MATH3001: Computational Applied Maths

Project in Mathematics

Youssef Kawook
201171981

Adrian Barker

10/04/2020

Contents

Abstract

This report tackles three important mathematical problems by using various numerical approximation schemes. Approximating mathematical problems computationally increases efficiency and generates approximations to some problems which would otherwise be impossible to solve. Although when approximating problems numerically errors are often induced. We will derive the schemes used in this report, which will also allow us to find analytical upper limits and estimations of the errors of each numerical scheme. By then analysing errors both analytically and numerically using graphs, comparisons between the schemes may then be made as to which scheme is most suitable under certain conditions.

When developing my code, I was aware of ethical issues around the incredible accuracy of numerical schemes to predict future events and the importance of being respectful and trustworthy. I'm conscious that although my work is only small-scale, when on a large-scale issues such as inaccurately predicting the orbits of a planet could mean loss of life for astronauts or millions of pounds in damages for companies like NASA.

1 How can we accurately evolve planetary orbits using geometric integrators?

1.1 Introduction

In this section, we consider a gravitational two-body problem in two-dimensions. In particular, the *Kepler orbit* in which a planets motion relative to a star of mass M, is only dependent on gravitational force, initial position and velocity, neglecting external forces. The stars mass is much larger than the planets so it is considered to be the center of mass of the system [18].
The acceleration of the planet is dependent on the mass of the star M, the position vector of the planet relative to the star $\boldsymbol{r} = r\hat{\boldsymbol{r}}$ (where $\hat{\boldsymbol{r}}$ is the unit vector and r is the distance between the two bodies) and the Gravitational constant G:

$$\frac{\mathrm{d}^2\mathbf{r}}{\mathrm{d}t^2} = \boldsymbol{F}(\boldsymbol{r}) = -\frac{GM}{r^3}\boldsymbol{r}. \tag{1.1}$$

We will look into the use of four integration schemes for approximating the evolution of the planet. These numerical methods are the Forward Euler (FE), Modified Euler (ME) (also called Heun's or Improved Euler), Leapfrog (LF) and Runge-Kutta (RK4) methods.

The numerical methods above aim to solve the following ODE for given initial values.

$$\dot{\boldsymbol{r}} = \boldsymbol{v},\ \dot{\boldsymbol{v}} = \boldsymbol{F}(\boldsymbol{r}) = -GM\frac{\hat{\boldsymbol{r}}}{r^2}. \tag{1.2}$$

For the purpose of comparison between the methods, we evolve the planets orbits starting at the apocentre on an $x-y$ plane such that the initial conditions are $x = a(1+e)$, $y = 0$, $v_x = 0$, $v_y = \sqrt{\frac{GM(1-e)}{a(1+e)}}$, orbital period $P = 2\pi\sqrt{\frac{a^3}{GM}}$, with semi major axis a and eccentricity e. For simplicity we let $GM = a = 1$.

The analytical prediction provides the shape of the planets orbital path:

$$x = a(\cos E + e), \qquad y = b\sin E, \tag{1.3}$$

where $b = a\sqrt{1-e^2}$ and $E \in [0, 2\pi]$.

We will now investigate each method to determine which are suitable for long-term orbit integrations.

1.2 Numerical integration methods

We will now go through each method giving the definition and their order for time step h. The order of a scheme is generally found by the degree of the *local truncation error* (LTE). LTEs are induced when removing remainder terms in the derivation of the methods using the Taylor series. The LTE of a scheme is the error at each step such that if a scheme has LTE $O(h^{k+1})$, the scheme is of order k [22]. This is because the order of a scheme is given by the *global truncation error* (GTE), which is the error accumulated after time t or $n = t/h$ steps. Thus, the GTE is generally $O(h^{k+1}) \times \frac{1}{h} = O(h^k)$.

Note: each method approximates the new position $\boldsymbol{r}_{n+1}$ and velocity $\boldsymbol{v}_{n+1}$ after each time step.

(i) Forward Euler method:

$$\boldsymbol{r}_{n+1} = \boldsymbol{r}_n + h\boldsymbol{v}_n, \qquad \boldsymbol{v}_{n+1} = \boldsymbol{v}_n + h\boldsymbol{F}(\boldsymbol{r}_n), \tag{1.4}$$

can simply be derived using the Taylor series.

If we let y_0 be the position of the planet in one dimension and y_1 be the new position at time $t_1 = t_0 + h$, then

$$y_1 = y_0 + hf(t_0, y_0). \tag{1.5}$$

Using Taylor series expansion,

$$y(t_0 + h) = y(t_0) + hy'(t_0) + \frac{1}{2}h^2y''(t_0) + O(h^3) \tag{1.6}$$

$$\Rightarrow LTE = y(t_0 + h) - y_1 = \frac{1}{2}h^2y''(t_0) + O(h^3)$$

$$\Rightarrow LTE = O(h^2), \tag{1.7}$$

so the method is of first-order. Thus the LTE is proportional to the square of the time step. The GTE for the FE method is $O(h)$ so the GTE is proportional to the step size [19].

(ii) Modified Euler method:

$$\boldsymbol{r}_{n+1} = \boldsymbol{r}_n + h\boldsymbol{v}_n, \qquad \boldsymbol{v}_{n+1} = \boldsymbol{v}_n + h\boldsymbol{F}(\boldsymbol{r}_{n+1}), \tag{1.8}$$

is also first-order so the GTE is also $O(h)$.

(iii) Leapfrog method:

$$\boldsymbol{r'} = \boldsymbol{r}_n + \frac{h}{2}\boldsymbol{v}_n, \qquad \boldsymbol{v}_{n+1} = \boldsymbol{v}_n + h\boldsymbol{F}(\boldsymbol{r'}), \qquad \boldsymbol{r}_{n+1} = \boldsymbol{r'} + \frac{h}{2}\boldsymbol{v}_{n+1}, \tag{1.9}$$

is of second-order, so the GTE is proportional to the the square of the step size.

(iv) Runge-Kutta method: Let $\boldsymbol{w} = (\boldsymbol{r}, \boldsymbol{v}^T)$, such that $\dot{\boldsymbol{w}} = \boldsymbol{f}(\boldsymbol{w}, t)$ then the scheme is given by

$$\boldsymbol{w}_{n+1} = \boldsymbol{w}_n + \frac{1}{6}(\boldsymbol{k}_1 + 2\boldsymbol{k}_2 + 2\boldsymbol{k}_3 + \boldsymbol{k}_4), \tag{1.10}$$

where

$$\boldsymbol{k}_1 = h\boldsymbol{f}(\boldsymbol{w}_n, t_n), \qquad \boldsymbol{k}_2 = h\boldsymbol{f}(\boldsymbol{w}_n + \frac{1}{2}\boldsymbol{k}_1, t_n + \frac{1}{2}h),$$
$$\boldsymbol{k}_3 = h\boldsymbol{f}(\boldsymbol{w}_n + \frac{1}{2}\boldsymbol{k}_2, t_n + \frac{1}{2}h), \qquad \boldsymbol{k}_3 = h\boldsymbol{f}(\boldsymbol{w}_n + \boldsymbol{k}_3, t_n + h).$$

This method is fourth-order. It is also important to note that the Forward Euler, Modified Euler and Leapfrog integration techniques use a single force evaluation per time increment (h), whereas the Runge-Kutta method applies four force evaluations for a single increment. In order to compare each method, we will ensure that the number of force evaluations is the same for each method.

From analysing the errors above we may hypothesize that the FE and ME schemes would be the least accurate as they are both of the lowest order, then LF to be more accurate and then RK4 scheme to be the most accurate of the four methods.

Figure 1 below shows the planets positions on the $x - y$ plane for a single orbital period ($N = 1$) starting at the apocentre using all four integration techniques ensuring that the number of force evaluations are equal.

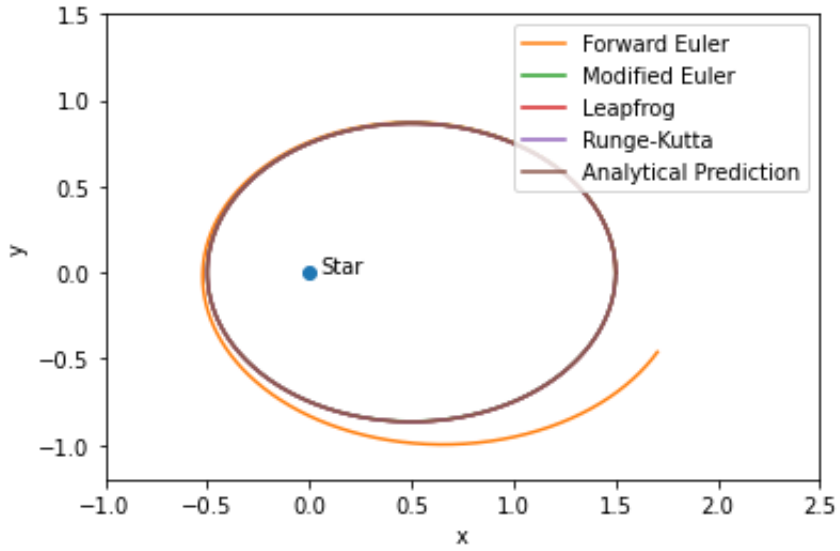

Figure 1: Orbit evaluations over a single orbital period for the four numerical methods over the analytical prediction ($e = 0.5$, $N = 1$, 1000 Force Evaluations)

It is clear from figure 1 that the Forward method is a very poor approximation after only a single orbit when compared to the other three methods and the analytical prediction. This is due to the FE's low order and large local/ global truncation error discussed in the previous section. The ME method performs much better than the FE method which could not be predicted using only the orders of the schemes (as they are both first-order). All the other methods are visually approximately the same after a single orbital period. Although these small errors may become very large over a long period of time. This takes us into the next section where we will look further into long-term integrations.

1.3 Error and Conservation

As a planet orbits a star the energy of the planet is conserved. The kinetic energy of a planet may be transformed into gravitational potential energy but the total of the two won't change.

Proof.

$$\begin{aligned}
\text{Energy} = \varepsilon &= \frac{1}{2}(v_x^2 + v_y^2) - \frac{GM}{r} \\
\Rightarrow \frac{\mathrm{d}\left[\varepsilon\right]}{\mathrm{d}t} = \dot{\varepsilon} &= \frac{\mathrm{d}}{\mathrm{d}t}\left[\frac{1}{2}(v_x^2 + v_y^2) - \frac{GM}{r}\right] \\
&= \frac{\mathrm{d}}{\mathrm{d}t}\left[\frac{1}{2}(v_x^2 + v_y^2)\right] - \frac{d}{dt}\left[\frac{GM}{r}\right] \\
&= \frac{1}{2}\left(2v_x\dot{v}_x + 2v_y\dot{v}_y\right) - GM\frac{\mathrm{d}}{\mathrm{d}t}\left[\frac{1}{\sqrt{x^2+y^2}}\right] \\
&= v_x\dot{v}_x + v_y\dot{v}_y + \frac{GM\left(2x\dot{x} + 2y\dot{y}\right)}{(x^2+y^2)^{\frac{3}{2}}} \\
&= v_x\dot{v}_x + v_y\dot{v}_y + \frac{GMx\dot{x}}{r^3} + \frac{GMy\dot{y}}{r^3} \\
&= v_x\dot{v}_x + v_y\dot{v}_y - v_x\dot{v}_x - v_y\dot{v}_y \\
&= 0
\end{aligned}$$

□

The angular momentum of the planet is also conserved.

Proof.

$$\begin{aligned}
\text{Angular Momentum} = L &= xv_y - yv_x \\
\Rightarrow \frac{\mathrm{d}[L]}{\mathrm{d}t} = \dot{L} &= \frac{\mathrm{d}}{\mathrm{d}t}\left[xv_y - yv_x\right] \\
&= x\dot{v}_y + \dot{x}v_y - (y\dot{v}_x + \dot{y}v_x) \\
&= x\dot{v}_y - y\dot{v}_x \\
&= x\left(-\frac{GMy}{r^3}\right) - y\left(-\frac{GMx}{r^3}\right) \\
&= -\frac{xyGM}{r^3} + \frac{xyGM}{r^3} \\
&= 0
\end{aligned}$$

□

Consequently, as we evolve the planetary orbits numerically, the energy and angular momentum should remain constant throughout. Hence in figures 3 and 6, and figures 4 and 7 showing fractional energy error $|\varepsilon - \varepsilon_0|/\varepsilon_0$ and fractional angular momentum error $|L - L_0|/L_0$ respectively, should show no deviation from zero.

Note: by dividing the energy/ momentum error by the initial energy/ momentum we produce the fractional error allowing us to see a proportional change.

Simple x-y plots show the path of the plant but they miss other details such as the velocity at which the planet is moving at for each scheme. Fractional energy error and angular momentum error are useful tools to compare methods as they also give a good indication of accuracy for variables such as velocity (speed and direction) at a given time.

Figure 2 shows that as discussed previously the FE method is very poor after only a single orbital period and progressively gets worse over time. It also shows that the ME, LF and RK4 methods appear to be fairly good approximations over longer time periods (100 orbital periods i.e N=100), with a larger deviation from the analytical prediction for the ME and LF methods compared to the RK4 method.
Note: in both figure 2 and 5 the approximations for the path of the planet using the ME and LF methods are almost matching and therefore overlap. Consequently, only the path of the LF method is visible.

