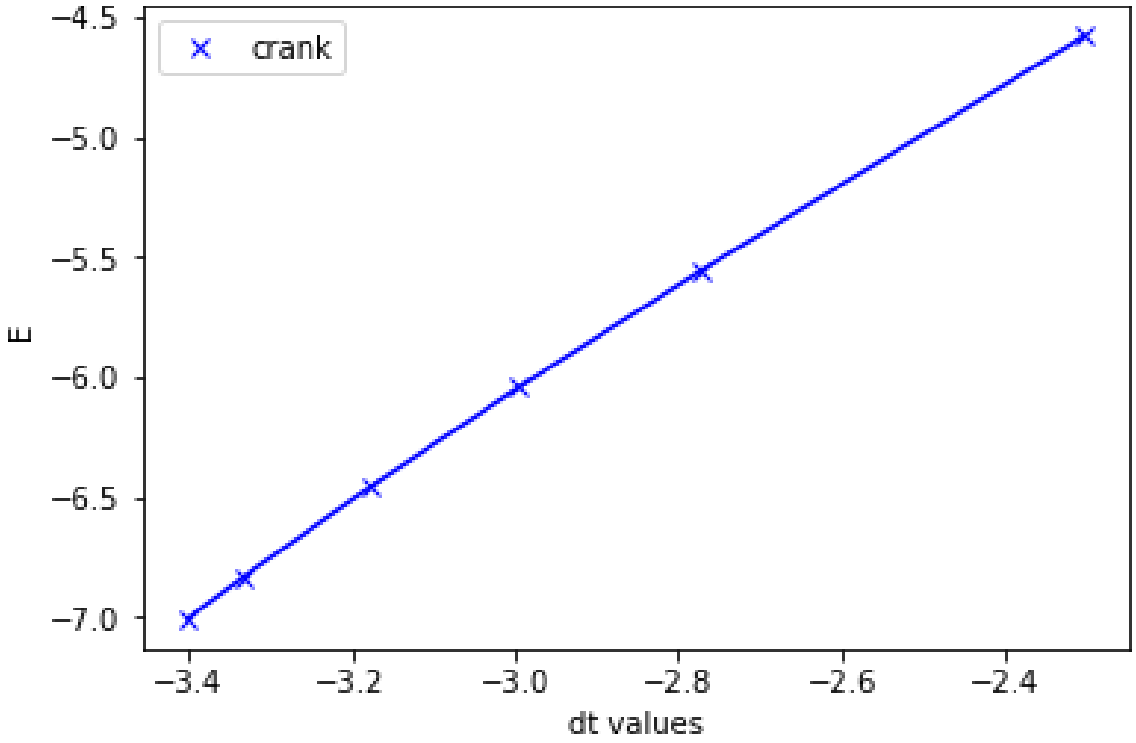


Figure 13: log plot of error of the crank scheme with varying dt

I also demonstrated second order error convergence when varying values of N. In this case the gradient of the log plot was 2.15. The values used were $\delta t = 0.0005, \quad t_{max} = 0.5, \quad N = [6, 7, 8, 9, 10, 11]$

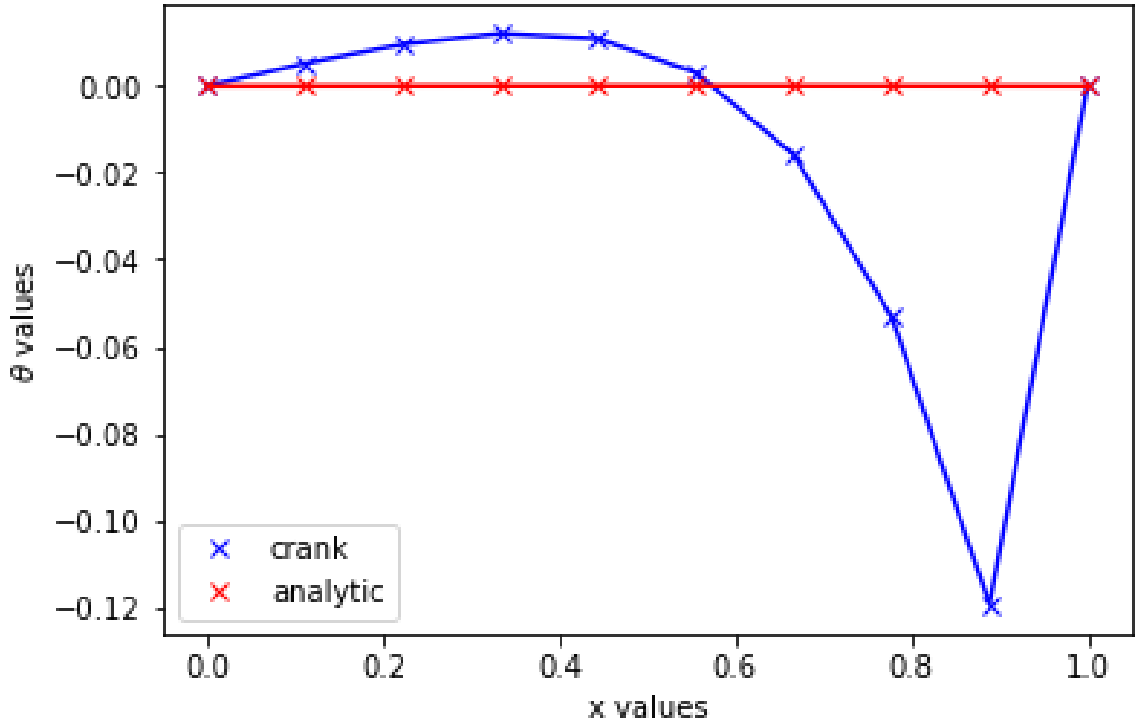


Figure 14: crank scheme for specific case $N = 10, C = 2/3$

Specific case $N = 10, C = \frac{2}{3}$ I implemented this scheme for this specific case, as I did for the forward step scheme earlier. The results are shown in figure 14. Clearly there are still some issues for this specific case but the Crank scheme is comparatively stable to the finite-difference scheme.

3.4 Crack Nicolson for first order one way wave equation

I thought it would be interesting to apply this scheme in another context so I have applied it to a simple wave equation, taking guidance from a 2006 paper written by S. Dong on the application of various finite difference methods to this equation.

The wave equation, with initial conditions is below.

$$u_t = cu_x, \quad c > 0, \quad u(x, 0) = f(x), \quad -\infty < x < \infty \tag{3.11}$$

The explicit solution is

$$u(x, t) = f(x + ct)$$

I applied this scheme for the, arbitrarily chosen, initial condition $f(x) = sin(2x)$. For the smaller values of c and t the approximation was relatively close to the analytic solution, but for larger values the error was significant. I believe this resulted from the fact I was implementing the scheme for a finite section of the x-plane, without the application of any boundary conditions. Since this is a one-way wave equation, I believe in the approximation the wave wanted to keep moving towards the right but was restricted by the boundary of the x-plane resulting in this effect. In order to approximate more effectively, one might try implementing on a wider range for x.