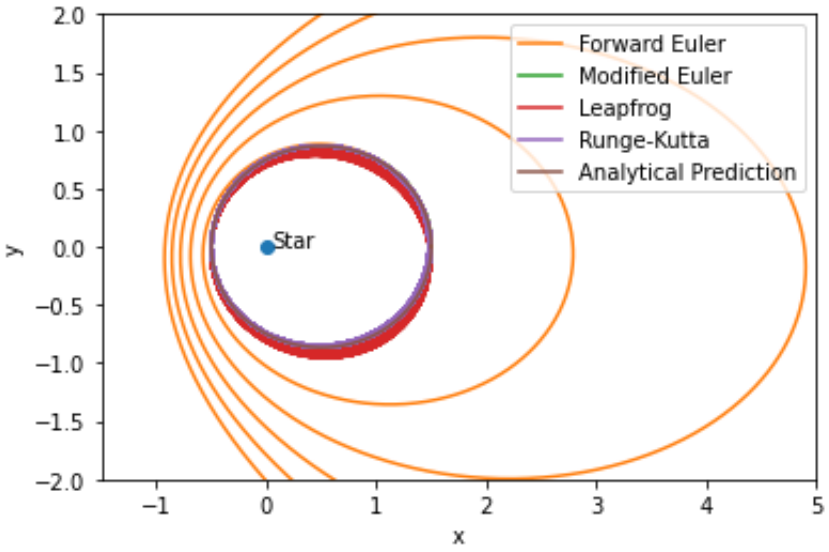

Figure 2: Orbit evaluations for 100 orbital periods for the four numerical methods over the Analytical prediction ($e = 0.5$, $N = 100$, 300 Force Evaluations)

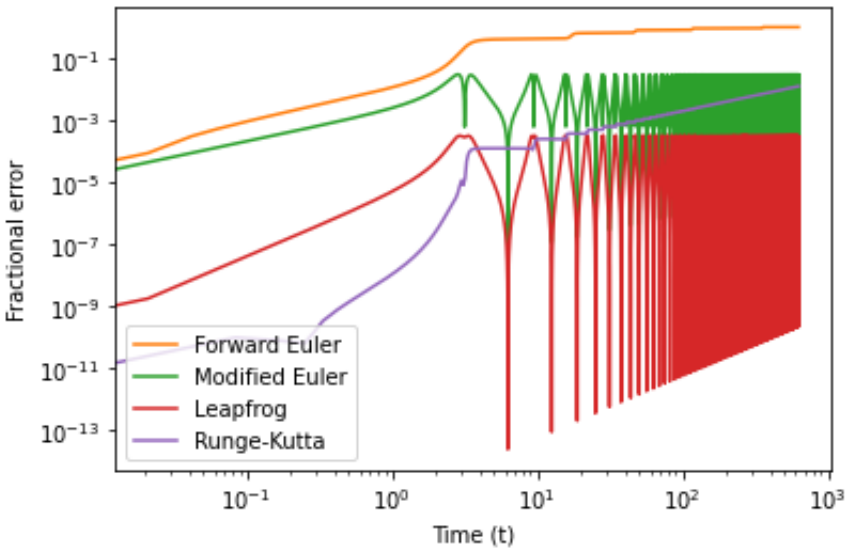

Figure 3: Fractional energy error for the four numerical methods ($e = 0.5$, $N = 100$, 300 Force Evaluations)

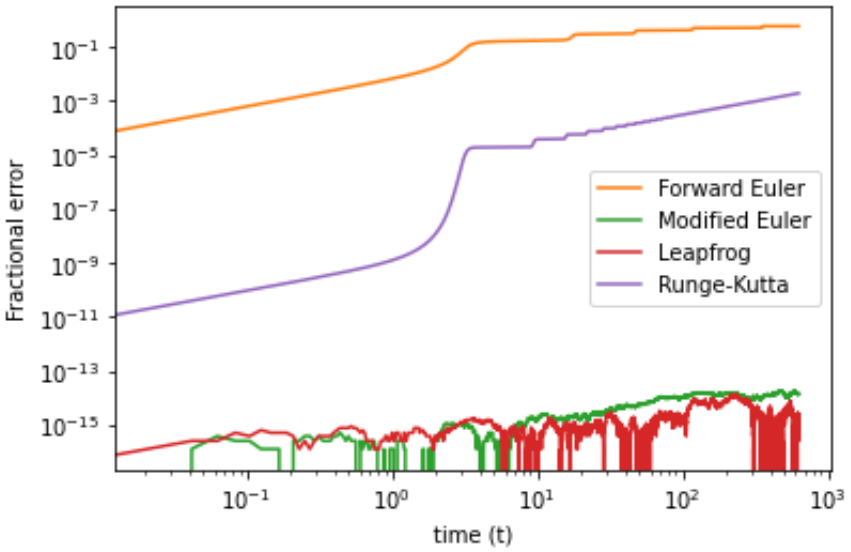

Figure 4: Fractional momentum error for the four numerical methods ($e = 0.5$, $N = 100$, 300 Force Evaluations)

We can see from figures 3 and 6 that the ME and LF methods fractional energy error oscillates and is bounded. The ME and LF methods follow a similar pattern but the LF methods fractional energy error is always the smallest of the two techniques with the LF methods fractional energy error being the best after a single orbital period.

It is visible from figures 2 and 5 that the plots for ME and LF appear to take a similar path. Also, their paths appear to be contained whereas both the FE and RK4 methods errors become very large in figure 5.

The maximum fractional errors don't increase over long periods of time and their paths are contained due to the ME and LF methods being geometric integration algorithms. This is beneficial in the long-term compared to the other techniques which appear to continuously increase. Geometric integrators are discussed in further detail in the Hamiltonian systems section.

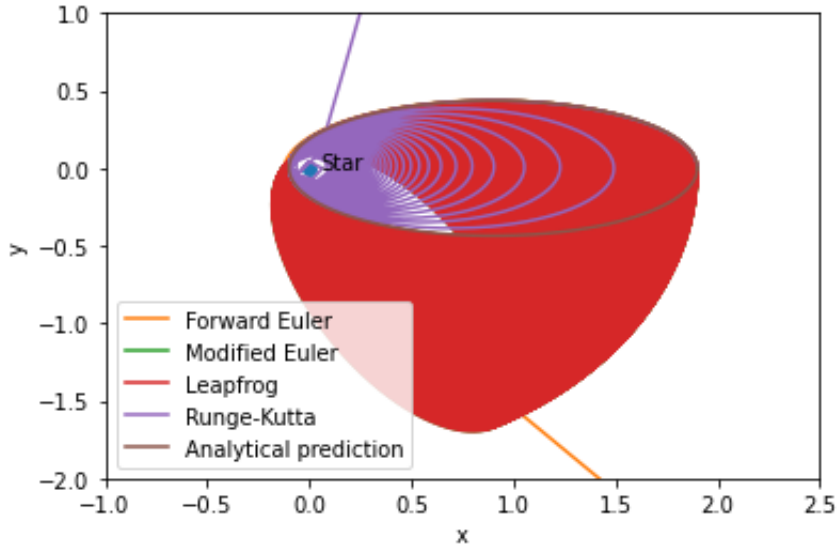

Figure 5: Orbit evaluations for 100 orbital periods for the four numerical methods over the Analytical prediction ($e = 0.9$, $N = 100$, 300 Force Evaluations)

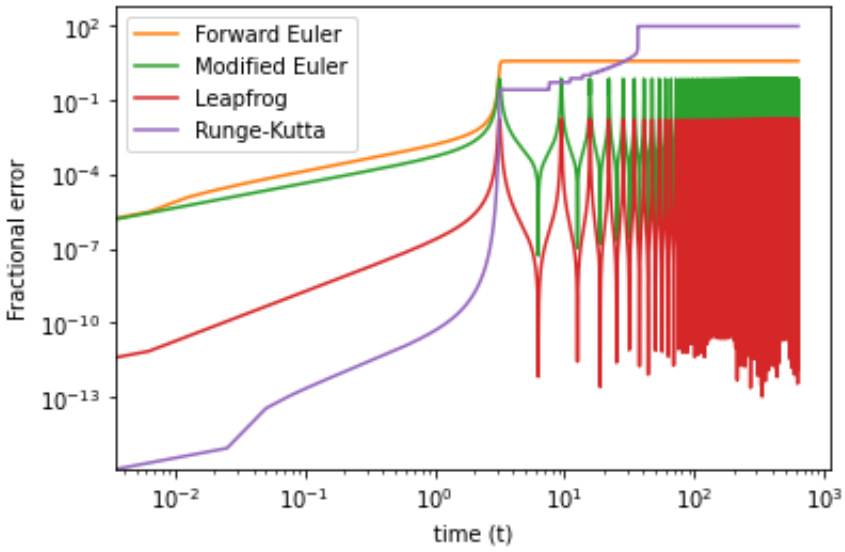

Figure 6: Fractional energy error for the four numerical methods ($e = 0.9$, $N = 100$, 300 Force Evaluations)

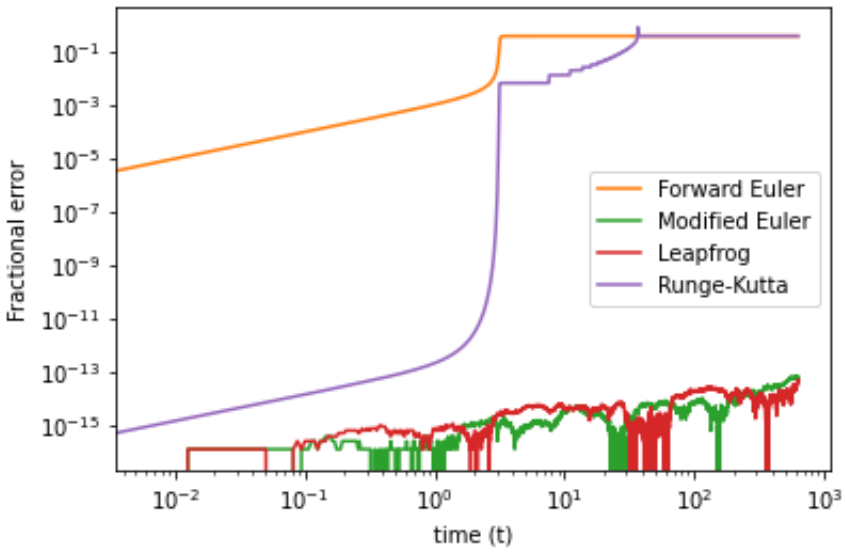

Figure 7: Fractional momentum error for the four numerical methods ($e = 0.9$, $N = 100$, 1000 Force Evaluations)

In figure 5 we can see the Forward Euler method becomes almost a straight line after going round the pericentre. Therefore, at the same time ($t = \pi$) the fractional energy error and the fractional momentum error become constant.

Also, in figure 5, just before $t = \pi$ (where the change in velocity and direction becomes great) as the RK4 method evolves it tends towards the star meaning the fractional energy and momentum error also increases rapidly just before $t = \pi$ (shown in figures 6 and 7). The planet then continues to get closer to the star until the velocity of the planet becomes very large. At this time the fractional energy error and fractional momentum error become constant in figures 6 and 7 respectively.

The key analysis from these 6 figures is that although the RK4 method has the largest order, since the ME and LF methods are geometric integrators they are more stable and therefore perform better in the long-term.

1.4 Reversibility

Good integration techniques retain key features of the system as they evolve. A beneficial feature for an integrator is reversibility, meaning that at any point the velocities can be reversed and we can "follow our footsteps" back to where we were a given amount of time ago. So, if you integrated until time 143h (143 steps) then you will be able to find your initial position and velocity by reversing the velocity and integrating for another 143 steps then reversing the velocity again to retain the correct direction.

Table 1 below gives the errors in position and velocity when each technique integrated a planet with initial position x_0 and velocity v_0, forward in time for 10 orbital periods and then backwards for 10 orbital periods. The final velocity was then reversed once again to calculate v_f at the final position x_f. Note for the methods to be reversible the absolute difference between the starting and finishing velocities and positions should be zero (allowing for small rounding error $< 10^{-10}$).

Technique	$\lvert x_0 - x_f \rvert$	$\lvert y_0 - y_f \rvert$	$\lvert v_{x_0} - v_{x_f} \rvert$	$\lvert v_{y_0} - v_{y_f} \rvert$
Forward Euler	14.342733	3.06667	0.182766	0.530407
Modified Euler	0.000140	0.004654	0.003582	0.000065
Leapfrog	3.996803×10^{-14}	4.643777×10^{-13}	3.593514×10^{-13}	1.310063×10^{-14}
Runge-Kutta	0.007958	0.067010	0.051826	0.000528

Table 1: Absolute differences between initial and final positions and velocities after being integrated forwards and then reversed as described ($e = 0.5$, $N = 10$, 300 force evaluations)

The reversibility of the integrators are ranked below:

1. Leapfrog
2. Modified Euler
3. Runge-Kutta
4. Forward Euler

The leapfrog integration method is the only method which is time reversible (allowing for small rounding error $< 10^{-10}$) and hence doesn't have numerical dissipation. Numerical dissipation is the error which occurs when differentials are calculated numerically. This is due to the structure of the LF method shown in figure 8 below.