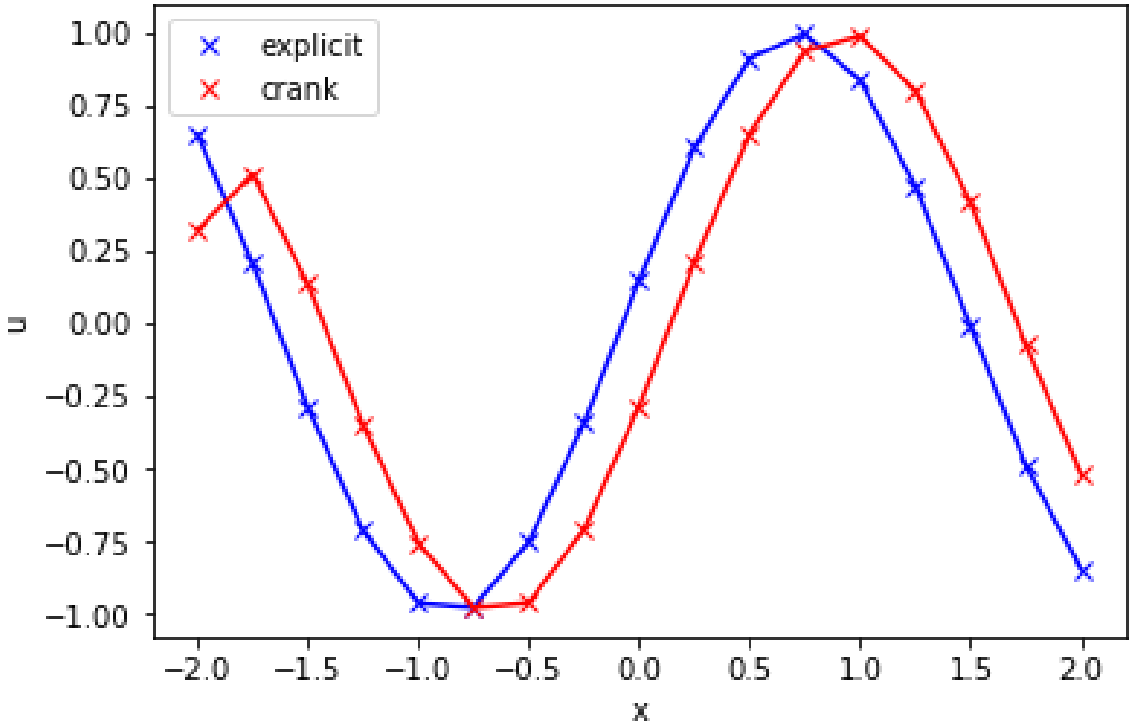


Figure 15: CNS and explicit solutions for c=0.5, t=0.3

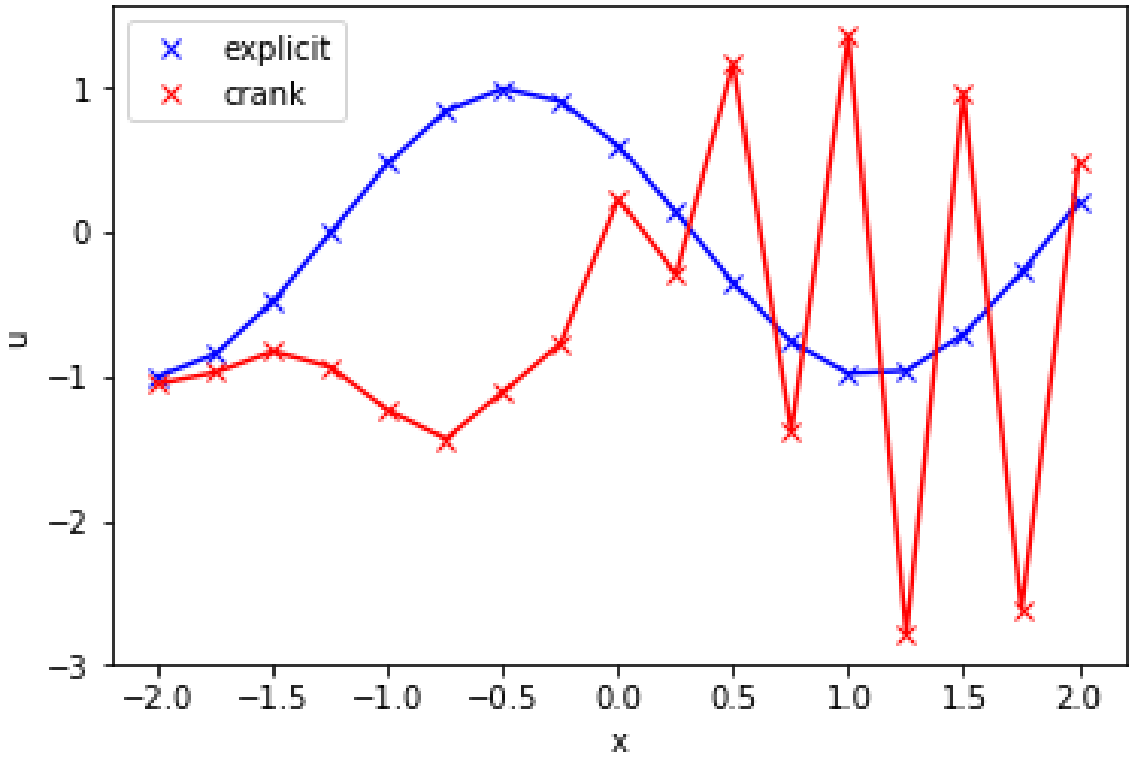


Figure 16: CNS and explicit solutions for c=0.5, t=5

4 Evolving Planetary Orbits Using Geometric Integrators

In this section we consider the gravitational two-body problem for a planet in orbit about a star of mass M, with gravitational constant G. $\boldsymbol{r}$ is the position vector of the planet relative to the star. We also have $\boldsymbol{r} = r \cdot \hat{\boldsymbol{r}}$, where $\hat{\boldsymbol{r}}$ is the corresponding unit vector and r is a constant.

$$\frac{d^2\boldsymbol{r}}{dt^2} = \boldsymbol{F}(\boldsymbol{r}) = -\frac{GM}{r^2} \cdot \boldsymbol{r} \tag{4.1}$$

We can rewrite (3.1) as a system of 2 first-order equations, for which $\boldsymbol{v}$ is the orbital velocity.

$$\dot{\boldsymbol{r}} = \boldsymbol{v}, \quad \dot{\boldsymbol{v}} = \boldsymbol{F}(\boldsymbol{r}) = -GM \cdot \frac{\hat{\boldsymbol{r}}}{r^2} \tag{4.2}$$

4.1 Analytic Solution

We have semi-major axis a and eccentricity e. We have made the assumption that $GM = a = 1$. Hence the period of the entire orbit is 2π. We assume the motion is in the $x - y$ plane and that the orbit starts from the furthest distance from the planet to the star. This gives us the intial conditions

$$x = (1 + e), \quad y = 0, \quad v_x = 0, \quad v_y = \sqrt{\frac{1 - e}{1 + e}} \tag{4.3}$$

We have the analytical solution given by

$$x = \cos E + e, \quad y = b \sin E, \quad b = \sqrt{1 - e^2}, \quad E \in [0, 2\pi] \tag{4.4}$$

4.2 Numerical Solutions

I implemented four time stepping schemes to approximate the analytical solution. The schemes are Forward Euler (order 1), Modified Euler (order 2), Leapfrog (order 2), RK4 (order 4) with respective equations listed below. All schemes were intially implemented with $e = 0.5$, using 1000 force evaluations per orbit. Figure 17 shows the plots of these approximations in this case. Forward Euler is the worst but there's little visible difference between the other 3 schemes.

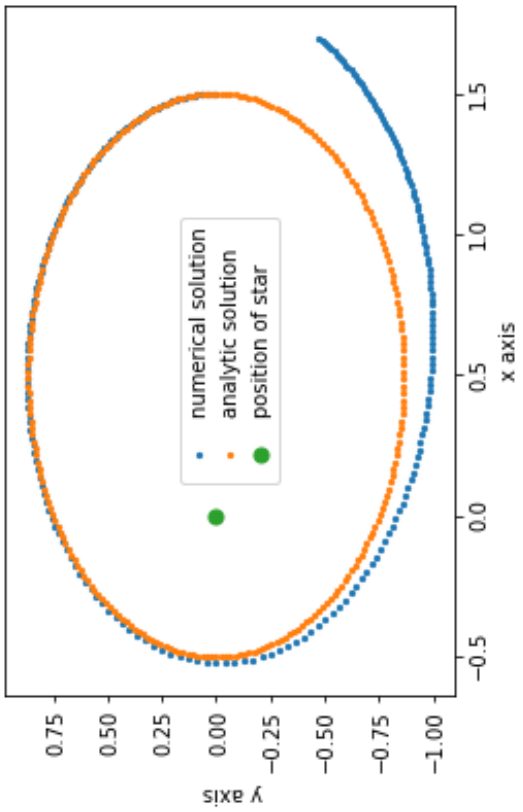
$$\boldsymbol{r}_{n+1} = \boldsymbol{r}_n + h \cdot \boldsymbol{v}_n, \quad \boldsymbol{v}_{n+1} = \boldsymbol{v}_n + h \cdot \boldsymbol{F}(\boldsymbol{r}_n) \tag{4.5}$$

$$\boldsymbol{r}_{n+1} = \boldsymbol{r}_n + h \cdot \boldsymbol{v}_n, \quad \boldsymbol{v}_{n+1} = \boldsymbol{v}_n + h \cdot \boldsymbol{F}(\boldsymbol{r}_{n+1}) \tag{4.6}$$

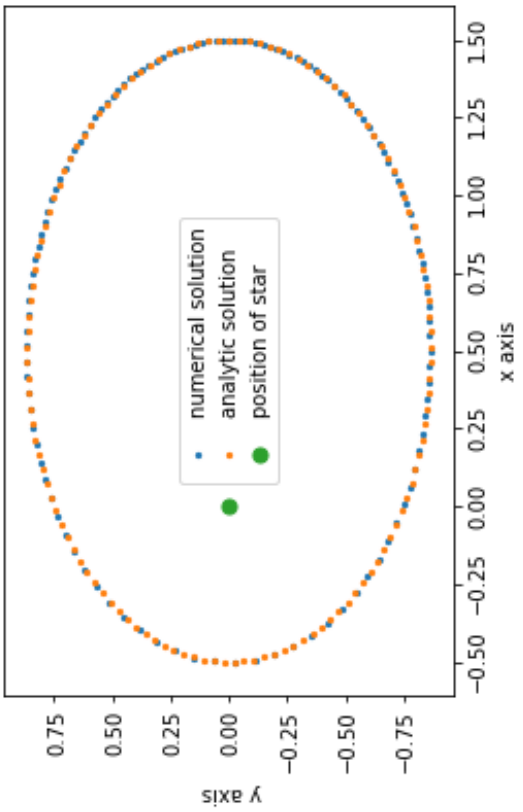
$$\boldsymbol{r'} = \boldsymbol{r}_n + \frac{h}{2}\boldsymbol{v}_n, \quad \boldsymbol{v}_{n+1} = \boldsymbol{v}_n + h \cdot \boldsymbol{F}(\boldsymbol{r'}), \quad \boldsymbol{r}_{n+1} + \frac{h}{2} \cdot \boldsymbol{v}_{n+1} \tag{4.7}$$

$$\boldsymbol{w} = (\boldsymbol{r}, \boldsymbol{v})^T, \quad \dot{\boldsymbol{w}} = \boldsymbol{f}(\boldsymbol{w}, t), \quad \boldsymbol{w}_{n+1} = \boldsymbol{w}_n + \tfrac{1}{6}(\boldsymbol{k}_1 + 2\boldsymbol{k}_2 + 2\boldsymbol{k}_3 + \boldsymbol{k}_4), \tag{4.8}$$

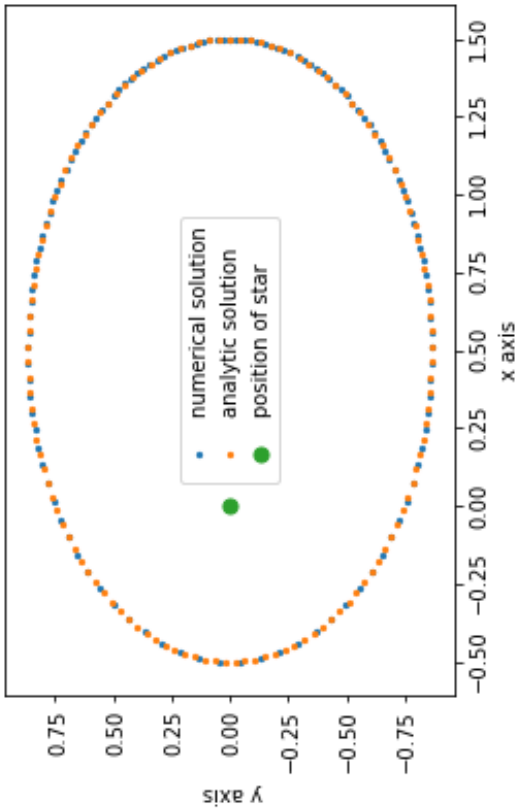
$$\boldsymbol{k}_1 = h\boldsymbol{f}(\boldsymbol{w}, t), \quad \boldsymbol{k}_2 = h\boldsymbol{f}(\boldsymbol{w} + \tfrac{1}{2}\boldsymbol{k}_1, t + \tfrac{1}{2}h)$$
$$\boldsymbol{k}_3 = h\boldsymbol{f}(\boldsymbol{w} + \tfrac{1}{2}\boldsymbol{k}_2, t + \tfrac{1}{2}h), \quad \boldsymbol{k}_4 = h\boldsymbol{f}(\boldsymbol{w} + \boldsymbol{k}_3, t + h)$$



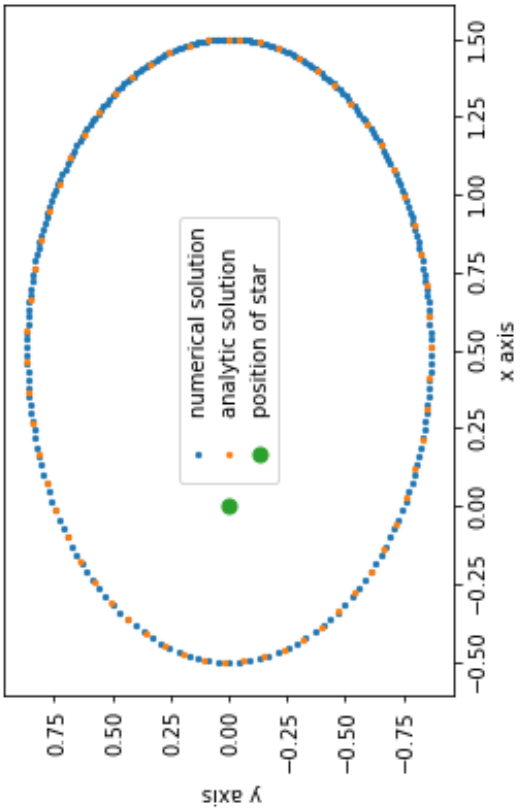
(a) Euler



(b) Modified Euler



(c) Leapfrog



(d) RK4

Figure 17: Comparisons of Approximations to Analytical Solutions, $e = 0.5$, 1 orbit

4.3 Other Eccentricities

When we change the value of e and run the approximation for several orbits the differences between the schemes become clearer. Figure 18 shows the approximations for $e = 0.9$ with $N = 5$ orbits.