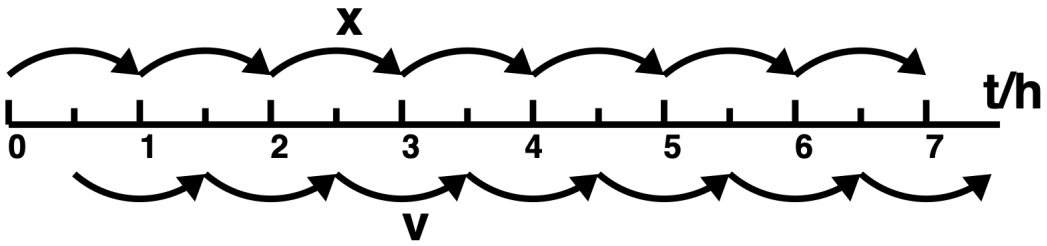

Figure 8: Sketch of the leapfrog method showing how a given velocity is used to generate both x_n and x_{n+1} [21]

x on the top represents the position evaluation at each point for each n, the points in between each integer represent the $\boldsymbol{r}'_n$ points where the velocity is re-evaluated. The sketch is like this as the velocity updates in between points $\boldsymbol{r}'_n$ and then an average of the old velocity $\boldsymbol{v}_n$ and new velocity $\boldsymbol{v}_{n+1}$ is used to calculate the new position such that $\boldsymbol{r}_{n+1} = \boldsymbol{r}_n + h/2(\boldsymbol{v}_n + \boldsymbol{v}_{n+1})$.

In order to prove the method is reversible, lets begin at $n = 0$ such that $\boldsymbol{r}_0$ and $\boldsymbol{v}_0$ is our starting position and velocity respectfully. Then

$$\boldsymbol{v}_1 = \boldsymbol{v}_0 + h\boldsymbol{F}(\boldsymbol{r}'_0), \qquad \boldsymbol{r}_1 = \boldsymbol{r}'_0 + \frac{h}{2}\boldsymbol{v}_1. \tag{1.11}$$

If we reverse the velocity such that $\boldsymbol{v}_1$ becomes $-\boldsymbol{v}_1$ and evaluate from $\boldsymbol{r}_1$ then,

$$\boldsymbol{v}_2 = -\boldsymbol{v}_1 + h\boldsymbol{F}(\boldsymbol{r}'_1), \qquad \boldsymbol{r}_2 = \boldsymbol{r}'_1 + \frac{h}{2}\boldsymbol{v}_2. \tag{1.12}$$

The velocity $\boldsymbol{v}_2$ can re-writen as

$$\boldsymbol{v}_2 = -\boldsymbol{v}_0 - h\boldsymbol{F}(\boldsymbol{r}_0') + h\boldsymbol{F}(\boldsymbol{r}_1'). \tag{1.13}$$

Now given

$$\boldsymbol{r}_0' = -\boldsymbol{r}_0 + \frac{h}{2}\boldsymbol{v}_0, \qquad \boldsymbol{r}_1' = \boldsymbol{r}_1 - \frac{h}{2}\boldsymbol{v}_1, \tag{1.14}$$

since $\boldsymbol{r}_1 = \boldsymbol{r}_0' + \frac{h}{2}\boldsymbol{v}_1$,

$$\begin{aligned} \boldsymbol{r}_1' &= \boldsymbol{r}_0' + \frac{h}{2}\boldsymbol{v}_1 - \frac{h}{2}\boldsymbol{v}_1 \\ \Rightarrow \boldsymbol{r}_1' &= \boldsymbol{r}_0'. \end{aligned} \tag{1.15}$$

So because $\boldsymbol{r}_0' = \boldsymbol{r}_1'$, the acceleration terms in (1.13) are evaluated at the same point and are therefore equal, we can write

$$\begin{aligned} \boldsymbol{v}_2 &= -\boldsymbol{v}_0 - h\boldsymbol{F}(\boldsymbol{r}_0') + h\boldsymbol{F}(\boldsymbol{r}_0') \\ \Rightarrow \boldsymbol{v}_2 &= -\boldsymbol{v}_0. \end{aligned} \tag{1.16}$$

Thus

$$\begin{aligned} \boldsymbol{r}_2 &= \boldsymbol{r}_1' + \frac{h}{2}\boldsymbol{v}_2 \\ \Rightarrow \boldsymbol{r}_2 &= \boldsymbol{r}_0' - \frac{h}{2}\boldsymbol{v}_0 \\ \Rightarrow \boldsymbol{r}_2 &= \boldsymbol{r}_0 + \frac{h}{2}\boldsymbol{v}_0 - \frac{h}{2}\boldsymbol{v}_1 \\ \Rightarrow \boldsymbol{r}_2 &= \boldsymbol{r}_0. \end{aligned} \tag{1.17}$$

Hence $\boldsymbol{v}_2 = -\boldsymbol{v}_0$ and $\boldsymbol{r}_2 = \boldsymbol{r}_0$ so the scheme is reversible.

If a scheme is reversible as it ensures energy and angular momentum are conserved over time. This apparent in figures 3, 4, 6 and 7 where the LFs methods error is bounded.

1.5 Hamilton Systems

A Hamiltonian, in a closed system, is a function which represents the sum of kinetic and potential energy. Equation (1.1) can be derived from a Hamiltonian therefore Liouville's theorem applies. Liouvillie's theorem states that the volume of phase-space remains constant. Our 4-dimensional phase-space (x, y, v_x, v_y) is therefore area conserving. So if we let dV be the phase space at time t and dV' be the phase space at time t', then dV'=dV. Therefore,

$$\int\int\int\int \mathrm{d}x'\,\mathrm{d}y'\,\mathrm{d}v_x'\,\mathrm{d}v_y' = \int\int\int\int |J|\,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}v_x\,\mathrm{d}v_y. \tag{1.18}$$

Thus $|J| = 1$.

By calculating the determinant of the Jacobian for a numerical method we can find if the phase space is conserved and is therefore called a *symplectic* method.

Jacobian determinant for the Forward Euler method:

$$|J| = \begin{vmatrix} 1 & 0 & h & 0 \\ 0 & 1 & 0 & h \\ \frac{h(3x_n^2-(x_n^2+y_n^2))}{(x_n^2+y_n^2)^{\frac{5}{2}}} & \frac{3x_n y_n}{(x_n^2+y_n^2)^{\frac{5}{2}}} & 1 & 0 \\ \frac{3x_n y_n}{(x_n^2+y_n^2)^{\frac{5}{2}}} & \frac{h(3y_n^2-(x_n^2+y_n^2))}{(x_n^2+y_n^2)^{\frac{5}{2}}} & 0 & 1 \end{vmatrix}$$

$$= 1 - \frac{h^2(3y_n^2 - r_n^2)}{r_n^5} - \frac{h^2(3x_n^2 - r_n^2)}{r_n^5} + \frac{h^4(3x_n^2 - r_n^2)(3y_n^2 - r_n^2)}{r_n^{10}} - \frac{9h^2x_n^2y_n^2}{r_n^{10}} \neq 1, \tag{1.19}$$

where $r_n = \sqrt{x_n^2 + y_n^2}$ is the distance from the x-y center (0,0). As the determinant is not equal to one the phase space is not conserved over time so the FE method isn't symplectic.

Jacobian determinant for the Modified Euler method:

$$|J| = \begin{vmatrix} 1 & 0 & h & 0 \\ 0 & 1 & 0 & h \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} = 1. \tag{1.20}$$

The determinant is equal to one as the phase-space volume doesn't change with respect to time so the ME method is symplectic.

It can be found that the Jacobian determinant for the Leapfrog method is also equal to one so it is also symplectic whereas the Runge-Kutta methods Jacobian determinant is not equal to one and is therefore not phase-space volume conserving.

Since the numerical Modified Euler and Leapfrog methods are symplectic, they preserve the geometric property of the exact equation, in that they preserve the flow of the system. The flow is the mapping of the system over time. Consequently, they are geometric integrators whereas the Forward Euler and Runge-Kutta methods arnt. This is beneficial in the long-term as it increases the stability of the integrator greatly. Meaning that over a long period of time the error doesn't grow (GTE stays low). Whereas methods such as the RK4 method aim to have small LTE but the GTE grows continuously over time [21].

1.6 Conclusion

In this section, we proved that the energy and angular momentum of a planet orbiting a star in a closed system is conserved and so the energy and angular momentum of the numerical scheme should also be conserved. The LF and ME methods conserved both energy and angular momentum best as they are geometric integrators (phase-space preserving). The LF method was the only reversible scheme and performed better than the other three methods in all of the tests for long-term accuracy.

To conclude, we found that if a scheme has the greatest single step accuracy/ largest order of convergence, it does not necessarily mean the best long-term accuracy. Other features, such as a scheme maintaining phase-space can mean that over long periods of time more of the initial behaviours of the scheme are preserved and are therefore more accurate in the long-term.

2 Solving the heat equation

2.1 Introduction

The heat equation gives us the relationship between the temperature change with respect to time and the temperatures rate of change across one-dimensional space (in our case, a solid metal bar),

$$\frac{\partial \theta}{\partial t} = K\frac{\partial^2 \theta}{\partial x^2}, \quad 0 \leq x \leq L, \quad t \geq 0. \tag{2.1}$$

We let L be the length of the bar and $\theta(x,t)$ to be the average temperature of the bar at a cross-section at distance x along the bar (to make the problem one dimensional). K is a positive constant coefficient, the thermal diffusivity of the bar.

In this chapter, we will analyse different numerical methods for approximating temperature change over time t by utilizing the heat equation with the temperature of the ends of the bar held at a fixed temperature. Since the heat equation (2.1) only involves the change of temperature, the temperature of these ends can be taken to be zero without loss of generality. Thus, we let $\theta(0,t) = \theta(L,t) = 0$.

The finite difference method one of the methods we will investigate. We will look into how the error scales and the stability of this scheme with different variables using Von-Neumann stability analysis. We will then explore an alternative method called the Crank-Nicolson scheme to try to resolve the instability of the finite difference method.

2.2 Derivation

Let the initial temperature distribution along the bar be given by $\theta(x,0) = f(x)$ then we have,

$$\frac{\partial \theta}{\partial t} = K\frac{\partial^2 \theta}{\partial x^2}, \qquad \theta(x,0) = f(x), \qquad \theta(0,t) = \theta(L,t) = 0. \tag{2.2}$$

Separation of variables is then employed by assuming a solution of the form,

$$\theta(x,t) = \varphi(x)G(t). \tag{2.3}$$

Subbing (2.3) into the initial equation (2.1),

$$\frac{\partial}{\partial t}[\varphi(x)G(t)] = K\frac{\partial^2}{\partial x^2}[\varphi(x)G(t)] \tag{2.4}$$

$$\Rightarrow \varphi(x)\frac{\mathrm{d}G}{\mathrm{d}t} = KG(t)\frac{\mathrm{d}^2}{\mathrm{d}x^2}[\varphi(x)]. \tag{2.5}$$

Then separating variables and noticing that for both sides of (2.5) to be equal for any given x and t, they must be constant and equal, so

$$\frac{1}{KG}\frac{\mathrm{d}G}{\mathrm{d}t} = \frac{1}{\varphi}\frac{\mathrm{d}^2\varphi}{\mathrm{d}x^2} = -\lambda \qquad \text{(sepatration constant)}, \tag{2.6}$$

therefore

$$\frac{\mathrm{d}G}{\mathrm{d}t} = -K\lambda G \qquad \text{and} \qquad \frac{\mathrm{d}^2\varphi}{\mathrm{d}x^2} = -\lambda\varphi. \tag{2.7}$$

Now we consider our boundary conditions $\theta(0,t)=0$ and $\theta(L,t)=0$, such that

$$\theta(0,t)=\varphi(0)G(t)=0 \qquad \text{and} \qquad \theta(L,t)=\varphi(L)G(t)=0. \tag{2.8}$$

Thus for non-trivial results $\varphi(0)=0$ and $\varphi(L)=0$ because if $G(t)=0$ then $\theta(x,t)=0$ for all t.

Now solving the spatial equation of (2.7), with the boundary conditions $\varphi(0)=0$ and $\varphi(L)=0$. Forming the boundary value problem,

$$\frac{\mathrm{d}^2\varphi}{\mathrm{d}x^2}+\lambda\varphi=0, \qquad \varphi(0)=0, \qquad \varphi(L)=0. \tag{2.9}$$

Case $\lambda>0$:
Characteristic polynomial,

$$r^2+\lambda=0 \tag{2.10}$$

$$\Rightarrow r_{1,2}=\pm\sqrt{-\lambda} \tag{2.11}$$

$$=\pm i\sqrt{\lambda} \tag{2.12}$$

with general solution, $\varphi(x)=c_1\cos(\sqrt{\lambda}x)+c_2\sin(\sqrt{\lambda}x)$.

Applying boundary conditions,

$$\varphi(0)=0 \;\Rightarrow\; \varphi(0)=c_1=0,$$

and

$$\varphi(L)=0 \;\Rightarrow\; \varphi(L)=c_2\sin(\sqrt{\lambda}L)=0$$

so

$$\Rightarrow c_2=0 \qquad \text{or} \qquad \sin(\sqrt{\lambda}L)=0,$$

for non-trivial results we take $\sin(\sqrt{\lambda}L)=0$, so $L\sqrt{\lambda}=n\pi$ for $n=1,2,3,...$

Note that it can be shown that there are only trivial results for $\lambda<0$ and $\lambda=0$.

Thus all eigenvalues and eigenfunctions are,

$$\lambda_n=(\frac{n\pi}{L})^2, \qquad \varphi_n(x)=\sin(\frac{n\pi x}{L}) \qquad n=1,2,3,... \tag{2.13}$$

We now solve the first-order time differential equation of (2.7) ,

$$\frac{\mathrm{d}G}{\mathrm{d}t}=-K\lambda_n G, \tag{2.14}$$

by again utilizing the separation of variables method,

$$\begin{aligned}
\frac{\mathrm{d}G}{G}&=-K\lambda_n\,\mathrm{d}t,\\
\Rightarrow \int\frac{1}{G}\,\mathrm{d}G&=\int -K\lambda_n\,\mathrm{d}t\\
\Rightarrow ln(G)+C&=-K\lambda_n t+D\\
\Rightarrow ln(G)&=-K\lambda_n t+a \quad \text{(where a = D − C)}\\
\Rightarrow G&=e^{-K\lambda_n t+a}\\
\Rightarrow G&=ce^{-K\lambda_n t},
\end{aligned} \tag{2.15}$$

where $c = e^a$ is a constant.

Combining our two solutions (2.13) and (2.13), and then replacing c by A_n to show that c will be different for each n, we get

$$\theta_n(x,t) = A_n \sin(\frac{n\pi x}{L}) e^{-k(\frac{n\pi}{L})^2 t} \qquad \text{n=1,2,3,...} \tag{2.16}$$

Due to *superposition* the combination of solutions (2.16) to the partial differential equation (2.1) is also a solution to equation (2.1).

Thus

$$\theta(x,t) = \sum_{n=1}^{\infty} A_n \sin(\frac{n\pi x}{L}) e^{-k(\frac{n\pi}{L})^2 t}, \tag{2.17}$$

satisfying the initial temperature distribution at $t = 0$, if

$$\theta(x,0) = f(x) = \sum_{n=1}^{\infty} A_n \sin(\frac{n\pi x}{L}), \tag{2.18}$$

which is equal to the *Fourier sine series.*

Now solving for A_n by firstly multiplying both sides by $\sin(\frac{m\pi x}{L})$ and integrating with respect to x within the interval $[-L, L]$,

$$\begin{aligned} \int_{-L}^{L} f(x)\sin(\frac{m\pi x}{L})\,\mathrm{d}x &= \int_{-L}^{L} \sum_{n=1}^{\infty} A_n \sin(\frac{n\pi x}{L})\sin(\frac{m\pi x}{L})\,\mathrm{d}x \\ &= \sum_{n=1}^{\infty} A_n \int_{-L}^{L} \sin(\frac{n\pi x}{L})\sin(\frac{m\pi x}{L})\,\mathrm{d}x. \end{aligned}$$

It can be found that the integral on the RHS equals L if $n = m$ and 0 if $n \neq m$ so it's *orthogonal*, therefore

$$\begin{aligned} \int_{-L}^{L} f(x)\sin(\frac{m\pi x}{L})\,\mathrm{d}x &= A_m L \\ \Rightarrow A_m = \frac{1}{L}\int_{-L}^{L} f(x)\sin(\frac{m\pi x}{L})\,\mathrm{d}x &\qquad \text{m = 1,2,3,...} \end{aligned} \tag{2.19}$$

Since the integral is *even*

$$A_m = \frac{2}{L}\int_{0}^{L} f(x)\sin(\frac{m\pi x}{L})\,\mathrm{d}x \qquad \text{m = 1,2,3,...} \tag{2.20}$$

Hence, subbing in n for m we have

$$\theta(x,t) = \sum_{n=1}^{\infty} A_n \sin(\frac{n\pi x}{L}) e^{-k(\frac{n\pi}{L})^2 t}, \tag{2.21}$$

with

$$A_n = \frac{2}{L}\int_0^L f(x)\sin(\frac{n\pi x}{L})\,dx \qquad \text{n = 1,2,3,...} \tag{2.22}$$

In the case where $f(x) = \sin(\frac{\pi x}{L})$, (2.22) equals 1 with its only term coming from $n = 1$ due its orthogonality (i.e $A_1 = 1$). Thus, (2.21) gives

$$\theta(x,t) = \sin(\frac{\pi x}{L})e^{-K(\frac{\pi}{L})^2 t}, \tag{2.23}$$

such that the temperature distribution stays the same but the temperature 'flattens' over time [5]. We will numerically approximate the results of this simple case in the next sections in order to analyse the various methods.

2.3 Finite-difference method

We approximate this problem numerically when $\theta(x,0) = f(x) = \sin(\pi x)$ as mentioned previously such that the analytical solution is given by equation (2.23). Fixing $K = 1$, $L = 1$ and calculating until $t_{max} = 1$, we use a finite-difference method to calculate the dependant variable θ in the domain $x \in [0,1]$ at N points thus the distance in between each point is given by $\delta x = 1/(N-1)$.

First-order central difference scheme for approximating the space derivative, derived by considering the Taylor expansion of $\theta(t)$ about t_i:

$$\theta(t_i + \delta t) = \theta(t_i) + \delta t\frac{\partial\theta}{\partial t}\Big|_{t_i} + \frac{\delta t^2}{2}\frac{\partial^2\theta}{\partial t^2}\Big|_{t_i} + ... \tag{2.24}$$

Then solving for $\frac{\partial\theta}{\partial t}$ we get,

$$\frac{\partial\theta}{\partial t}\Big|_{t_i} = \frac{\theta(t_i+\delta t) - \theta(t_i)}{\theta t} - \frac{\delta t^2}{2}\frac{\partial^2\theta}{\partial t^2}\Big|_{t_i} + ... \tag{2.25}$$

Thus

$$\frac{\partial\theta(x,t)}{\partial t} = \frac{\theta(x,t+\delta t) - \theta(x,t)}{\theta t} + O(\delta t), \tag{2.26}$$

where $\frac{\delta t^2}{2}\frac{\partial^2\theta}{\partial t^2}$ is the truncation error of the approximation so we write $O(\delta t)$ using big O notation.

Second-order central difference scheme for approximating the space derivative, derived by summing the Taylor expansion of $\theta(x)$ about x_i in the positive and negative direction:

$$\theta(x_i+\delta x) + \theta(x_i - \delta x) = \theta_{i+1} + \theta_{i-1} = 2\theta_i + (\delta x)^2\frac{\partial^2\theta}{\partial x^2}\Big|_{x_i} + \frac{2(\delta x)^4}{4!}\frac{\partial^4\theta}{\partial x^4}\Big|_{x_i} + ... \tag{2.27}$$

Then solving for $\frac{\partial^2\theta}{\partial x^2}$ we get,

$$\frac{\partial^2\theta}{\partial x^2}\Big|_{x_i} = \frac{\theta_{i+1} + 2\theta_i + \theta_{i-1}}{\delta x^2} + \frac{(\delta x)^2}{12}\frac{\partial^4\theta}{\partial x^4}\Big|_{x_i} + ... \tag{2.28}$$

Therefore,

$$\frac{\partial^2\theta(x,t)}{\partial x^2} = \frac{\theta(x+\delta x,t) - 2\theta(x,t) + \theta(x-\delta x,t)}{(\delta x)^2} + O((\delta x)^2), \tag{2.29}$$

written in big O notation, where $\frac{(\delta x)^2}{12}\frac{\partial^4\theta}{\partial x^4}$ is the truncation error.

Then is we sub in our approximations for the time and space derivatives (2.26) and (2.29), into the Heat equation (2.1),

$$\frac{\theta(x,t+\delta t) - \theta(x,t)}{\delta t} + O(\delta t) = K\frac{\theta(x+\delta x,t) - 2\theta(x,t) + \theta(x-\delta x,t)}{(\delta x)^2} + O((\delta x)^2) \tag{2.30}$$

$$\Rightarrow \theta(x,t+\delta t) = \theta(x,t) + \frac{K\delta t}{(\delta x)^2}[\theta(x+\delta x,t) - 2\theta(x,t) + \theta(x-\delta x,t)] + O(\delta t) + O((\delta x)^2). \tag{2.31}$$

Hence the numerical scheme *Forward Time, Centered Space* (FTCS):

$$\theta_i^{m+1} = \theta_i^m + \frac{K\delta t}{(\delta x)^2}\left[\theta_{i+1}^m - 2\theta_i^m + \theta_{i-1}^m\right], \tag{2.32}$$

where θ_i^m approximates $\theta(i\delta x, m\delta t)$ and the initial temperatures θ_i^0 are calculated analytically. We define the *Courant number*, $C = \frac{K\delta t}{(\delta x)^2}$. The scheme is explicit as the solution at time $(m+1)$ only depends on the solution at time m [14].

Below are some plots to show the accuracy of the approximation over time, figures 9a and 9b show that the FTCS scheme provides a good approximation as it follows the correct concave shape at both times $t = 0.25$ in figure 9a and $t = 0.75$ in figure 9b. In both plots, we can see that the numerical scheme decays quicker than the analytical solution, with the numerical scheme being almost a third smaller than the analytical solution at $t = 0.75$ although the magnitude is small.

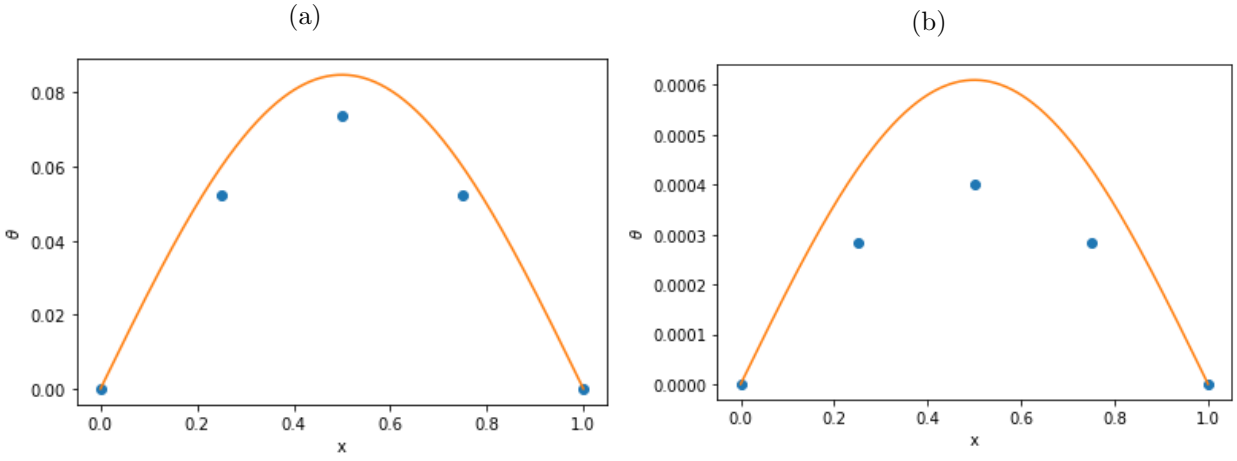

Figure 9: Orange line is the analytical prediction to the heat equation. Blue dots plot the FTCS numerical approximation to the heat equation ($N = 5$, $C = 1/3$). (a) $t = 0.25$. (b) $t = 0.75$.

From the table 2 you can see that $\theta(x, 0.25)$ the same positions $x = 0.25$ and $x = 0.75$ this is due to the symmetrical form of the initial condition $f(x) = \sin(x)$ and the numerical scheme. Also the error

in the centre is greater than the neighbouring points in magnitude and proportion. This is not only due to the greater magnitude of θ at $x = 0.5$ but also because from the numerical scheme we know that both $x = 0.25$ and $x = 0.75$ have a neighbouring point at one of the boundary points meaning that instead of using three approximations from the previous time step ($x = 0.25,\ 0.5,\ 0.75$), they are only using two ($x = 0.25,\ 0.5$ or $x = 0.5,\ 0.75$).

x	$\theta(x, 0.25)$	$\lvert\theta(x, 0.25) - \theta_{an}(x, 0.25)\rvert$	$\theta(x, 0.75)$	$\lvert\theta(x, 0.75) - \theta_{an}(x, 0.75)\rvert$
0	0	0	0	0
0.25	0.052160070	0.007806101	0.000283821	0.000147449
0.5	0.073765479	0.011039493	0.000401383	0.000208524
0.75	0.052160070	0.007806101	0.000283821	0.000147449
1	0	1.038561381 e-17	0	7.469212309 e-20

Table 2: Numerical values of the FTCS scheme and absolute error of the solutions at the two times and five nodes, where θ_{an} is the analytical solution.

Note: error at $x = 1$ should be 0 as the boundary conditions are $\theta(0, t) = \theta(L, T) = 0$ but there is a small error in the python code which means $\sin\pi \neq 0$.

To analyse the accuracy of the scheme across the whole of the bar we can use the *root-mean-square error* (RMSE), given by $E = \sqrt{\int_0^1 |\theta(x,t) - \theta_{an}(x,t)|^2 \mathrm{d}x}$, where θ_{an} is the analytical solution (2.23).

Equation 2.31 used to derive the FTCS scheme has a combination of two errors, temporal $O(\delta t)$ and spatial $O((\delta x)^2)$. Thus figure 10 below shows two behaviours in the RMSE with respect to the Courant number C. As we increase the Courant number ($C = \frac{K\delta t}{(\delta x)^2}$), since K and δx is fixed δt must increase so the total number of evaluations/ time steps decreases. The effects on the errors are

- spatial error: the local truncation error $\frac{(\delta x)^2}{12}\frac{\partial^4\theta}{\partial x^4}$ doesn't change with respect to δt but since there are fewer steps there are fewer local spatial errors summed together so the global truncation error decreases. This is the behaviour shown initially in figure 11a as C increases to 0.2.
- temporal error: the local truncation error $\frac{\delta t^2}{2}\frac{\partial^2\theta}{\partial t^2}$ increases as δt increases. Thus the global truncation error also increases. This increase in error becomes dominant over the decrease in spatial error as δt increases after a certain value of C. This behaviour is shown in figure 11a after $C = 0.2$.