4.4 Energy and Angular Momentum

Energy and Angular Momentum are defined below.

$$E = \frac{1}{2}(v_x^2 + v_y^2) - \frac{GM}{r}, \quad L = xv_y - yv_x$$

Both of these values are conserved for a Keplerian orbit - meaning the derivative with respect to t is 0.

To show conservation of energy first note $r = \sqrt{(x^2 + y^2)}$

$$\frac{dE}{dt} = \frac{1}{2}(2\dot{v_x}v_x + 2\dot{v_y}v_y) - GM\frac{d}{dt}\left(\frac{1}{r}\right) = \frac{-GM}{r^3}(xv_x + yv_y) - GM \cdot \frac{-(xv_x + yv_y)}{r^3} = 0$$

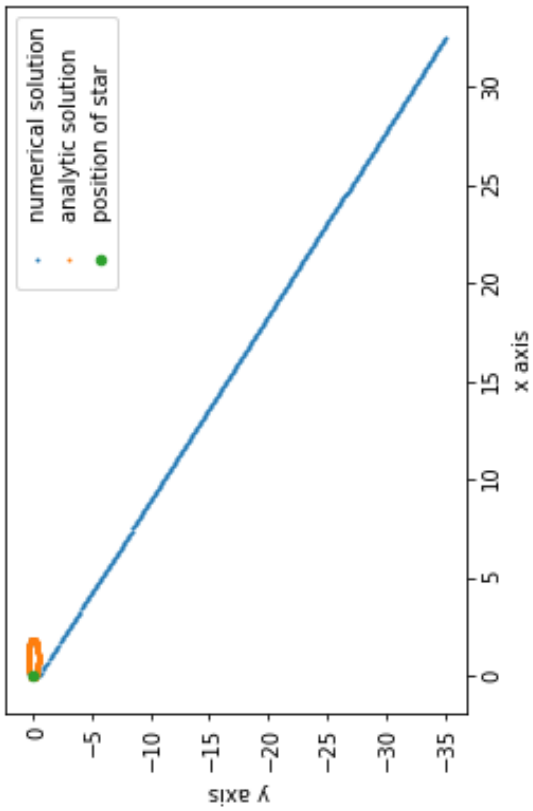
And for angular momentum

$$\frac{dL}{dt} = \dot{x}v_y + x\dot{v_y} - \dot{y}v_x - y\dot{v_x} = v_xv_y + x\frac{GM}{r^3}y - v_yv_x - y\frac{GM}{r^3}x = 0$$

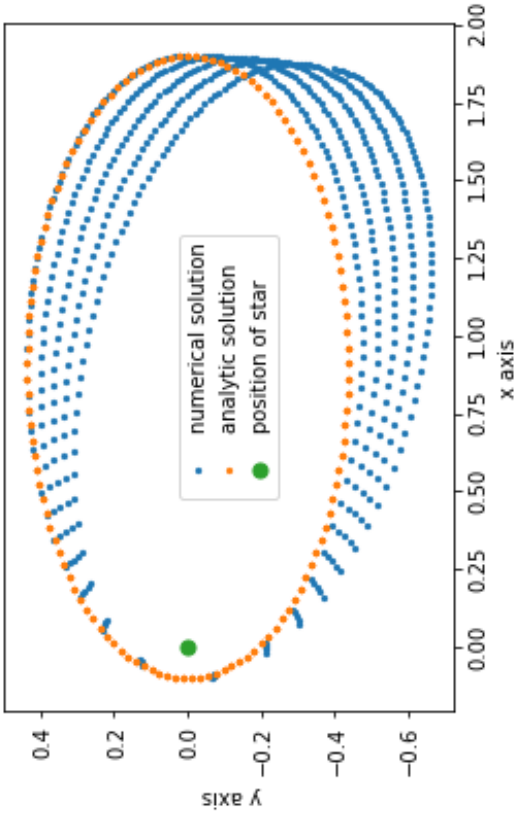
I ran each of the 4 numerical methods for $N = 100$ orbital periods, with $e = 0.9$ and 300 force evaluations per orbit, and calculated the fractional error for both energy and angular momentum as a function of t. This is shown in plots 19 and 20. The fractional error is defined below

$$\frac{|E - E_0|}{|E_0|}, \quad \frac{|L - L_0|}{|L_0|}$$

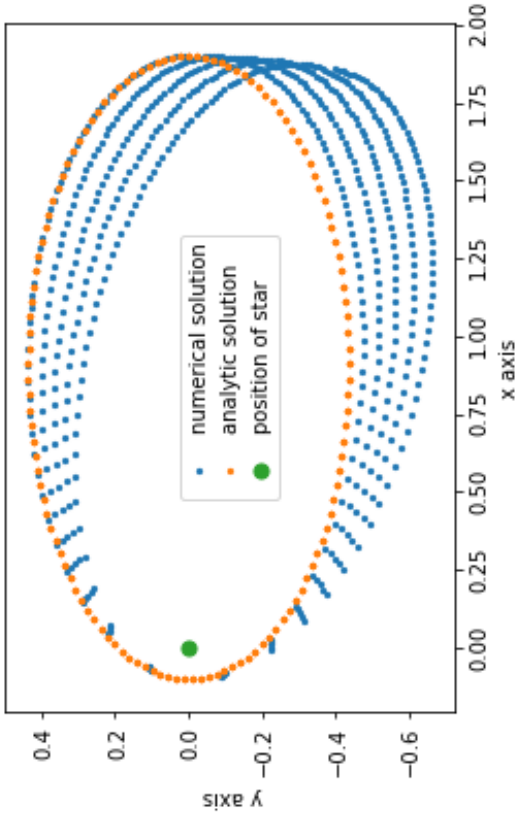
These plots show that the fractional error for both energy and angular momentum is much lower for the second-order schemes than for RK4 and forward Euler. This is interesting as generally one expects RK4 to be the "better" scheme, because of its higher order. This does fall largely in line with what we see in figure 18, where th second order schemes are far better approximations of the orbit.



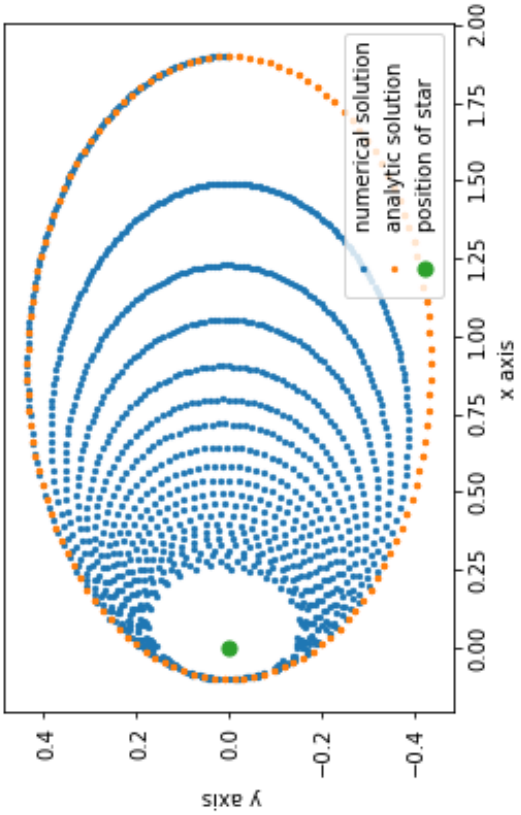
(a) Euler



(b) Modified Euler



(c) Leapfrog



(d) RK4

Figure 18: Comparisons of Approximations to Analytical Solutions, $e = 0.9$, 5 orbits

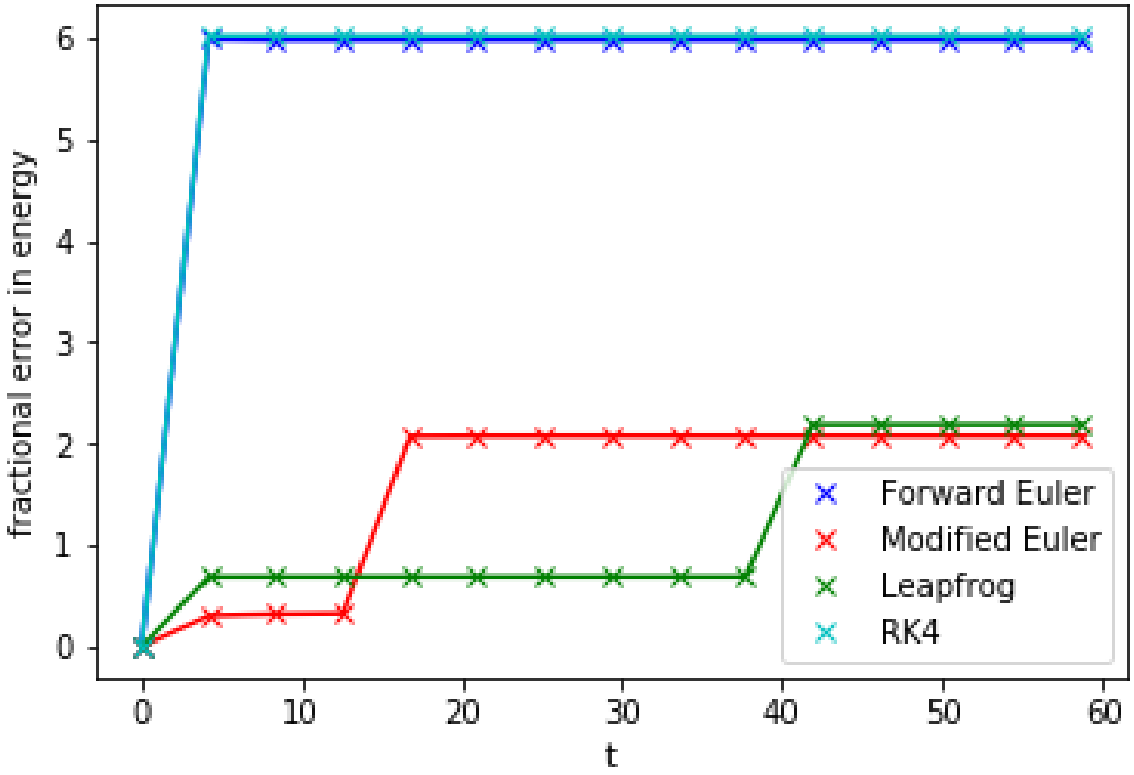


Figure 19: Fractional Error of Energy, $e = 0.9$

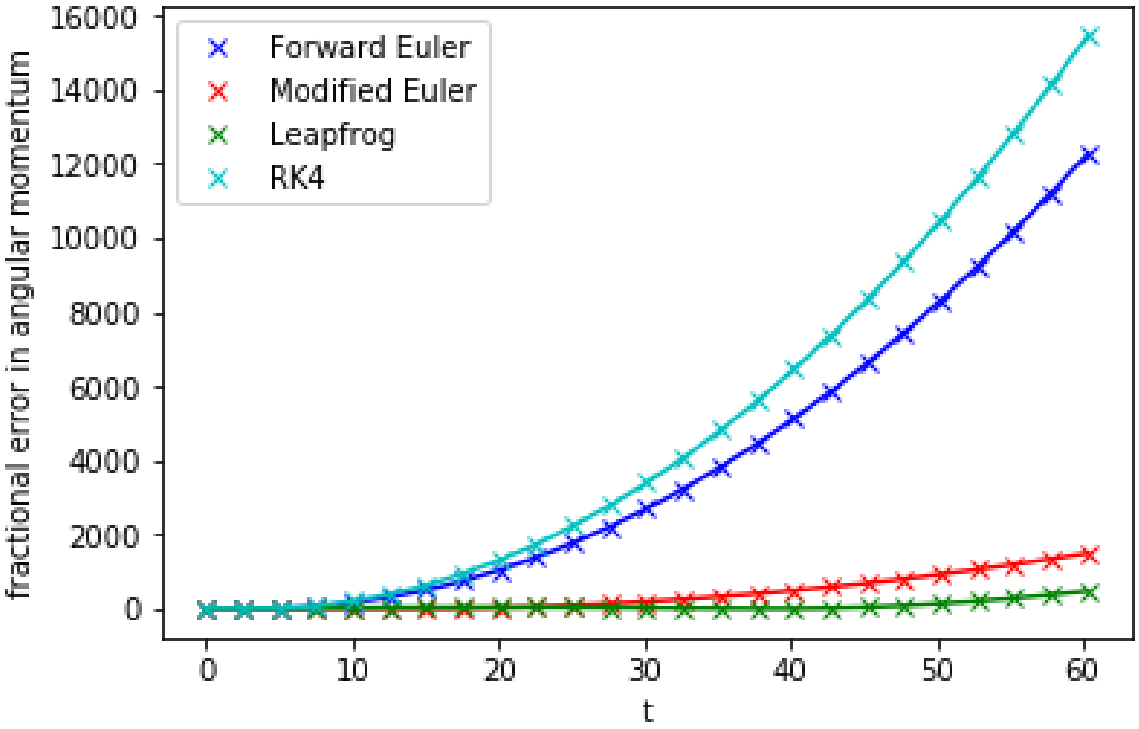


Figure 20: Fractional Error of Angular Momentum

References

Burden, L. and Faires, D. 2001. *Numerical Analysis.* 7th ed. Pacific Grove, California: Brooks/Cole

Wikipedia User: Chetvorno. 2008. *Simple Gravity Pendulum.* [Online]. [Accessed: 23rd January 2018]. Available from: `https://commons.wikimedia.org`

Dong, S. 2006. Finite Difference Methods for the Hyperbolic Wave Partial Differential Equations. [Online]. [Accessed: 17th March 2018]. Available from: `https://w3.pppl.gov/m3d/1dwave/2006-04-12_18.085_Wave.pdf`

Epperson, J. F. 2002. *An Introduction to Numerical Methods and Analysis.* New York, New York: John Wiley and Sons, Inc.

Goldberg, D. 1991. What Every Computer Scientist Should Know About Floating-Point Arithmetic. *ACM Computing Surveys*. **23**(1), pp.5-48.

Matplotlib. 2018. Matplotlib - Pyplot Summary. [Online]. [Accessed: 17th March 2018]. Available from: `https://matplotlib.org/api/pyplot_summary.html`

SciPy. 2018. NumPy v1.14 Manual. [Online]. [Accessed: 17th March 2018]. Available from: `https://docs.scipy.org/doc/numpy/index.html`

Recktenwald, G. W. 2011. Finite-Difference Approximations to the Heat Equation. [Online]. [Accessed: 19th March 2018]. Available from: `http://www.nada.kth.se/~jjalap/numme/FDheat.pdf`

A Appendix - Python Code

A.1 Pendulum

Listing 1: Subroutines

```
def theta_dot(theta,nonlin):

import numpy as np

dtheta=0*theta # initialise as array of correct size

dtheta[0]=theta[1]
if nonlin == 0:
dtheta[1]=-theta[0]
else:
dtheta[1]=-np.sin(theta[0])

return dtheta

def step_euler(theta,dt,nonlin)
# Euler time step for theta[0,1] by time dt

# calculate time derivative
dtheta=theta_dot(theta,nonlin)

# update
theta_out=theta+dt*dtheta

return theta_out

def step_rk2a(theta,dt,nonlin):
# RK2 time step for theta[0,1] by time dt

# calculate time derivative
dtheta1=theta_dot(theta,nonlin)

# first trial step
theta1=theta+dt*dtheta1

# calculate time derivative 2
dtheta2=theta_dot(theta1,nonlin)

# update
theta_out=theta+0.5*dt*(dtheta1+dtheta2)

return theta_out

def step_rk3(theta,dt,nonlin):
#RK3 time step for theta[0,1] by time dt

k1=theta_dot(theta,nonlin)
```

```
k2=theta_dot(theta+k1*dt*0.5,nonlin)

k3=theta_dot(theta-k1*dt+2*k2*dt,nonlin)

theta_out=theta+dt*(k1+4*k2+k3)/6

return theta_out

def step_rk4(theta,dt,nonlin):
#RK4 time step for theta[0,1] by time dt

k1=theta_dot(theta,nonlin)
k2=theta_dot(theta+k1*dt*0.5,nonlin)
k3=theta_dot(theta+k2*dt*0.5,nonlin)
k4=theta_dot(theta+k3*dt,nonlin)

#update
theta_out=theta+dt*(k1+2*k2+2*k3+k4)/6

return theta_out

def test_rk(dt,rk):
import numpy as np
#returns error of different RKs with different time steps
#starting angle is 1, linear equation only, tmax=5
#dt time step
#rk 1, 2, 3, or 4

# initialising variables
t=0                     # start time at 0
theta=np.array([1,0])   # theta[0]=theta, theta[1]=\dot theta
n=0                     # number of steps taken

if rk==1:
timestep=step_euler
elif rk==2:
timestep=step_rk2a
elif rk==3:
timestep=step_rk3
elif rk==4:
timestep=step_rk4

nsteps=round(5/dt) # number of steps

while n < nsteps:

theta=timestep(theta,dt,0)

t=t+dt
n=n+1
```

```
theta_exact=np.cos(t)
error=np.abs(theta_exact-theta[0])
return(error)

def linin(tr,thetar):

#use linear interpolation to work out quarter step

theta=thetar[-3:-1]
t=tr[-3:-1]

tquart = t[0] - theta[0]*(t[1]-t[0])/(theta[1]-theta[0])

#return full step approximation
return tquart*4

def quadin(tr,thetar):
import numpy as np
#lagrange interpolation with 3 points

#slice the arrays to get final 3 points
t=tr[-4:-1]
theta=thetar[-4:-1]

#coefficients of the polynomial
coeff2=0
coeff1=0
coeff0=0

for i in range(0,3):
#iterate to get coefficient for x^2
coeff2=coeff2+(theta[i]/((t[i]-t[np.mod(i+1,3)])\
*(t[i]-t[np.mod(i+2,3)])))

#iterate to get coefficient for x
coeff1=coeff1-theta[i]*(t[np.mod(i+1,3)]+t[np.mod(i+2,3)])\
/((t[i]-t[np.mod(i+1,3)])*(t[i]-t[np.mod(i+2,3)]))

#iterate to get coefficient for 1
coeff0=coeff0+(theta[i]*t[np.mod(i+1,3)]*t[np.mod(i+2,3)])\
/((t[i]-t[np.mod(i+1,3)])*(t[i]-t[np.mod(i+2,3)]))

#find roots of polynomial
root=np.polynomial.polynomial.polyroots([coeff0,coeff1,coeff2])

#return root in correct range
if t[1]<root[0]<t[2]:
return root[0]*4
elif t[1]<root[1]<t[2]:
return root[1]*4

def quadin2(tr,thetar):
```

```
import numpy as np

#lagrange interpolation with 3 points

#picking out the 3 points
theta1=thetar[-4]
theta2=thetar[-3]
theta3=thetar[-2]
t1=tr[-4]
t2=tr[-3]
t3=tr[-2]

#defining the coefficients of the quadratic

#coefficient of t^2
coeff2=theta1/((t1-t2)*(t1-t3))+\
theta2/((t2-t1)*(t2-t3))+\
theta3/((t3-t1)*(t3-t2))

#coefficient of t
coeff1=(theta1*(-1)*(t2+t3))/((t1-t2)*(t1-t3))+\
(theta2*(-1)*(t1+t3))/((t2-t1)*(t2-t3))+\
(theta3*(-1)*(t1+t2))/((t3-t1)*(t3-t2))

#coefficient of 1
coeff0=(theta1*t2*t3)/((t1-t2)*(t1-t3))+\
(theta2*t1*t3)/((t2-t1)*(t2-t3))+\
(theta3*t1*t2)/((t3-t1)*(t3-t2))

#find roots of polynomial
root=np.polynomial.polynomial.polyroots([coeff0,coeff1,coeff2])

#return root in correct range
if t2<root[0]<t3:
return root[0]*4
elif t2<root[1]<t3:
return root[1]*4

def cubein(tr,thetar):

import numpy as np

#slice the arrays
t=tr[-4:]
theta=thetar[-4:]

#define 4 variables
coeff3=0
coeff2=0
coeff1=0
coeff0=0
```

```
#iterate to define the variables
for i in range(0,4):

coeff3=coeff3+theta[i]/((t[i]-t[np.mod(i+1,4)])\
*(t[i]-t[np.mod(i+2,4)])*(t[i]-t[np.mod(i+3,4)]))

coeff2=coeff2-theta[i]*(t[np.mod(i+1,4)]+t[np.mod(i+2,4)]\
+t[np.mod(i+3,4)])/((t[i]-t[np.mod(i+1,4)])\
*(t[i]-t[np.mod(i+2,4)])*(t[i]-t[np.mod(i+3,4)]))

coeff1=coeff1+theta[i]*(t[np.mod(i+1,4)]*t[np.mod(i+2,4)]\
+t[np.mod(i+1,4)]*t[np.mod(i+3,4)]+t[np.mod(i+2,4)]\
*t[np.mod(i+3,4)])/((t[i]-t[np.mod(i+1,4)])\
*(t[i]-t[np.mod(i+2,4)])*(t[i]-t[np.mod(i+3,4)]))

coeff0=coeff0-theta[i]*((t[np.mod(i+1,4)]*t[np.mod(i+2,4)]\
*t[np.mod(i+3,4)]))/((t[i]-t[np.mod(i+1,4)])*(t[i]-t[np.mod(i+2,4)])\
*(t[i]-t[np.mod(i+3,4)]))

#find roots of polynomial
root=np.polynomial.polynomial.polyroots([coeff0,coeff1,coeff2,coeff3])

if t[1]<root[0]<t[1]:
return root[0]*4
elif t[1]<root[1]<t[2]:
return root[1]*4
elif t[1]<root[2]<t[2]:
return root[2]*4

def polyin(tr,thetar,d):
#same as linin/ quadin/ cubein but using inbuilt functions

import numpy as np

#d is degree 1, 2, or 3.
#slicing the arrays and using polyfit to fit a polynomial
if d==5:
theta=thetar[-6:]
t=tr[-6:]
pol=np.polyfit(t,theta,5)

if d==4:
theta=thetar[-5:]
t=tr[-5:]
pol=np.polyfit(t,theta,4)

if d==3:
theta=thetar[-4:]
t=tr[-4:]
pol=np.polyfit(t,theta,3)

elif d==2:
```

```
theta=thetar[-4:-1]
t=tr[-4:-1]
pol=np.polyfit(t,theta,2)

elif d==1:
theta=thetar[-3:-1]
t=tr[-3:-1]
pol=np.polyfit(t,theta,1)

#finding roots of the polynomial
root=np.polynomial.polynomial.polyroots(pol[::-1])

#return root in the range
for i in range(0,d):
if t[0]<root[i]<t[d]:
return root[i]*4

def polyin2(tr,thetar,d):
#estimates a polynomial but across the whole data set & for any d

import numpy as np

pol=np.polyfit(tr,thetar,d)

#finding roots of the polynomial
root=np.polynomial.polynomial.polyroots(pol[::-1])

#return root in the range
for i in range(0,d):
if tr[-3]<root[i]<tr[-2]:
return root[i]*4
```