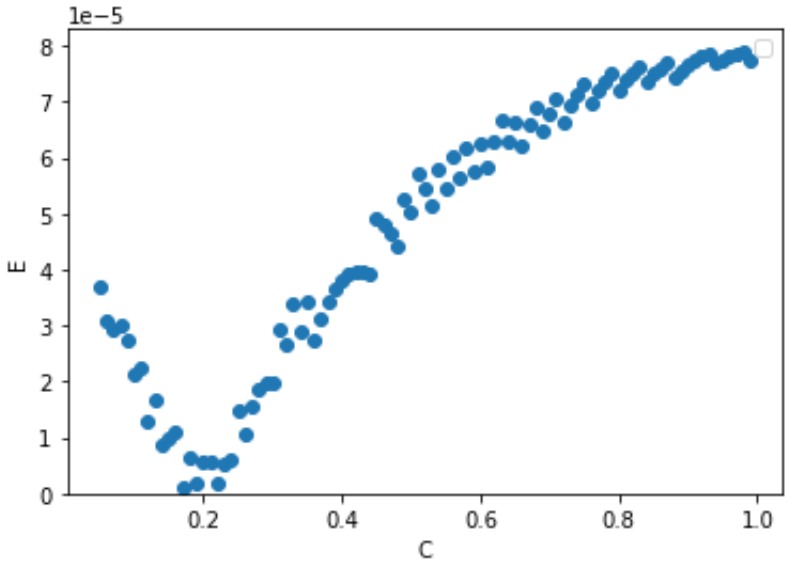

Figure 10: Shows the relationship between the RMSE (E) and the Courant number C ($N = 5$, $t = 1$)

As the number of points N increases $\delta x = 1/(N-1)$ decreases. This decrease in δx decreases the local truncation error of the spatial term $\frac{(\delta x)^2}{12}\frac{\partial^4 \theta}{\partial x^4}$ also decreases. Since this error is equivalent to $O((\delta x)^2$ the error converges at a rate of $1/N^2$. This relationship can be seen in figure 11b below.

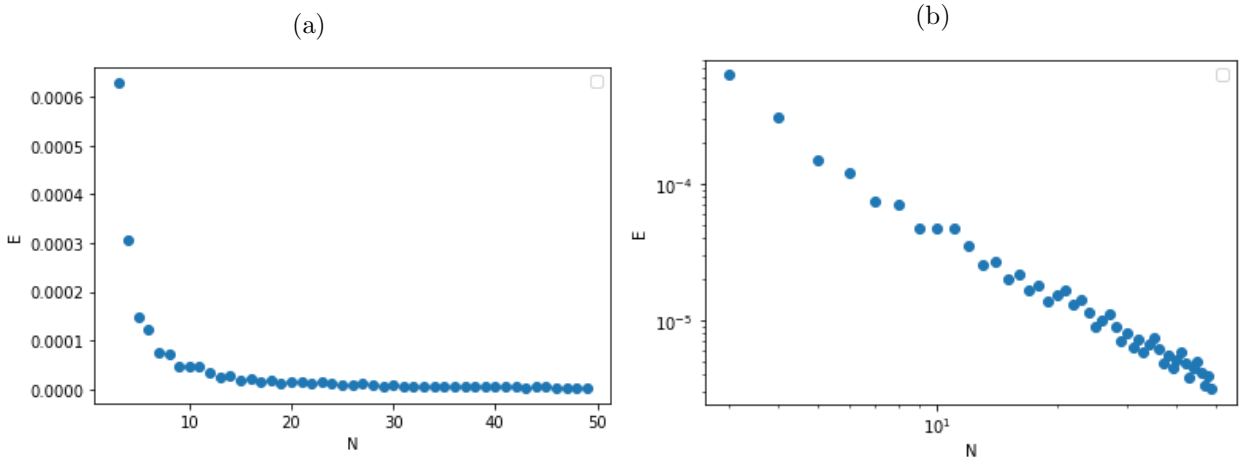

Figure 11: Shows the relationship between the RMSE (E) and the number of points N ($N = 5$, $C = 0.1$, $t = 0.75$). (a) Plot on linear axis. (b) Plot on logarithmic axis.

2.4 Von-Neumann stability analysis

If a numerical scheme is unstable, the error at time t causes magnified errors at future times. This causes errors to grow incredibly large after a given period of time.

Figure 12 below shows how the FTCS schemes unstable behaviour appears after time $t = 0.49$

when $C = \frac{2}{3}$. The FTCS solution begins to oscillate and the magnitude of the solution grows exponentially over time. The magnitude of the FTCS approximations at some of the nodes are greater than 10^7 at time $t = 1$, which is of course incredibly inaccurate to the analytical solution.

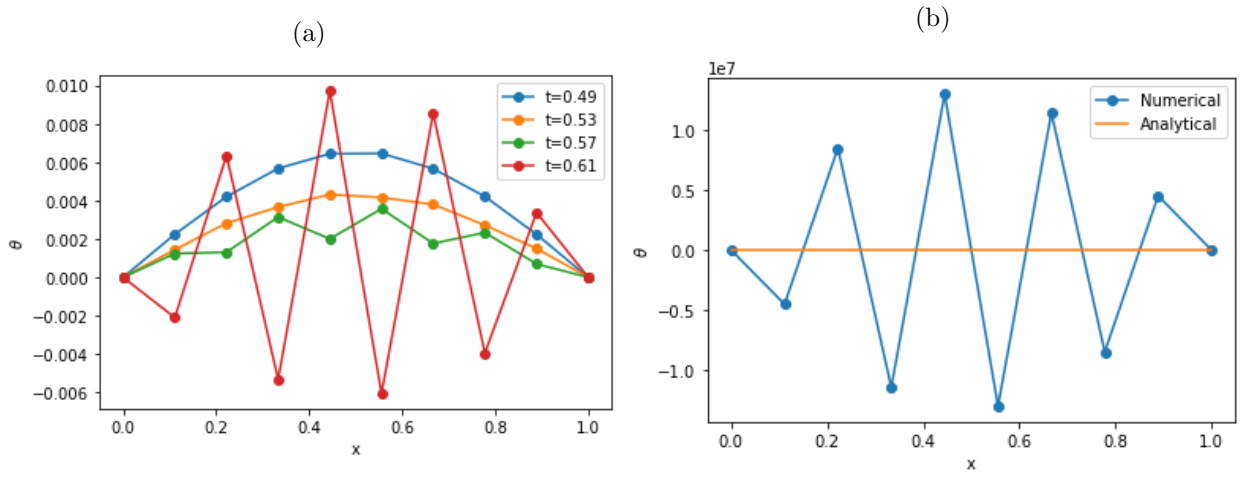

Figure 12: Shows instability of the FTCS schemes approximation of the heat equation with particular parameters ($N = 10$, $C = 2/3$) (a) FTCS approximation for times $t = 0.49, 0.53, 0.57, 0.61$ (b) FTCS schemes approximation and analytical prediction at $t = 1$ with θ plotted on a 10^7 scale.

Von-Neumann stability analysis is a technique used to check the stability of finite difference schemes. We sub

$$\theta(j\delta x, m\delta t) = \xi^m e^{ikj\delta x}, \tag{2.33}$$

into the FTCS scheme in order to test the stability, where k is a real spatial wavenumber and $\xi(k)$ is the amplitude, dependent on k. The absolute amplitude must be less than one because if greater than one the temperature θ would grow exponentially since ξ is raised to the power of m, shown in figure 12.

Thus subbing (2.33) into (2.32),

$$\xi^{m+1}e^{ikj\delta x} = \xi^m e^{ikj\delta x} + \frac{K\delta t}{(\delta x)^2}[\xi^m e^{ik(j+21)\delta x} - 2\xi^m e^{ikj\delta x} + \xi^m e^{ik(j-1)\delta x}], \tag{2.34}$$

then dividing by $\xi^m e^{ikj\delta x}$,

$$\xi = 1 + C[e^{ik\delta x} - 2 + e^{-ik\delta x}].$$

Given

$$\sin^2 x = \frac{e^{2ix} - 2 + e^{-2ix}}{-4}, \tag{2.35}$$

we have

$$\xi = 1 + C[-4\sin^2(\frac{1}{2}k\delta x)]$$
$$\Rightarrow |\xi| = |(1 - 4C\sin^2(\frac{1}{2}k\delta x))|. \tag{2.36}$$

Then, for the scheme to be stable $|\xi| \leq 1$, so

$$\begin{aligned} -1 \leq 1 - 4C\sin^2(\frac{1}{2}k\delta x) &\leq 1 \\ \Rightarrow -1 \leq 1 - 4C &\leq 1 \\ \Rightarrow -2 \leq -4C &\leq 0 \\ \Rightarrow 0 \leq C &\leq \frac{1}{2}. \end{aligned} \tag{2.37}$$

This explains why the FTCS scheme was unstable in figure 2.33 because for that example $C = 2/3 \nleq 1/2$ so according to the Von Neumann stability analysis the scheme was unstable.

2.5 Crank-Nicolson scheme

An alternative numerical method is the Crank-Nickolson scheme. The scheme is implicit as the solution at time $(m+1)$ depends on both the solution at time m and the solution at time $m+1$, therefore a matrix is required to calculate the solution.

The scheme uses the following to calculate the solution,

$$\frac{\theta_i^{m+1} - \theta_i^m}{\delta t} = \frac{K}{2}\left[\frac{\theta_{i+1}^{m+1} + 2\theta_i^{m+1} + \theta_{i-1}^{m+1}}{\delta x^2} + \frac{\theta_{i+1}^m + 2\theta_i^m + \theta_{i-1}^m}{\delta x^2}\right], \tag{2.38}$$

where θ_i^m approximates $\theta(i\delta x, m\delta t)$ as described previously.

It is also important to note that the Crank-Nicolson scheme is second-order convergent with respect to time as the schemes local truncation error is given by $O(\delta t^2) + O(\delta x^2)$. Therefore the temporal error converges quicker than that of the FTCS scheme which has only first-order convergence due to its temporal truncation error $O(\delta t)$.

Rearranging we get,

$$-\frac{C}{2}\theta_{i-1}^{m+1} + (1+C)\theta_i^{m+1} - \frac{C}{2}\theta_{i+1}^{m+1} = \frac{C}{2}\theta_{i-1}^m + (1-C)\theta_i^m + \frac{C}{2}\theta_{i+1}^m, \tag{2.39}$$

so for N=5 nodes, we have

$$A = \begin{pmatrix} 1+C & -\frac{C}{2} & 0 \\ -\frac{C}{2} & 1+C & -\frac{C}{2} \\ 0 & -\frac{C}{2} & 1+C \end{pmatrix} \tag{2.40}$$

and

$$\boldsymbol{b}(\boldsymbol{\theta}^m) = \begin{pmatrix} \frac{1}{2}C\theta_0^m + (1-C)\theta_1^m + \frac{1}{2}C\theta_2^m \\ \frac{1}{2}C\theta_1^m + (1-C)\theta_2^m + \frac{1}{2}C\theta_3^m \\ \frac{1}{2}C\theta_2^m + (1-C)\theta_3^m + \frac{1}{2}C\theta_4^m \end{pmatrix}, \tag{2.41}$$

such that $A\boldsymbol{\theta}^{m+1} = \boldsymbol{b}(\boldsymbol{\theta}^m)$. Our matrix only calculates interior nodes as the exterior nodes equal zero ($\theta_0 = \theta_4 = 0$). Therefore A is only a 3×3 matrix instead of a 5×5 matrix. This scheme can be expanded for any number of points [14].

Figure 13 below uses the Crank-Nicholson scheme to approximate the heat equation with the same parameters as used in the previous section for figure 12.

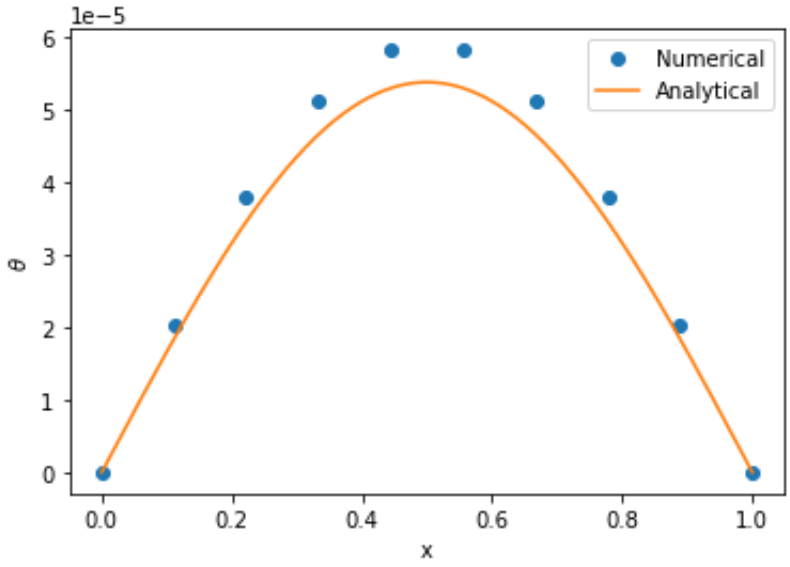

Figure 13: Crank-Nicolson schemes approximation of the heat equation showing stability ($N = 10$, $C = 2/3$, $t = 10$)

Figure 13 shows that the Crank-Nicholson scheme is accurate and stable for $C = 2/3$ and in turn a much better approximation when compared to the FTCS scheme.

2.6 Conclusion

To conclude, we found the finite difference method FTCS, is an explicit scheme formed by combining second-order spatial and first-order temporal difference schemes causing it to also have a combination of spatial $O((\delta x)^2)$ and temporal $O((\delta t))$ errors.