Listing 2: Timestep Error

```
import subs2ki as su, numpy as np, matplotlib.pyplot as plt, scipy as sc

#11 values for dt multiplied by factor of 2
dt=np.array([0.004,0.008,0.016,0.032,0.064,0.128])

#initializing arrays for the error plots
rk1=np.array([])
rk2=np.array([])
rk3=np.array([])
rk4=np.array([])

#run for all values of dt
for i in range(0,6):
rk1=np.append(rk1,su.test_rk(dt[i],1))
rk2=np.append(rk2,su.test_rk(dt[i],2))
rk3=np.append(rk3,su.test_rk(dt[i],3))
rk4=np.append(rk4,su.test_rk(dt[i],4))
```

```
plt.plot(np.log2(dt),np.log2(rk1),'bx')
plt.plot(np.log2(dt),np.log2(rk1),label="Euler")
plt.plot(np.log2(dt),np.log2(rk2),'yx')
plt.plot(np.log2(dt),np.log2(rk2),label="RK2")
plt.plot(np.log2(dt),np.log2(rk3),'gx')
plt.plot(np.log2(dt),np.log2(rk3),label="RK3")
plt.plot(np.log2(dt),np.log2(rk4),'rx')
plt.plot(np.log2(dt),np.log2(rk4),label="RK4")
plt.legend()
plt.xlabel("Value␣of␣" r'$\log_2␣\Delta␣t$')
plt.ylabel("Value␣of␣" r'$\log_2E$')
plt.show()

print (np.log2(rk1[5])-np.log2(rk1[0]))/(np.log2(dt[5])-np.log2(dt[0]))
print (np.log2(rk2[5])-np.log2(rk2[0]))/(np.log2(dt[5])-np.log2(dt[0]))
print (np.log2(rk3[5])-np.log2(rk3[0]))/(np.log2(dt[5])-np.log2(dt[0]))
print (np.log2(rk4[5])-np.log2(rk4[0]))/(np.log2(dt[5])-np.log2(dt[0]))

print (sc.special.gamma(1.0/4))**2/np.sqrt(np.pi)
```

Listing 3: Period Estimate Error

```
import numpy as np, subs2ki as su, scipy, matplotlib.pyplot as plt

# parameters
dt=0.01      # time step
a=np.pi/2    # initial angle
nonlin=1      # 0 or 1, for linear or nonlinear

dtr=np.array([])
errorr=np.array([])

while dt<0.5:

# initialising variables
t=0                    # start time at 0
theta=np.array([a,0])  # theta[0]=theta, theta[1]=\dot theta
n=0
i=0                    # number of steps taken

# record t and theta
tr=np.array([])
thetar=np.array([])

#repeats until the sign changes
while i<2:

#rk3 for other graph
theta=su.step_rk2a(theta,dt,nonlin)

t=t+dt
n=n+1
```

```
tr=np.append(tr,t)
thetar=np.append(thetar,theta[0])

if theta[0]<0:
i=i+1

#linear interpolation of final 2 points to get estimate of period
tapprox = su.linin(tr,thetar)

#exact period value non-linear
exactt = (scipy.special.gamma(0.25))**2/(np.sqrt(np.pi))

#error
error = abs(tapprox - exactt)

dtr=np.append(dtr,dt)
errorr=np.append(errorr,error)

dt=dt*2

#print dtr, errorr

logdt=np.log2(dtr)
logerror=np.log2(errorr)

#convergence plot
plt.plot(logdt,logerror,'bx')
plt.plot(logdt,logerror)
plt.title("")
plt.xlabel("value␣of␣" r'$\log_2\Delta␣t$')
plt.ylabel("value␣of␣" r'$\log_2␣E$')
plt.show()

print (logerror[-1]-logerror[1])/(logdt[-1]-logdt[1])
```

Listing 4: Period Estimate Plot

```
import numpy as np, subs2ki as su, scipy, matplotlib.pyplot as plt

# parameters
dt=0.001      # time step
a=(np.pi)/48    # initial angle
nonlin=1      # 0 or 1, for linear or nonlinear
tarray=np.array([])
aarray=np.array([])

while a<(5*np.pi)/6:
# initialising variables
t=0                    # start time at 0
theta=np.array([a,0])  # theta[0]=theta, theta[1]=\dot theta
n=0
```

```
i=0                    # number of steps taken

# record t and theta
tr=np.array([])
thetar=np.array([])

#repeats until the sign changes
while i<2:

theta=su.step_rk4(theta,dt,nonlin)

t=t+dt

n=n+1

tr=np.append(tr,t)
thetar=np.append(thetar,theta[0])

if theta[0]<0:
i=i+1

#linear interpolation of final 2 points to get estimate of period
tapprox = su.cubein(tr,thetar)
tarray=np.append(tarray,tapprox)
aarray=np.append(aarray,a/np.pi)

a=a+(np.pi/24)

piline = np.array([2*np.pi for j in xrange(len(tarray))])

plt.plot(aarray,tarray)
plt.plot(aarray,tarray,'bx',label='T(a)')
plt.plot(aarray,piline,'r--',label='Limit␣of␣' r'$2\pi$')
plt.xlabel("a"r'$/\pi$')
plt.ylabel("estimate␣of␣T(a)")
plt.legend()
plt.show
```

A.2 Heat Equation

Listing 5: Subroutines

```
def analyticsol(k,dt,n,dx,tmax):
#returns the analytic solution of the equation

import numpy as np

#define m as number of points in [0,tmax]
m=int(tmax/dt+1)

#defining a mxn matrix for the theta values
theta=np.zeros(shape=(m,n))
```

```
#filling the matrix with theta values
for i in range(0,m):
for j in range (0,n):
#analytic solution
theta[i][j]=np.sin(np.pi*j*dx)*np.exp(-k*(np.pi**2)*i*dt)

return theta

def numericalsol(c,dt,n,tmax):
#finding the numerical solution

import numpy as np

#define m as number of points in [0,tmax]
m=int(tmax/dt+1)

#define dx
dx=1.0/(n-1)

#defining the mxn matrix for the theta values
theta=np.zeros(shape=(m,n))

#intial conditions for theta=f(x)
for j in range(1,n-1):
theta[0][j]=np.sin(j*np.pi*dx)

#using the numerical scheme
for i in range(1,m):
for j in range(1,n-1):
theta[i][j]=(theta[i-1][j]+c*(theta[i-1][j+1]-2*theta[i-1][j]+theta[i-1][j-1]))

return theta

def crank(dt,dx,m):

import numpy as np

#defining the constant r
r=dt/(dx**2)
r=r/2

#defining D and I
D=np.matrix('2,-1,0;-1,2,-1;0,-1,2')
I=np.eye(3)

#defining the matrix A1
A1=I+r*D
A1=np.linalg.inv(A1)

#defining the matrix A2
A2=I-r*D
```

```
#defining A
A=np.dot(A1,A2)

#defining the theta matrix
theta=np.zeros(shape=(m,5))

#inputting the initial values for theta
for j in range(1,4):
theta[0,j]=np.sin(j*np.pi*dx)

#finding the next set of theta values by matrix multiplication
for i in range(0,m-1):
thetam=theta[i,1:4]

thetamplus1=np.dot(A,thetam)

theta[i+1,1:4]=thetamplus1

return theta

def crank10(dt,dx,m):

import numpy as np

#defining the constant r
r=(dt/(dx**2))/2

#defining D and I
D=np.matrix('2,-1,0,0,0,0,0,0;-1,2,-1,0,0,0,0,0;\
0,-1,2,-1,0,0,0,0;0,0,-1,2,-1,0,0,0;\
0,0,0,0,-1,2,-1,0;0,0,0,0,0,0,-1,2,-1;\
0,0,0,0,0,0,-1,2;0,0,0,0,0,0,0,-1')
I=np.eye(8)

#defining the matrix A1
A1=np.linalg.inv(I+r*D)

#defining the matrix A2
A2=I-r*D

#defining A
A=np.dot(A1,A2)

#defining the theta matrix
theta=np.zeros(shape=(m,10))

#inputting the initial values for theta
for j in range(1,9):
theta[0][j]=np.sin(j*np.pi*dx)

#finding the next set of theta values by matrix multiplication
for i in range(0,m-1):
```

```
thetam=theta[i][1:9]

thetamplus1=np.dot(A,thetam)

theta[i+1][1:9]=thetamplus1

return theta

def crankn(dt,dx,m,n):

import numpy as np

#defining the constant r
r=dt/(dx**2)
r=r/2

#defining D and I
D=np.zeros((n-2,n-2))

for i in range(0,n-2):
D[i,i]=2

for i in range (0,n-3):
D[i,i+1]=-1
D[i+1,i]=-1

I=np.eye(n-2)

#defining the matrix A1
A1=I+r*D
A1=np.linalg.inv(A1)

#defining the matrix A2
A2=I-r*D

#defining A
A=np.dot(A1,A2)

#defining the theta matrix
theta=np.zeros(shape=(m,n))

#inputting the initial values for theta
for j in range(1,n-1):
theta[0,j]=np.sin(j*np.pi*dx)

#finding the next set of theta values by matrix multiplication
for i in range(0,m-1):
thetam=theta[i,1:n-1]

thetamplus1=np.dot(A,thetam)

theta[i+1,1:n-1]=thetamplus1
```

```
return theta

def mse(c,n,dt,dx,tmax,k,approx):
#calculates root mean squared error
#approx is 0 for numericalsol, 1 for crank nicholson

if approx==0:
#numerical solutions matrix
num=numericalsol(c,dt,n,tmax)

elif approx==1:
#define m
m=int((tmax/dt)+1)

#crank matrix
num=crankn(dt,dx,m,n)

#analytical solutions matrix
ana=analyticsol(k,dt,n,dx,tmax)

#error matrix
error=abs(num-ana)

#removing the initial values
error=error[1:,1:n-1]

#mean of error values
mse=error.mean()

return mse
```

Listing 6: Analytic and Numerical

```
import subs as su
import numpy as np
import matplotlib.pyplot as plt

#initialising the variables
c=1.0/3
n=5
dx=1.0/(n-1)
dt=c*dx**2
tmax=1.0
k=1.0

#using the sub routine to get theta matrices
num=su.numericalsol(c,dt,n,tmax)
ana=su.analyticsol(k,dt,n,dx,tmax)

print num[36], ana[36], num[36]-ana[36]
```

```
#defining the range for x
x=np.arange(0,tmax+dx/2,dx)

#plotting the graphs 12*dt=0.25
plt.plot(x,ana[12],'rx',label="analytical␣0.25")
plt.plot(x,num[12],'g+',label="numerical␣0.25")
plt.xlabel("x␣values")
plt.ylabel(r'$\theta$'"␣values")
plt.plot(x,ana[36],'bx',label="analytical␣0.75")
plt.plot(x,num[36],'c+',label="numerical␣0.75")
plt.legend()
plt.show()
```

Listing 7: Mean Square Error 1

```
import subs as su
import numpy as np
import matplotlib.pyplot as plt

#varying n and keeping dt fixed
n=4
dt=0.0001
dx=1.0/(n-1)
c=dt/(dx**2)
tmax=0.5
k=1.0

#new arrays
narray=np.array([])
dxarray=np.array([])
msearray2=np.array([])

#varying n values
while n<21:

#using subroutine to calculate root-mse
mse=su.mse(c,n,dt,dx,tmax,k,0)

#append to array
msearray2=np.append(msearray2,mse)

#append n
narray=np.append(narray,n)

#append dx
dxarray=np.append(dxarray,dx)

n=n+1
dx=1.0/(n-1)
c=dt/(dx**2)

#plotting graph of n vs error
```

```
plt.plot(narray,msearray2,'bx')
plt.plot(narray,msearray2)
plt.xlabel('N values')
plt.ylabel('E')
plt.show()

#plotting log plot
plt.plot(np.log(dxarray),np.log(msearray2),'bx')
plt.plot(np.log(dxarray),np.log(msearray2))
plt.xlabel('ln dx values')
plt.ylabel('ln E')
plt.show()

print (np.log(msearray2[-1])-np.log(msearray2[0]))/(np.log(dxarray[-1])-np.log(dxarray
```

Listing 8: Mean Square Error 2

```
import subs as su, numpy as np, matplotlib.pyplot as plt
#initialising variables
n=21
dx=1.0/(n-1)
tmax=0.5
k=1.0

#defining arrays for graph
tarray=np.array([1.0/850,1.0/840,1.0/830,1.0/820])
msearray=np.array([])

#varying c values
for i in range(0,4):

c=tarray[i]/(dx**2)

#using subroutine to calculate root mean squared error
mse=su.mse(c,n,tarray[i],dx,tmax,k,0)

#append to the array
msearray=np.append(msearray,mse)

print (np.log(msearray[-1])-np.log(msearray[1]))/(np.log(tarray[-1])-np.log(tarray[1])

#plotting graph of c vs error
plt.plot(tarray,msearray)
plt.plot(tarray,msearray,'bx')
plt.xlabel("dt values")
plt.ylabel("E")
plt.show()

#rounding error example
n=21
dx=1.0/(n-1)
tmax=0.5
```

```
k=1.0

#defining arrays for graph
tarray=np.array([0.00001,0.00005,0.0001])
msearray=np.array([])

#varying c values
for i in range(0,3):

c=tarray[i]/(dx**2)

#using subroutine to calculate root mean squared error
mse=su.mse(c,n,tarray[i],dx,tmax,k,0)

#append to the array
msearray=np.append(msearray,mse)

print (msearray[-1]-msearray[1])/(tarray[-1]-tarray[1])

#plotting graph of c vs error
plt.plot(tarray,msearray)
plt.plot(tarray,msearray,'bx')
plt.xlabel("dt values")
plt.ylabel("E")
plt.show()
```

Listing 9: Oscillation

```
import subs as su, numpy as np, matplotlib.pyplot as plt

#initialising the variables
c=2.0/3
n=8
dx=1.0/(n-1)
dt=c*dx**2
tmax=1.0
k=1.0

#using the sub routine to get theta matrices
num=su.numericalsol(c,dt,n,tmax)
ana=su.analyticsol(k,dt,n,dx,tmax)

#defining the range for x
x=np.arange(0,tmax+dx/2,dx)

#plotting the graphs for tmax=1
plt.plot(x,ana[-1],'rx',label="analytical")
plt.plot(x,ana[-1],'r')
plt.plot(x,num[-1],'g+',label="numerical")
plt.plot(x,num[-1],'g')
plt.xlabel("x values")
plt.ylabel(r'$\theta$'" values")
```

```
plt.legend()
plt.show()

absl=np.abs(ana[-1]-num[-1])
print np.mean(absl)
```

Listing 10: Crank MSE

```
import numpy as np, subs as su, matplotlib.pyplot as plt

#intializing variables
n=21
dx=1.0/(n-1)
tmax=0.5
k=1.0

#defining arrays for graph
tarray=np.array([1.0/30,1.0/28,1.0/24,1.0/20,1.0/16,1.0/10])
msearray1=np.array([])

#varying c values
for i in range(0,6):

#define c
c=((n-1)**2)*tarray[i]

#using subroutine to calculate root mean squared error
mse1=su.mse(c,n,tarray[i],dx,tmax,k,1)

#append to the array
msearray1=np.append(msearray1,mse1)

#log plot
plt.plot(np.log(tarray),np.log(msearray1),'bx',label='crank')
plt.plot(np.log(tarray),np.log(msearray1),'b')
plt.xlabel('dt values')
plt.ylabel('E')
plt.legend()
plt.show()

print (np.log(msearray1[-1])-np.log(msearray1[0]))/(np.log(tarray[-1]-np.log(tarray[0]

#intializing variables
dt=0.00005
tmax=0.5
k=1.0

#defining arrays for graph
dxarray=np.array([1.0/5,1.0/6,1.0/7,1.0/8,1.0/9,1.0/10])
narray=np.array([6,7,8,9,10,11])
msearray1=np.array([])
```

```
#varying c values
for i in range(0,6):

#using subroutine to calculate root mean squared error
mse1=su.mse(c,narray[i],dt,dxarray[i],tmax,k,1)

#append to the array
msearray1=np.append(msearray1,mse1)

print (np.log(msearray1[-1])-np.log(msearray1[0]))/(np.log(dxarray[-1])-np.log(dxarray
```

Listing 11: Crank 2/3

```
import numpy as np, subs as su, matplotlib.pyplot as plt

#initializing variables
c=2.0/3
n=10
dx=1.0/(n-1)
dt=c*(dx**2)
tmax=1.0
k=1.0
m=int((tmax/dt)+1)

#running the crank scheme
crank=su.crankn(dx,dt,m,n)

#analytic solution
ana=su.analyticsol(k,dt,n,dx,tmax)

#defining x range
x=np.arange(0,1.001,dx)

#plotting numerical solution against the analytic one
plt.plot(x,crank[-1],'bx',label="crank")
plt.plot(x,crank[-1],'b')
plt.plot(x,ana[-1],'rx',label="analytic")
plt.plot(x,ana[-1],'r')
plt.xlabel('x␣values')
plt.ylabel(r'$\theta$'"␣values")
plt.legend()
plt.show()
```

Listing 12: Wave Subroutines

```
def crankn(c,dt,dx,tmax):

import numpy as np

#defining the constant r
r=c*dt/dx

#defining n, m
```

```
n=int((4/dx)+1)
m=int((tmax/dt)+1)

#defining A
A=np.zeros((n,n))

for i in range(0,n):
A[i,i]=1

for i in range (0,n-1):
A[i,i+1]=-r/4
A[i+1,i]=r/4

#defining A1
A1=np.matrix.transpose(A)

#inverting A1
A1inv=np.linalg.inv(A1)

#defining b
b=np.zeros(n)
b[0]=-r/2
b[-1]=r/2

#defining the u matrix
u=np.zeros(shape=(m,n))

#inputting the initial values for u
for j in range(0,n):
u[0,j]=np.sin(2*(-2.0+j*dx))

#finding the next set of u values by matrix multiplication
for i in range(0,m-1):
um=u[i,0:n]

mid=np.dot(A,um)

uplus1=np.dot(A1inv,mid+b)

u[i+1,0:n]=uplus1

return u
```

Listing 13: Wave Plots

```
import subs as su, numpy as np, matplotlib.pyplot as plt

c=0.5
tmax=0.3

crank=su.crankn(c,0.001,0.25,tmax)
```

```
print crank[-1]

x=np.arange(-2,2.125,0.25)

analytic=np.array([])

for i in range(0,len(x)):
ana=np.sin(2*x[i]+c*tmax)
analytic=np.append(analytic,ana)

print analytic

plt.plot(x,analytic,'bx',label="explicit")
plt.plot(x,analytic,'b')
plt.plot(x,crank[-1],'rx',label="crank")
plt.plot(x,crank[-1],'r')
plt.xlabel("x")
plt.ylabel("u")
plt.legend()
plt.show()
```