Using the Von-Neumann stability analysis on the FTCS scheme we found that if the Courant number is less than or equal to a half ($C \leq 1/2$) then the system is stable but if the Courant number is greater than a half ($C > 1/2$) then the scheme becomes unstable so the approximations error magnifies and oscillates over time. Alternatively, the implicit finite difference method, Crank-Nicolson scheme, is unconditionally stable and the temporal error scales at a quicker rate than the FTCS scheme. Hence the Crank-Nicolson scheme is the superior method of the two.

3 Using random numbers to integrate

3.1 Introduction

In mathematics examples of integrals are usually analytically solvable but in most real-world cases integrals are only numerically solvable. To approximate integrals numerically, we find the sum of function evaluations for a number of samples n,

$$\int_a^b f(x)\,\mathrm{d}x \approx \sum_{i=1}^{N} w_i f(x_i). \tag{3.1}$$

This technique is commonly used in methods such as the Trapezoidal and Simpsons rule in one dimension. Where n sample points are spaced equally across the range $[a, b]$ including the two extremities. These methods are very effective in one dimension but are almost impossible to solve in multiple dimensions. This is because the computational expense grows exponentially with the number of dimensions. Say we are calculating an integral in D dimensions and in each dimension we have n points then we must sum $N = n^D$ function evaluations, this is known as the *curse of dimensionality* [9].

In this chapter, we will look into solving integrals numerically in small and large dimensions and compare the actual and analytical errors of the numerical integration techniques used.

3.2 Monte Carlo integration

A numerical scheme for calculating integrals is the Monte Carlo method. This method uses random numbers in order to obtain a sample. It is incredibly difficult to generate a truly random sample using a computer, as computers are only able to complete predictable, repetitive and deterministic tasks. Consequently, numbers generated by a random number generator are called *pseudo-random* meaning they are actually part of a sequence generated by an algorithm. In most statistical cases using pseudo-random numbers generated by a standard random number generator passes the users empirical testing and is therefore deemed sufficient [17].

The Monte Carlo integration method approximates the integral of a function, say f, over a region W, of which could form a complicated shape. It approximates the integral by defining a corresponding discrete function, say g, within a sample region V such that $W \in V$. The discrete function equals f at N random points if within the region W and equals 0 outside of the region W ($V \backslash W$),

$$g(x_i) = \begin{cases} f(x_i) & x_i \in W \\ 0 & x_i \in V \backslash W. \end{cases} \tag{3.2}$$

Finally, the method then finds the mean of the function at the N points and multiplies this by the volume of V in order to calculate the approximation for the integral,

$$\int f\,dV \approx V\,\langle f \rangle \pm V\sqrt{\frac{\langle f^2 \rangle - \langle f \rangle^2}{N}}, \tag{3.3}$$

with the angle brackets denoting the mean over the N points such that

$$\langle f \rangle \equiv \frac{1}{N}\sum_{i=1}^{N} g(x_i), \qquad \langle f^2 \rangle \equiv \frac{1}{N}\sum_{i=1}^{N} g^2(x_i), \tag{3.4}$$

where $x_0, ..., x_N$ are random points, uniformly distributed across the sample space V [13].

Note: the error estimate is given by the last, plus/minus term, of the approximation equation (3.3). The error is equal to the volume of the sample space V, multiplied by the standard deviation, divided by the square root of the number of point N.

Thus, the sampling region V should be as small as possible whilst enclosing all of the integrating region W in order to reduce error. This is because reducing the sampling area, reduces the number of points where g equals zero, which increases the proportion of points which improve the accuracy of the mean (lying within W).

3.3 One-dimensional integration

The Monte Carlo integration approximation in one-dimension is given by,

$$\int_a^b f(x)\,\mathrm{d}x \approx \frac{(b-a)}{N}\sum_{i=1}^{N} f(x_i). \tag{3.5}$$

The error is given by,

$$(b-a)\frac{\sigma}{\sqrt{N}}. \tag{3.6}$$

In order to compare the effectiveness of the Monte-Carlo method in one dimension we will also use the trapezoidal Rule on a uniform grid,

$$\int_a^b f(x)\,\mathrm{d}x \approx \frac{b-a}{2N}\sum_{k=1}^{N}(f(x_{k-1}) + f(x_k)), \tag{3.7}$$

with error,

$$\frac{(b-a)^3}{12N^2}f''(\xi), \tag{3.8}$$

where $a < \xi < b$ [2].

Thus the maximum absolute error can be given by,

$$|\text{error}| \leq \frac{(b-a)^3}{12N^2}\text{Max}_{[a,b]}f''(x). \tag{3.9}$$

Now using both methods we will approximate the following integral,

$$\frac{1}{2}\int_0^2 \cos^2(x)\mathrm{d}x. \tag{3.10}$$

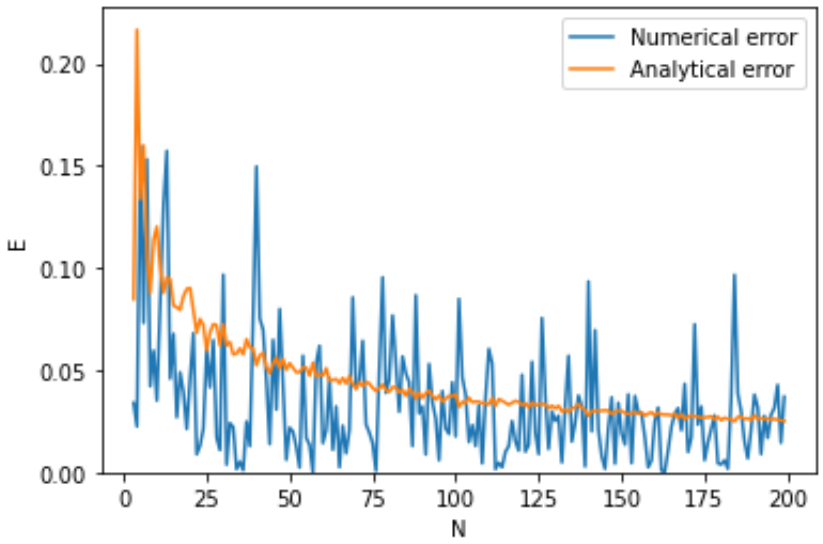

Figure 14: Absolute error incurred when approximating (3.10) using the Monte Carlo for different N. Analytical error calculated using (3.6) .

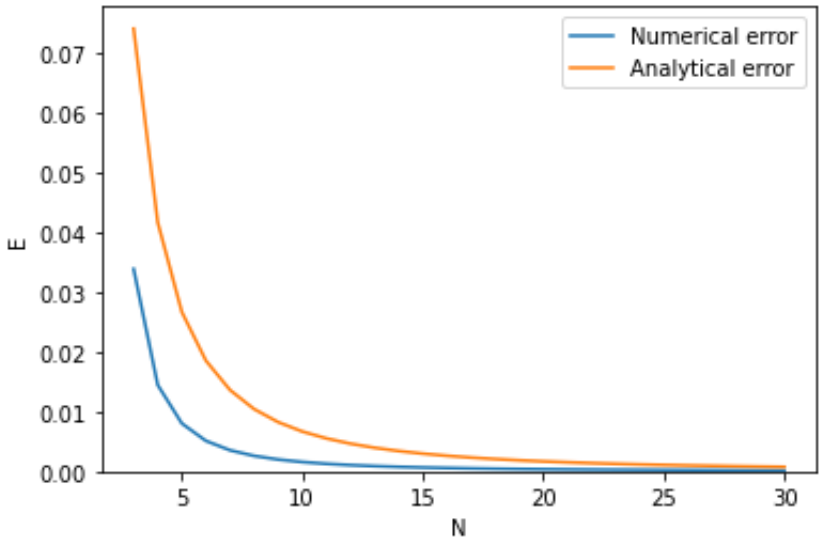

Figure 15: Absolute error incurred when approximating (3.10) using the Trapezoidal rule for different N. Analytical error calculated using (3.9).

Figure 14 and 15 show that the error inured from the Monte-Carlo method in one-dimension is much greater than that of the trapezoidal rule. We can also see that the rate in which the error decreases for the Monte-Carlo method ($N^{-\frac{1}{2}}$) is less than that of the Trapezoidal rule (N^{-2}) [3]. Its also important to note the volatility of the Monte-Carlo method in figure 17 this is due to the randomness of the sample points of each run for the different N values. Whereas the Trapezoidal rules points are in almost the same position for $N+1$ as they are for N as they are spaced equally. This means the approximations for each N are similar thus the line is smooth.

The Trapezoidal rule is better for one-dimensional integration as it links adjacent points a and

b. Since the function is continuous the trapezoidal scheme approximates the path of the function between the points by interpolating with a straight line connecting $f(a)$ and $f(b)$. This means that after only a small number of points the approximation of the functions is close to the exact curve. However, the Monte Carlo method doesn't link adjacent points therefore less data is used for the approximation. Hence the Monte Carlo method is not favourable over the Trapezoidal method in one-dimension.

3.4 Quadrature schemes in multiple dimensions

As we have seen in the last chapter the Monte Carlo method isn't favourable in one-dimension but it's when we are integrating in larger dimensions

$$\int_V f(\boldsymbol{x})\,\mathrm{d}V = \int_{a_D}^{b_D} ... \int_{a_1}^{b_1} f(x_1, ..., x_D)\,\mathrm{d}x_1 ... \mathrm{d}x_D, \tag{3.11}$$

where Monte Carlo integration becomes advantageous. This is due to the exponential growth of function evaluations required when increasing one-dimensional quadrature schemes (Trapezoidal rule) by applying Fubini's theorem.

This is because as discussed in the introduction for each point in one dimension we must evaluate each of the points at n new points in the other dimension and so on (assuming there are n points in each integral).

If we let the partitions $P_1, ..., P_D$ be sets containing $n_1, ..., n_D$ points in consecutive dimensions respectively which are equally spaced within the integration intervals $[a_1, b_1], ..., [a_D, b_D]$, such that $P_i = \{x_i^{[k_1]}, ..., x_i^{[k_{n_i}]}\}$ with $a_i = x_i^{[k_1]} < ... < x_i^{[k_j]} < ... < x_i^{[k_{n_i}]} = b_i$. Then our sampling set is $P = P_1 \times ... \times P_D$ over the rectangular sampling region $\Omega = [a_1, b_1] \times ... \times [a_D, b_D]$. Figure 16 displays this below for two-dimensions.

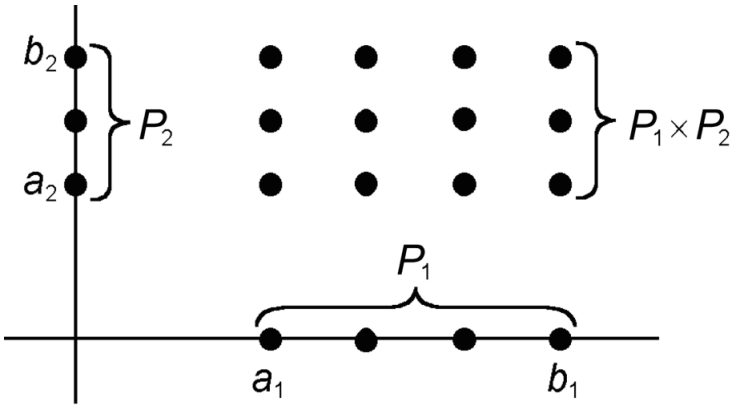

Figure 16: Example sample set $P = P_1 \times P_2$ for two-dimensions within the region $\Omega = [a_1, b_1] \times [a_2, b_2]$ with $n_1 = 4$ and $n_2 = 3$. Thus there are $4 \times 3 = 12$ points within the set P [9].

A quadrature scheme uses a quadrature rule function $w(P)$ in order to assign a set of weights to

the partition P such that their sum equals the area of the integrating region Ω and

$$\lim_{min(n_i)\to+\infty} \sum_{k_D=0}^{n_D} ... \sum_{k_1=0}^{n_1} w^{[k_1,...,k_D]} f(\boldsymbol{x}^{[k_1,...,k_D]}) = \int_{a_D}^{b_D} ... \int_{a_1}^{b_1} f(x_1, ..., x_D)\, \mathrm{d}x_1 ... \mathrm{d}x_D, \tag{3.12}$$

where $\boldsymbol{x}^{[k_1,...,k_D]}$ is the position vector with coordinates $(x^{k_1}, ..., x^{k_D})$ and $w^{[k_1,...,k_D]}$ is the weight for the point with the same position vector.

One-dimensional quadrature rules $w_i(P_i)$ such as the trapezoidal rule can then be expanded to multiple dimensions using the product rule

$$w^{[k_1,...,k_D]} = w^{[k_1]} \times ... \times w^{[k_D]}. \tag{3.13}$$

Then the sum of function output for each point multiplied by the weights using (3.1) provides the approximation for the integral [9].

Note: the error of the Monte Carlo method remains proportional to $N^{\frac{1}{2}}$ irrespective of the dimension.

Let the number of points for each integral in each dimension be n. Then, if a quadrature scheme is used and the rectangular sampling region Ω:

- fits perfectly within the boundaries of the domain of the function, then the error in each dimension will be proportional to n^{-2} using the trapezoidal or Simpson's rule. Now given the total number of points equals $N = n^D$, then $n = N^{-\frac{1}{D}}$ so the error is proportional to $N^{-\frac{2}{D}}$ this is the *curse of dimensionality* mentioned in the introduction of this chapter. Thus in four-dimensional space the errors of both the quadrature and Monte Carlo method will scale at the same rate $N^{-\frac{1}{2}}$. When $D > 4$ the Monte Carlo methods error will converge at a quicker rate.
- doesn't exclusively cover the domain of the function then parts of the function within Ω will be discontinuous therefore higher-order convergence schemes such as the Trapezoidal and Simpson's rules assumptions won't be satisfied (as the function isn't continuously differentiable) so instead the error will scale as n^{-1} using Left and Right Riemann sums approximation [11]. Consequently, the error is proportional to $N^{-\frac{1}{D}}$ so when $D > 2$ the Monte Carlo methods error will converge at a quicker rate.

3.5 Monte Carlo method in multiple dimensions

For an example we will approximate the area of a unit circle by integrating:

$$f = \begin{cases} 1 & x_1^2 + x_2^2 < 1 \\ 0 & \text{otherwise} \end{cases}, \tag{3.14}$$

over the domain $V = \{(x_1, x_2) : |x_1| < 1, |x_2| < 1\}$. The analytical solution is π.

Figure 17 below shows the error of the Monte Carlo method is proportional to $1/\sqrt{N}$ as it is clear that both the analytical error approximation and the absolute numerical error follow the curve of $E = 1/\sqrt{N}$.

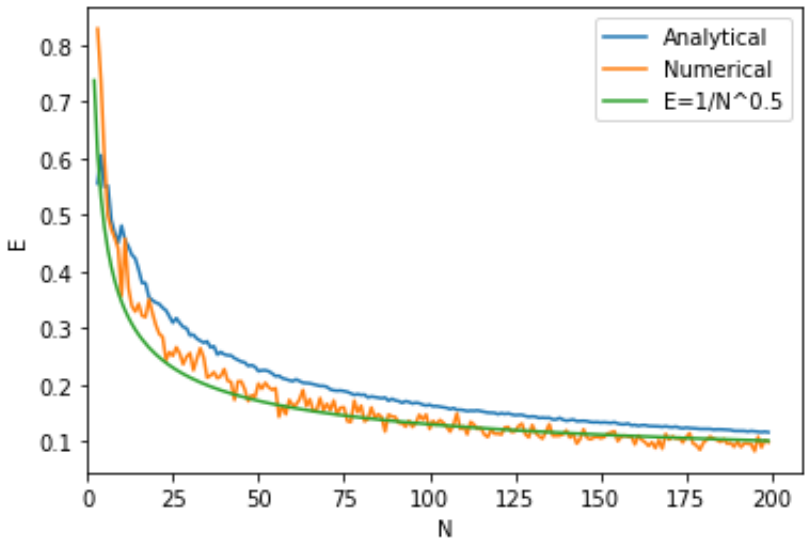

Figure 17: Shows analytical error and average numerical error scales as $N^{-\frac{1}{2}}$ when approximating (3.14) using the Monte Carlo method (3.3).

This example can be extended to integrate in larger dimensions:

Dimension	Exact Volume	Monte Carlo approximation	Absolute Error
2	$\pi \approx 3.141593$	3.1472	0.005607
3	$(4/3)\pi \approx 4.188790$	4.2368	0.048010
4	$(1/2)\pi^2 \approx 4.934802$	5.0560	0.121198
5	$(8/15)\pi^2 \approx 5.263789$	5.2288	0.034989
6	$(1/6)\pi^3 \approx 5.167713$	5.5040	0.336287
7	$(16/105)\pi^3 \approx 4.724766$	4.3264	0.398366

Table 3: Monte Carlo methods approximation to (3.14) and the absolute error from the exact volume [6] ($N = 10000$).

Table 3 shows that the Monte Carlo method accurately approximates the volume of higher dimensional hyperspheres as the error converges as $1/\sqrt{N}$, irrespective of the dimension.

3.6 Conclusion

To conclude, quadrature schemes are most effective in small dimensions ($D < 3$), providing high accuracy and quicker convergence especially in one-dimension but progressively become worse especially if the integration region is discontinuous. This is because quadrature schemes are effected by the curse of dimensionality where error scales at a rate of $N^{-\frac{1}{D}}$ when the integration region is discontinuous or $N^{-\frac{2}{D}}$ when the region fits perfectly in the domain. On the other hand, the Monte Carlo integration method is better in large dimensions as the error scales at the same rate ($N^{-\frac{1}{2}}$) irrespective of the dimension. Also, function evaluations grow exponentially ($N = n^D$) for quadrature schemes, making it almost impossible to calculate integrals of large dimensions such as a 50-dimensional integral. Hence when evaluating an integral, parameters such as these must be taken into account when deciding which scheme will be most suitable/ effective.

References

[1] Ames, W.F. (1977). *Numerical methods for partial differential equations.* 2nd ed. Editorial: New York, N.Y.: Academic Press, New York.

[2] Anon, (2020). *Numerical Integration - Midpoint, Trapezoid, Simpson's rule.* [online] Available at: `https://chem.libretexts.org/@go/page/10269` [Accessed 16 Mar. 2021].

[3] Atkinson, K.E. (2004). *An introduction to numerical analysis.* New York: John Wiley and Sons.

[4] Coleman, M.P. (2013). *An introduction to partial differential equations with MATLAB.* Boca Raton: Chapman And Hall/Crc.

[5] Dawkins, P. (2019). *Differential Equations - Solving the Heat Equation.* [online] tutorial.math.lamar.edu. Available at: `https://tutorial.math.lamar.edu/classes/de/solvingheatequation.aspx` [Accessed 16 Mar. 2021].

[6] Dewdney, A.K. (n.d.). *N-Dimensions.* [online] Keith's Think Zone. Available at: `https://thinkzone.wlonk.com/MathFun/Dimens.htm.`

[7] Dunn, W.L. and J Kenneth Shultis (2012). *Exploring Monte Carlo methods.* Amsterdam ; Boston: Elsevier/Academic Press.

[8] Goldstein, H., Poole, C. and Safko, J. (2002). *Classical mechanics.* 3rd ed. San Francisco: Addison Wesley.

[9] Holton, G.A. (2010). *Value-at-risk : theory and practice.* 2nd ed. Boca Raton, Fla.: Chapman & Hall/Crc.

[10] Kalos, M.H. and Whitlock, P.A. (1986). *Monte Carlo methods.* Vol. 1 Basics. New York Wiley.

[11] McClure, M. (2017). *Riemann Error.* [online] marksmath.org. Available at: `https://marksmath.org/visualization/RiemannError/` [Accessed 25 Mar. 2021].

[12] Murray, C.D. and Dermott, S.F. (1999). *Solar system dynamics.* Cambridge [U. A.] Cambridge Univ. Press.

[13] Press, W.H. and Al, E. (2007). *Numerical recipes : the art of scientific computing.* Cambridge ; New York: Cambridge University Press.

[14] Recktenwald, G. (2011). *Finite-Difference Approximations to the Heat Equation.* [online] . Available at: `http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=A73B07B5FEDF017DC554CEF16B015E9F?doi=10.1.1.408.4054&rep=rep1&type=pdf` [Accessed 16 Mar. 2021].

[15] Smith, G.D. (1985). *Numerical solution of partial differential equations : finite difference methods.* 3rd ed. Oxford: Oxford University Press, Cop.

[16] Vallieres, M. (2008). Time-Reversability. [online] www.physics.drexel.edu. Available at: `http://www.physics.drexel.edu/~valliere/PHYS305/Diff_Eq_Integrators/time_reversal/#:~:text=Time` [Accessed 21 Mar. 2021].

[17] Vallieres, M. (2014). *Generating Random Variables.* [online] `www.physics.drexel.edu.` Available at: `http://www.physics.drexel.edu/~valliere/PHYS305/Monte_Carlo/Monte_Carlo_story/node1.html` [Accessed 22 Mar. 2021].

[18] Wikipedia. (2019). Kepler orbit. [online] Available at: `https://en.wikipedia.org/wiki/Kepler_orbit` [Accessed 27 Nov. 2019].

[19] Wikipedia. (2020). *Euler method.* [online] Available at: `https://en.wikipedia.org/wiki/Euler_method` [Accessed 16 Mar. 2021].

[20] Wikipedia. (2020b). *Von Neumann stability analysis.* [online] Available at: `https://en.wikipedia.org/wiki/Von_Neumann_stability_analysis` [Accessed 28 Mar. 2021].

[21] Young, P. (2013). *The leapfrog method and other "symplectic" algorithms for integrating Newton's laws of motion.* [online] . Available at: `http://physics.ucsc.edu/~peter/242/leapfrog.pdf` [Accessed 21 Mar. 2021].

[22] Zeltkevic, M. (1998). *Forward and Backward Euler Methods.* [online] web.mit.edu. Available at: `https://web.mit.edu/10.001/Web/Course_Notes/Differential_Equations_Notes/node3.html` [Accessed 16 Mar. 2021].

A Evolving planetary orbits appendix

Forward Euler:

```
import matplotlib.pyplot as plt
import math
import numpy as np
import pickle
#FE

GM=1
e=0.5

#define parameters
P=2*math.pi
tstart=0
tend=P
h=P/300


#set initial conditions
x0=(1+e); y0=0; vx0=0; vy0=((1-e)/(1+e))**0.5;
E0=0.5*(vx0**2+vy0**2)-GM/((x0**2+y0**2)**0.5); L0=x0*vy0-y0*vx0;
t=tstart
xn=x0; yn=y0; vxn=vx0; vyn=vy0


#define arrays to store data for plotting/analysis
x=[]; y=[]; vx=[]; vy=[];
E=[0,]; L=[0,]; T=[tstart,]


#main time-stepping loop using 1st order Euler
while t<tend:
 rn=math.sqrt(xn*xn+yn*yn);
 Fxn=-xn/rn**3; Fyn=-yn/rn**3
 xnp1=xn+h*vxn
 ynp1=yn+h*vyn
 vxnp1=vxn+h*Fxn
 vynp1=vyn+h*Fyn
 t=t+h

 En=0.5*(vxnp1**2+vynp1**2)-GM/rn
 Efe=abs((En-E0)/E0)

 Ln=xnp1*vynp1-ynp1*vxnp1
 Lfe=abs((Ln-L0)/L0)
```

```
 xn=xnp1; yn=ynp1; vxn=vxnp1; vyn=vynp1;
 x.append(xn); y.append(yn); vx.append(vxn); vy.append(vyn);
 E.append(Efe); L.append(Lfe); T.append(t)
```

```
w=[]; z=[]
C = np.arange(0,2*math.pi,0.01)
b=math.sqrt(1-e**2)
for i in C:
    w.append(math.cos(i)+e); z.append(b*math.sin(i))
```

```
#plot results
plt.plot(x,y)
plt.plot(w,z)
plt.show()

plt.plot(T,E)
plt.yscale("log")
plt.xscale("log")
plt.show()

plt.plot(T,L)
plt.yscale("log")
plt.xscale("log")
plt.show()

print(T[-1])
```

Leapfrog:

```
import matplotlib.pyplot as plt
import math
import numpy as np
import pickle
#LF

GM=1
e=0.9

#define parameters
P=2*math.pi
```

```
tstart=0
tend=10**2*P
h=P/1000


#set initial conditions
x0=(1+e); y0=0; vx0=0; vy0=((1-e)/(1+e))**0.5;
L0=x0*vy0-y0*vx0; E0=0.5*(vx0**2+vy0**2)-GM/((x0**2+y0**2)**0.5);
t=tstart
xn=x0; yn=y0; vxn=vx0; vyn=vy0;


#define arrays to store data for plotting/analysis
x=[]; y=[]; vx=[]; vy=[];
E=[0,]; L=[0,]; T=[tstart,]


#main time-stepping loop using 1st order Euler
while t<tend:
 xqn=xn+(h/2)*vxn
 yqn=yn+(h/2)*vyn

 rn=math.sqrt(xqn*xqn+yqn*yqn);

 Fxqn=-xqn/rn**3; Fyqn=-yqn/rn**3

 vxnp1=vxn+h*Fxqn
 vynp1=vyn+h*Fyqn

 xnp1=xqn+(h/2)*vxnp1
 ynp1=yqn+(h/2)*vynp1

 t=t+h

 En=0.5*(vxnp1**2+vynp1**2)-GM/((xnp1**2+ynp1**2)**0.5)
 Efe=abs((En-E0)/E0)

 Ln=xnp1*vynp1-ynp1*vxnp1
 Lfe=abs((Ln-L0)/L0)


 xn=xnp1; yn=ynp1; vxn=vxnp1; vyn=vynp1;

 x.append(xn); y.append(yn); vx.append(vxn); vy.append(vyn);
 E.append(Efe); L.append(Lfe); T.append(t)
```

```
w=[]; z=[]
C = np.arange(0,2*math.pi,0.01)
b=math.sqrt(1-e**2)
for i in C:
    w.append(math.cos(i)+e); z.append(b*math.sin(i))



#plot results
plt.plot(x,y)
plt.plot(w,z)
plt.show()

plt.plot(T,E)
plt.yscale("log")
plt.xscale("log")
plt.show()

plt.plot(T,L)
plt.yscale("log")
plt.xscale("log")
plt.show()

pickle.dump(x, open("xiii", "wb"))
pickle.dump(y, open("yiii", "wb"))
pickle.dump(E, open("Eiii", "wb"))
pickle.dump(L, open("Liii", "wb"))
```