A.3 Geometric Integrators

Listing 14: Subroutines

```
#euler time step code
def euler(h,r,v):
import numpy as np

#defining empty array for new r
rplus1=np.array([0.0,0.0])

#defining empty array for new v
vplus1=np.array([0.0,0.0])

#updating the values for new r
rplus1[0]=r[0]+h*v[0]
rplus1[1]=r[1]+h*v[1]

#defining unit vector rhat and constant rconst
rconst=np.linalg.norm(r)
rhat=r/rconst

#updating the values for new v
vplus1[0]=v[0]-(h*rhat[0])/(rconst**2)
vplus1[1]=v[1]-(h*rhat[1])/(rconst**2)

#return one array thats r and v
return np.append(rplus1,vplus1)

#returns analytic solution
def analytical(e,E):
import numpy as np
```

```
#define b
b=np.sqrt(1-e**2)

#analytical solution
r=np.array([np.cos(E)+e,b*np.sin(E)])

return r

#modified euler time step code
def modeuler(h,r,v):
import numpy as np

#defining empty array for new r
rplus1=np.array([0.0,0.0])

#defining empty array for new v
vplus1=np.array([0.0,0.0])

#updating the values for new r
rplus1[0]=r[0]+h*v[0]
rplus1[1]=r[1]+h*v[1]

#defining unit vector rhat and constant rconst
rconst=np.linalg.norm(rplus1)
rhat=rplus1/rconst

#updating the values for new v
vplus1[0]=v[0]-(h*rhat[0])/(rconst**2)
vplus1[1]=v[1]-(h*rhat[1])/(rconst**2)

#return one array thats r and v
return np.append(rplus1,vplus1)

#leapfrog timestep code
def leapfrog(h,r,v):
import numpy as np

#defining empy array for r'
rprime=np.array([0.0,0.0])

#defining empty array for new r
rplus1=np.array([0.0,0.0])

#defining empty array for new v
vplus1=np.array([0.0,0.0])

#updating values for r'
rprime[0]=r[0]+(h/2)*v[0]
rprime[1]=r[1]+(h/2)*v[1]

#defining unit vector r'hat and constant r'
```

```
rconst=np.linalg.norm(rprime)
rhat=rprime/rconst

#updating values for v
vplus1[0]=v[0]-(h*rhat[0])/(rconst**2)
vplus1[1]=v[1]-(h*rhat[1])/(rconst**2)

#updating values for r
rplus1[0]=rprime[0]+(h/2)*vplus1[0]
rplus1[1]=rprime[1]+(h/2)*vplus1[1]

#return one array thats r and v
return np.append(rplus1,vplus1)

def RK4(h,r,v):
import numpy as np

#defining the constants
k1=h*v
k11=h*(-1/(np.linalg.norm(r)**3))*r

k2=h*(v+0.5*k11)
k22=h*(-1/(np.linalg.norm(r+0.5*k1)**3))*(r+0.5*k1)

k3=h*(v+0.5*k22)
k33=h*(-1/(np.linalg.norm(r+0.5*k2)**3))*(r+0.5*k2)

k4=h*(v+k33)
k44=h*(-1/(np.linalg.norm(r+k3)**3))*(r+k3)

#defining new r value
rplus1=r+((k1+2*k2+2*k3+k4)/6.0)

#defining new v value
vplus1=v+((k11+2.0*k22+2.0*k33+k44)/6.0)

#return one array thats r and v
return np.append(rplus1,vplus1)
```

Listing 15: Euler

```
import numpy as np, subs as su, matplotlib.pyplot as plt

#intializing variables
e=0.9
h=(2.0*np.pi)/1000
N=5

#defining initial conditions for t=0
r=[1.0+e,0.0]
v=[0.0,np.sqrt((1.0-e)/(1.0+e))]
```

```
#defining empy arrays for numerical solution
rarray=np.array(r)
varray=np.array(v)

#number of steps to make a full orbit
n=int((N*2*np.pi)/h)+1

#looping to determine n new points
for i in range(0,n-1):

#using the time step to move on one step
newr=su.euler(h,rarray[-2:],varray[-2:])[0:2]
newv=su.euler(h,rarray[-2:],varray[-2:])[2:4]

#addinf the new points to the array
rarray=np.append(rarray,newr)
varray=np.append(varray,newv)

#defining empty array for the analytical solution
rana=np.array(r)

for i in range(1,n):

#defining E
E=i*h

#returning new values for r
newr=su.analytical(e,E)

#appending to the array
rana=np.append(rana,newr)

#slicing into x and y values only using a fraction of the points
xnum=rarray[0:len(rarray):30]
ynum=rarray[1:len(rarray):30]
xana=rana[0:len(rana):30]
yana=rana[1:len(rana):30]

#plotting the graph
plt.scatter(xnum,ynum,s=1,label="numerical␣solution")
plt.scatter(xana,yana,s=1,label="analytic␣solution")
plt.scatter(0.0,0.0,s=20,label="position␣of␣star")
plt.xlabel("x␣axis")
plt.ylabel("y␣axis")
plt.legend()
plt.show()
```

Listing 16: Modified Euler

```
import numpy as np, subs as su, matplotlib.pyplot as plt

#intializing variables
```

```
e=0.5
h=(2.0*np.pi)/500
N=1

#defining initial conditions for t=0
r=[1.0+e,0.0]
v=[0.0,np.sqrt((1.0-e)/(1.0+e))]

#defining empy arrays for numerical solution
rarray=np.array(r)
varray=np.array(v)

#number of steps to make a full orbit
n=int((N*2*np.pi)/h)+1

#looping to determine n new points
for i in range(0,n-1):

#using the time step to move on one step
newr=su.modeuler(h,rarray[-2:],varray[-2:])[0:2]
newv=su.modeuler(h,rarray[-2:],varray[-2:])[2:4]

#addinf the new points to the array
rarray=np.append(rarray,newr)
varray=np.append(varray,newv)

#defining empty array for the analytical solution
rana=np.array(r)

for i in range(1,n):

#defining E
E=i*h

#returning new values for r
newr=su.analytical(e,E)

#appending to the array
rana=np.append(rana,newr)

#slicing into x and y values only using a fraction of the points
xnum=rarray[0:len(rarray):8]
ynum=rarray[1:len(rarray):8]
xana=rana[0:len(rana):8]
yana=rana[1:len(rana):8]

#plotting the graph
plt.scatter(xnum,ynum,s=5,label="numerical␣solution")
plt.scatter(xana,yana,s=5,label="analytic␣solution")
plt.scatter(0.0,0.0,s=50,label="position␣of␣star")
plt.xlabel("x␣axis")
plt.ylabel("y␣axis")
```

```
plt.legend()
plt.show()
```

Listing 17: Leapfrog

```
import numpy as np, subs as su, matplotlib.pyplot as plt

#intializing variables
e=0.9
h=(2.0*np.pi)/500
N=5

#defining initial conditions for t=0
r=[1.0+e,0.0]
v=[0.0,np.sqrt((1.0-e)/(1.0+e))]

#defining empy arrays for numerical solution
rarray=np.array(r)
varray=np.array(v)

#number of steps to make a full orbit
n=int((N*2*np.pi)/h)+1

#looping to determine n new points
for i in range(0,n-1):

#using the time step to move on one step
newr=su.leapfrog(h,rarray[-2:],varray[-2:])[0:2]
newv=su.leapfrog(h,rarray[-2:],varray[-2:])[2:4]

#addinf the new points to the array
rarray=np.append(rarray,newr)
varray=np.append(varray,newv)

#defining empty array for the analytical solution
rana=np.array(r)

for i in range(1,n):

#defining E
E=i*h

#returning new values for r
newr=su.analytical(e,E)

#appending to the array
rana=np.append(rana,newr)

#slicing into x and y values only using a fraction of the points
xnum=rarray[0:len(rarray):8]
ynum=rarray[1:len(rarray):8]
xana=rana[0:len(rana):8]
```

```
yana=rana[1:len(rana):8]

#plotting the graph
plt.scatter(xnum,ynum,s=5,label="numerical␣solution")
plt.scatter(xana,yana,s=5,label="analytic␣solution")
plt.scatter(0.0,0.0,s=50,label="position␣of␣star")
plt.xlabel("x␣axis")
plt.ylabel("y␣axis")
plt.legend()
plt.show()
```

Listing 18: RK4

```
import numpy as np, subs as su, matplotlib.pyplot as plt

#intializing variables
e=0.9
h=(2.0*np.pi)/250
N=5

#defining initial conditions for t=0
r=[1.0+e,0.0]
v=[0.0,np.sqrt((1.0-e)/(1.0+e))]

#defining empy arrays for numerical solution
rarray=np.array(r)
varray=np.array(v)

#number of steps to make a full orbit
n=int((N*2*np.pi)/h)+1

#looping to determine n new points
for i in range(0,n-1):

#using the time step to move on one step
newr=su.RK4(h,rarray[-2:],varray[-2:])[0:2]
newv=su.RK4(h,rarray[-2:],varray[-2:])[2:4]

#addinf the new points to the array
rarray=np.append(rarray,newr)
varray=np.append(varray,newv)

#defining empty array for the analytical solution
rana=np.array(r)

for i in range(1,n):

#defining E
E=i*h

#returning new values for r
newr=su.analytical(e,E)
```

```
#appending to the array
rana=np.append(rana,newr)

#slicing into x and y values only using a fraction of the points
xnum=rarray[0:len(rarray):2]
ynum=rarray[1:len(rarray):2]
xana=rana[0:len(rana):8]
yana=rana[1:len(rana):8]

#plotting the graph
plt.scatter(xnum,ynum,s=5,label="numerical␣solution")
plt.scatter(xana,yana,s=5,label="analytic␣solution")
plt.scatter(0.0,0.0,s=50,label="position␣of␣star")
plt.xlabel("x␣axis")
plt.ylabel("y␣axis")
plt.legend()
plt.show()

print rarray[-1]
```

Listing 19: Energy

```
import numpy as np, subs as su, matplotlib.pyplot as plt

#FORWARD EULER
#intializing variables
e=0.9
p=2*np.pi
N=100
h=p/300

#defining initial conditions for t=0
r=[1.0+e,0.0]
v=[0.0,np.sqrt((1.0-e)/(1.0+e))]

#defining empy arrays for numerical solution
rarray=np.array(r)
varray=np.array(v)

#number of steps to make a n orbits
n=int((N*p)/h)+1

#looping to determine n new points
for i in range(0,n-1):

#using the time step to move on one step
newr=su.euler(h,rarray[-2:],varray[-2:])[0:2]
newv=su.euler(h,rarray[-2:],varray[-2:])[2:4]

#addinf the new points to the array
rarray=np.append(rarray,newr)
```

```
varray=np.append(varray,newv)

#defining array of time
tarray=np.arange(0,(n-1)*h,h)

#defining E0
E0=0.5*((1.0-e)/(1.0+e))-(1/np.linalg.norm(r))

#defining fractional error in energy for forward euler
Eerror=np.array([])

for i in range(0,n-1):
Ei=0.5*((varray[2*i])**2+(varray[2*i+1])**2)-(1/np.linalg.norm([rarray[2*i],rarray[2*i

frac=(np.absolute(Ei-E0))/np.absolute(E0)

Eerror=np.append(Eerror,frac)

#MODIFIED EULER
#intializing variables
e=0.9
p=2*np.pi
N=100
h=p/150

#defining initial conditions for t=0
r=[1.0+e,0.0]
v=[0.0,np.sqrt((1.0-e)/(1.0+e))]

#defining empy arrays for numerical solution
rarray2=np.array(r)
varray2=np.array(v)

#number of steps to make a n orbits
n=int((N*p)/h)+1

#looping to determine n new points
for i in range(0,n-1):

#using the time step to move on one step
newr=su.modeuler(h,rarray2[-2:],varray2[-2:])[0:2]
newv=su.modeuler(h,rarray2[-2:],varray2[-2:])[2:4]

#addinf the new points to the array
rarray2=np.append(rarray2,newr)
varray2=np.append(varray2,newv)

#defining fractional error in energy for mod euler
Eerror2=np.array([])

for i in range(0,n-1):
Ei=0.5*((varray2[2*i])**2+(varray2[2*i+1])**2)-(1/np.linalg.norm([rarray2[2*i],rarray2
```

```
frac=(np.absolute(Ei-E0))/np.absolute(E0)

Eerror2=np.append(Eerror2,frac)

#LEAPFROG
#intializing variables
e=0.9
p=2*np.pi
N=100
h=p/150

#defining initial conditions for t=0
r=[1.0+e,0.0]
v=[0.0,np.sqrt((1.0-e)/(1.0+e))]

#defining empy arrays for numerical solution
rarray3=np.array(r)
varray3=np.array(v)

#number of steps to make a n orbits
n=int((N*p)/h)+1

#looping to determine n new points
for i in range(0,n-1):

#using the time step to move on one step
newr=su.leapfrog(h,rarray3[-2:],varray3[-2:])[0:2]
newv=su.leapfrog(h,rarray3[-2:],varray3[-2:])[2:4]

#addinf the new points to the array
rarray3=np.append(rarray3,newr)
varray3=np.append(varray3,newv)

#defining array of time
tarray2=np.arange(0,(n-1)*h,h)

#defining fractional error in energy for mod euler
Eerror3=np.array([])

for i in range(0,n-1):
Ei=0.5*((varray3[2*i])**2+(varray3[2*i+1])**2)-(1/np.linalg.norm([rarray3[2*i],rarray3

frac=(np.absolute(Ei-E0))/np.absolute(E0)

Eerror3=np.append(Eerror3,frac)

#RK4
#intializing variables
e=0.9
p=2*np.pi
N=100
```

```
h=p/75

#defining initial conditions for t=0
r=[1.0+e,0.0]
v=[0.0,np.sqrt((1.0-e)/(1.0+e))]

#defining empy arrays for numerical solution
rarray4=np.array(r)
varray4=np.array(v)

#number of steps to make a n orbits
n=int((N*p)/h)+1

#looping to determine n new points
for i in range(0,n-1):

#using the time step to move on one step
newr=su.RK4(h,rarray4[-2:],varray4[-2:])[0:2]
newv=su.RK4(h,rarray4[-2:],varray4[-2:])[2:4]

#addinf the new points to the array
rarray4=np.append(rarray4,newr)
varray4=np.append(varray4,newv)

#defining array of time
tarray3=np.arange(0,(n-1)*h,h)

#defining fractional error in energy for mod euler
Eerror4=np.array([])

for i in range(0,n-1):
Ei=0.5*((varray4[2*i])**2+(varray4[2*i+1])**2)-(1/np.linalg.norm([rarray4[2*i],rarray4

frac=(np.absolute(Ei-E0))/np.absolute(E0)

Eerror4=np.append(Eerror4,frac)

plt.plot(tarray[0:3000:200],Eerror[0:3000:200],'bx',label="Forward␣Euler")
plt.plot(tarray[0:3000:200],Eerror[0:3000:200],'b')
plt.plot(tarray2[0:1500:100],Eerror2[0:1500:100],'rx',label="Modified␣Euler")
plt.plot(tarray2[0:1500:100],Eerror2[0:1500:100],'r')
plt.plot(tarray2[0:1500:100],Eerror3[0:1500:100],'gx',label="Leapfrog")
plt.plot(tarray2[0:1500:100],Eerror3[0:1500:100],'g')
plt.plot(tarray3[0:750:50],Eerror4[0:750:50],'cx',label="RK4")
plt.plot(tarray3[0:750:50],Eerror4[0:750:50],'c')
plt.xlabel("t")
plt.ylabel("fractional␣error␣in␣energy")
plt.legend()
plt.show()
```

Listing 20: Angular Momentum

```
import numpy as np, subs as su, matplotlib.pyplot as plt

#FORWARD EULER
#intializing variables
e=0.9
p=2*np.pi
N=100
h=p/300

#defining initial conditions for t=0
r=[1.0+e,0.0]
v=[0.0,np.sqrt((1.0-e)/(1.0+e))]

#defining empy arrays for numerical solution
rarray=np.array(r)
varray=np.array(v)

#number of steps to make a n orbits
n=int((N*p)/h)+1

#looping to determine n new points
for i in range(0,n-1):

#using the time step to move on one step
newr=su.euler(h,rarray[-2:],varray[-2:])[0:2]
newv=su.euler(h,rarray[-2:],varray[-2:])[2:4]

#addinf the new points to the array
rarray=np.append(rarray,newr)
varray=np.append(varray,newv)

#defining array of time
tarray=np.arange(0,(n-1)*h,h)

#defining L0
L0=r[0]*v[1]

#defining fractional error in energy for forward euler
Lerror=np.array([])

for i in range(0,n-1):
Li=rarray[2*i]*rarray[2*i+1]-rarray[2*i+1]*varray[2*i]

frac=(np.absolute(Li-L0))/np.absolute(L0)

Lerror=np.append(Lerror,frac)

#MODIFIED EULER
#intializing variables
e=0.9
p=2*np.pi
N=100
```

```
h=p/150

#defining initial conditions for t=0
r=[1.0+e,0.0]
v=[0.0,np.sqrt((1.0-e)/(1.0+e))]

#defining empy arrays for numerical solution
rarray2=np.array(r)
varray2=np.array(v)

#number of steps to make a n orbits
n=int((N*p)/h)+1

#looping to determine n new points
for i in range(0,n-1):

#using the time step to move on one step
newr=su.modeuler(h,rarray2[-2:],varray2[-2:])[0:2]
newv=su.modeuler(h,rarray2[-2:],varray2[-2:])[2:4]

#addinf the new points to the array
rarray2=np.append(rarray2,newr)
varray2=np.append(varray2,newv)

#defining fractional error in energy for mod euler
Lerror2=np.array([])

for i in range(0,n-1):
Li=rarray2[2*i]*rarray2[2*i+1]-rarray2[2*i+1]*varray2[2*i]

frac=(np.absolute(Li-L0))/np.absolute(L0)

Lerror2=np.append(Lerror2,frac)

#LEAPFROG
#intializing variables
e=0.9
p=2*np.pi
N=100
h=p/150

#defining initial conditions for t=0
r=[1.0+e,0.0]
v=[0.0,np.sqrt((1.0-e)/(1.0+e))]

#defining empy arrays for numerical solution
rarray3=np.array(r)
varray3=np.array(v)

#number of steps to make a n orbits
n=int((N*p)/h)+1
```

```
#looping to determine n new points
for i in range(0,n-1):

#using the time step to move on one step
newr=su.leapfrog(h,rarray3[-2:],varray3[-2:])[0:2]
newv=su.leapfrog(h,rarray3[-2:],varray3[-2:])[2:4]

#addinf the new points to the array
rarray3=np.append(rarray3,newr)
varray3=np.append(varray3,newv)

#defining array of time
tarray2=np.arange(0,(n-1)*h,h)

#defining fractional error in energy for mod euler
Lerror3=np.array([])

for i in range(0,n-1):
Li=rarray3[2*i]*rarray3[2*i+1]-rarray3[2*i+1]*varray3[2*i]

frac=(np.absolute(Li-L0))/np.absolute(L0)

Lerror3=np.append(Lerror3,frac)

#RK4
#intializing variables
e=0.9
p=2*np.pi
N=100
h=p/75

#defining initial conditions for t=0
r=[1.0+e,0.0]
v=[0.0,np.sqrt((1.0-e)/(1.0+e))]

#defining empy arrays for numerical solution
rarray4=np.array(r)
varray4=np.array(v)

#number of steps to make a n orbits
n=int((N*p)/h)+1

#looping to determine n new points
for i in range(0,n-1):

#using the time step to move on one step
newr=su.RK4(h,rarray4[-2:],varray4[-2:])[0:2]
newv=su.RK4(h,rarray4[-2:],varray4[-2:])[2:4]

#addinf the new points to the array
rarray4=np.append(rarray4,newr)
varray4=np.append(varray4,newv)
```

```
#defining array of time
tarray3=np.arange(0,(n-1)*h,h)

#defining fractional error in energy for mod euler
Lerror4=np.array([])

for i in range(0,n-1):
Li=rarray4[2*i]*rarray4[2*i+1]-rarray4[2*i+1]*varray4[2*i]

frac=(np.absolute(Li-L0))/np.absolute(L0)

Lerror4=np.append(Lerror4,frac)

plt.plot(tarray[0:3000:120],Lerror[0:3000:120],'bx',label="Forward␣Euler")
plt.plot(tarray[0:3000:120],Lerror[0:3000:120],'b')
plt.plot(tarray2[0:1500:60],Lerror2[0:1500:60],'rx',label="Modified␣Euler")
plt.plot(tarray2[0:1500:60],Lerror2[0:1500:60],'r')
plt.plot(tarray2[0:1500:60],Lerror3[0:1500:60],'gx',label="Leapfrog")
plt.plot(tarray2[0:1500:60],Lerror3[0:1500:60],'g')
plt.plot(tarray3[0:750:30],Lerror4[0:750:30],'cx',label="RK4")
plt.plot(tarray3[0:750:30],Lerror4[0:750:30],'c')
plt.xlabel("t")
plt.ylabel("fractional␣error␣in␣angular␣momentum")
plt.legend()
plt.show()
```