Modified Euler:

```
import matplotlib.pyplot as plt
import math
import numpy as np
import pickle
#ME

GM=1
e=0.5

#define parameters
P=2*math.pi
tstart=0
tend=10**2*P
h=P/300


#set initial conditions
x0=(1+e); y0=0; vx0=0; vy0=((1-e)/(1+e))**0.5;
L0=x0*vy0-y0*vx0; E0=0.5*(vx0**2+vy0**2)-GM/((x0**2+y0**2)**0.5);
```

```
t=tstart
xn=x0; yn=y0; vxn=vx0; vyn=vy0


#define arrays to store data for plotting/analysis
x=[]; y=[]; vx=[]; vy=[];
E=[0,]; L=[0,]; T=[tstart,]


#main time-stepping loop using 1st order Euler
while t<tend:
 xnp1=xn+h*vxn
 ynp1=yn+h*vyn

 rn=math.sqrt(xnp1*xnp1+ynp1*ynp1);
 Fxnp1=-xnp1/rn**3; Fynp1=-ynp1/rn**3

 vxnp1=vxn+h*Fxnp1
 vynp1=vyn+h*Fynp1
 t=t+h


 En=0.5*(vxnp1**2+vynp1**2)-GM/rn
 Efe=abs((En-E0)/E0)

 Ln=xnp1*vynp1-ynp1*vxnp1
 Lfe=abs((Ln-L0)/L0)



 xn=xnp1; yn=ynp1; vxn=vxnp1; vyn=vynp1;
 x.append(xn); y.append(yn); vx.append(vxn); vy.append(vyn);
 E.append(Efe); L.append(Lfe); T.append(t)




w=[]; z=[]
C = np.arange(0,2*math.pi,0.01)
b=math.sqrt(1-e**2)
for i in C:
    w.append(math.cos(i)+e); z.append(b*math.sin(i))



#plot results
plt.plot(x,y)
plt.plot(w,z)
```

```
plt.show()

plt.plot(T,E)
plt.yscale("log")
plt.xscale("log")
plt.show()

plt.plot(T,L)
plt.yscale("log")
plt.xscale("log")
plt.show()


pickle.dump(x, open("xii", "wb"))
pickle.dump(y, open("yii", "wb"))
pickle.dump(E, open("Eii", "wb"))
pickle.dump(L, open("Lii", "wb"))
```

RK4:

```
import matplotlib.pyplot as plt
import math
import numpy as np
import pickle
#RK4

GM=1
e=0.5

#define parameters
P=2*math.pi
tstart=0
tend=10**2*P
h=P/75


t=tstart
k1=[0,0,0,0]; k2=[0,0,0,0]; k3=[0,0,0,0]; k4=[0,0,0,0]; wn=[(1+e),0,0,((1-e)/(1+e))**0.5];
E0=0.5*(wn[2]**2+wn[3]**2)-GM/((wn[0]**2+wn[1]**2)**0.5);
L0=wn[0]*wn[3]-wn[1]*wn[2];


#define arrays to store data for plotting/analysis
x=[]; y=[]; vx=[]; vy=[];
E=[0,]; L=[0,]; T=[tstart,]

#h*f function
def hf(xn, yn, vxn, vyn):
    rn=math.sqrt(xn*xn+yn*yn);
```

```
    return [h*vxn, h*vyn, -h*xn/rn**3, -h*yn/rn**3]

#main time-stepping loop using Runge-Kutta
while t<tend:

    k1= hf(wn[0], wn[1], wn[2], wn[3])

    k2= hf(wn[0]+(1/2)*k1[0], wn[1]+(1/2)*k1[1], wn[2]+(1/2)*k1[2], wn[3]+(1/2)*k1[3])

    k3= hf(wn[0]+(1/2)*k2[0], wn[1]+(1/2)*k2[1], wn[2]+(1/2)*k2[2], wn[3]+(1/2)*k2[3])

    k4= hf(wn[0]+k3[0], wn[1]+k3[1], wn[2]+k3[2], wn[3]+k3[3])

    for i in range(4):
        wn[i]=  wn[i]+(1/6)*(k1[i]+2*k2[i]+2*k3[i]+k4[i])



    En=0.5*(wn[2]**2+wn[3]**2)-GM/((wn[0]**2+wn[1]**2)**0.5)
    Efe=abs((En-E0)/E0)

    Ln=wn[0]*wn[3]-wn[1]*wn[2]
    Lfe=abs((Ln-L0)/L0)



    t=t+h
    x.append(wn[0]); y.append(wn[1]); vx.append(wn[2]); vy.append(wn[3]);
    E.append(Efe); L.append(Lfe); T.append(t)

w=[]; z=[]
C = np.arange(0,2*math.pi,0.01)
b=math.sqrt(1-e**2)
for i in C:
    w.append(math.cos(i)+e); z.append(b*math.sin(i))


#plot results
plt.plot(x,y)
plt.plot(w,z)
plt.show()

plt.plot(T,E)
plt.yscale("log")
plt.xscale("log")
plt.show()

plt.plot(T,L)
```

```
plt.yscale("log")
plt.xscale("log")
plt.show()

pickle.dump(T, open("Tiv", "wb"))
pickle.dump(x, open("xiv", "wb"))
pickle.dump(y, open("yiv", "wb"))
pickle.dump(E, open("Eiv", "wb"))
pickle.dump(L, open("Liv", "wb"))
```

Angular Momentum Plot:

```
import pickle
import matplotlib.pyplot as plt

T=pickle.load(open("T","rb"))
Tiv=pickle.load(open("Tiv","rb"))
Li=pickle.load(open("Li","rb"))
Lii=pickle.load(open("Lii","rb"))
Liii=pickle.load(open("Liii","rb"))
Liv=pickle.load(open("Liv","rb"))

plt.plot(T,Li, label='Forward Euler', color='C1')
plt.plot(T,Lii, label='Modified Euler', color='C2')
plt.plot(T,Liii, label='Leapfrog', color='C3')
plt.plot(Tiv,Liv, label='Runge-Kutta', color='C4')

plt.yscale("log")
plt.xscale("log")

plt.xlabel('time (t)')
plt.ylabel('Fractional error')
plt.legend()


plt.show()
```

Energy Plot:

```
import pickle
import matplotlib.pyplot as plt


T=pickle.load(open("T","rb"))
Tiv=pickle.load(open("Tiv","rb"))
Ei=pickle.load(open("Ei","rb"))
Eii=pickle.load(open("Eii","rb"))
Eiii=pickle.load(open("Eiii","rb"))
Eiv=pickle.load(open("Eiv","rb"))
```

```
plt.plot(T,Ei, label='Forward Euler', color='C1')
plt.plot(T,Eii, label='Modified Euler', color='C2')
plt.plot(T,Eiii, label='Leapfrog', color='C3')
plt.plot(Tiv,Eiv, label= 'Runge-Kutta', color='C4')

plt.yscale("log")
plt.xscale("log")

plt.xlabel('Time (t)')
plt.ylabel('Fractional error')
plt.legend()


plt.show()
```

Orbit Plots:

```
import pickle
import matplotlib.pyplot as plt
import math
import numpy as np

xi=pickle.load(open("xi","rb"))
xii=pickle.load(open("xii","rb"))
xiii=pickle.load(open("xiii","rb"))
xiv=pickle.load(open("xiv","rb"))

yi=pickle.load(open("yi","rb"))
yii=pickle.load(open("yii","rb"))
yiii=pickle.load(open("yiii","rb"))
yiv=pickle.load(open("yiv","rb"))

w=[]; z=[]
C = np.arange(0,2*math.pi,0.01)
b=math.sqrt(1-0.5**2)
for i in C:
    w.append(math.cos(i)+0.5); z.append(b*math.sin(i))

plt.plot(0,0, marker='o')
plt.annotate('Star', (0.05, 0))

plt.plot(xi,yi, label='Forward Euler')
plt.plot(xii,yii, label='Modified Euler')
plt.plot(xiii,yiii, label='Leapfrog')
plt.plot(xiv,yiv, label='Runge-Kutta')
plt.plot(w,z, label=('Analytical Prediction'))
```

```
plt.xlabel('x')
plt.ylabel('y')
plt.legend()

plt.xlim([-1.5,5])
plt.ylim([-2,2])

plt.show()
```

B Heat equation Appendix

Plot of (x,θ):

```
import matplotlib.pyplot as plt
import math
import numpy as np

L=1
K=1
N=5
C=1/3

Dx=L/(N-1)

tstart=0
tend=1
Dt=C*Dx**2


#set initial conditions
t=tstart
tn=[]
T = [[] for i in range(N)]


for i in range(N):
    T[i].append(math.sin(i*Dx*math.pi))


#numerical calculation for time steps
t=t+Dt
indx=0;
while t<tend:

    T[0].append(0)

    for i in range(1,N-1):
        T[i].append(T[i][indx]+C*(T[i+1][indx]-2*T[i][indx]+T[i-1][indx]))

    T[N-1].append(0)

    tn.append(t)
    indx=indx+1
    t=t+Dt

xn=[]
x=0
```

```
for i in range(N):
    xn.append(x)
    x=x+Dx



#analytical calculation for time step
Ta = [[] for i in range(1000)]
t=tstart
ta=[]
xa=[]
x=0
Dx=L/(1000-1)

while t<tend:
    for i in range(1000):
        Ta[i].append(math.sin(math.pi*i*Dx)*math.e**(-math.pi**2*t))

    t=t+Dt


for i in range(1000):
    xa.append(x)
    x=x+Dx



#collecting the data
Tnt=[]
Tat=[]

for i in range(N):
    Tnt.append(T[i][12])

for i in range(1000):
    Tat.append(Ta[i][12])



plt.plot(xn,Tnt, label='Numerical', linestyle='', marker='o')
plt.plot(xa,Tat, label='Analytical')
plt.xlabel('x')
plt.ylabel('$\\theta}$')
plt.legend()
plt.show()
```

Plot of (C,E):

```
import matplotlib.pyplot as plt
```

```
import math
import numpy as np


CC=[i for i in np.arange(0.05, 1, 0.01)]
EC=[]

for i in CC:


    L=1
    K=1
    N=5
    C=i

    Dx=L/(N-1)

    tstart=0
    tend=1
    Dt=C*Dx**2


    #set initial conditions
    t=tstart
    tn=[]
    T = [[] for i in range(N)]


    for i in range(N):
        T[i].append(math.sin(i*Dx*math.pi))


    #numerical calculation for time steps
    t=t+Dt

    indx=0;
    while t<tend:

        T[0].append(0)

        for i in range(1,N-1):
            T[i].append(T[i][indx]+C*(T[i+1][indx]-2*T[i][indx]+T[i-1][indx]))

        T[N-1].append(0)

        tn.append(t)
        indx=indx+1
        t=t+Dt
```

```
    xn=[]
    x=0
    for i in range(N):
        xn.append(x)
        x=x+Dx



    ex=[]
    t=0.99

    for i in range(N):
        ex.append(abs(T[i][int(t/Dt)]-math.sin(math.pi*i*Dx)*math.e**(-math.pi**2*t)))



    E=0

    for i in range(N):
        E=E+ex[i]**2
    E=E**0.5

    EC.append(E)

plt.plot(CC,EC, linestyle='', marker='o')
plt.xlabel('C')
plt.ylabel('E')
plt.legend()
plt.ylim(ymin=0)
plt.show()
```

Plot of (N,E):

```
import matplotlib.pyplot as plt
import math
import numpy as np


NN=[i for i in range(3 ,50)]
EN=[]

for i in NN:


    L=1
    K=1
    N=i
    C=0.1
```

```
Dx=L/(N-1)

tstart=0
tend=1
Dt=C*Dx**2


#set initial conditions
t=tstart
tn=[]
T = [[] for i in range(N)]


for i in range(N):
    T[i].append(math.sin(i*Dx*math.pi))


#numerical calculation for time steps
t=t+Dt

indx=0;
while t<tend:

    T[0].append(0)

    for i in range(1,N-1):
        T[i].append(T[i][indx]+C*(T[i+1][indx]-2*T[i][indx]+T[i-1][indx]))

    T[N-1].append(0)

    tn.append(t)
    indx=indx+1
    t=t+Dt

xn=[]
x=0
for i in range(N):
    xn.append(x)
    x=x+Dx



ex=[]
t=0.75

for i in range(N):
    ex.append(abs(T[i][int(t/Dt)]-math.sin(math.pi*i*Dx)*math.e**(-math.pi**2*t)))
```

```
    E=0

    for i in range(N):
        E=E+ex[i]**2
    E=E**0.5

    EN.append(E)

plt.plot(NN,EN, linestyle='', marker='o')
plt.xlabel('N')
plt.ylabel('E')
plt.legend()
plt.show()
```

Plot of (x,θ) at multiple times:

```
import matplotlib.pyplot as plt
import math
import numpy as np

L=1
K=1
N=10
C=2/3

Dx=L/(N-1)

tstart=0
tend=1
Dt=C*Dx**2


#set initial conditions
t=tstart
tn=[]
T = [[] for i in range(N)]


for i in range(N):
    T[i].append(math.sin(i*Dx*math.pi))


#numerical calculation for time steps
t=t+Dt

indx=0;
```

```
while t<tend:

    T[0].append(0)

    for i in range(1,N-1):
        T[i].append(T[i][indx]+C*(T[i+1][indx]-2*T[i][indx]+T[i-1][indx]))

    T[N-1].append(0)

    tn.append(t)
    indx=indx+1
    t=t+Dt

xn=[]
x=0
for i in range(N):
    xn.append(x)
    x=x+Dx



#analytical calculation for time step
Ta = [[] for i in range(1000)]
t=tstart
ta=[]
xa=[]
x=0
Dx=L/(1000-1)

while t<tend:
    for i in range(1000):
        Ta[i].append(math.sin(math.pi*i*Dx)*math.e**(-math.pi**2*t))

    t=t+Dt


for i in range(1000):
    xa.append(x)
    x=x+Dx



#collecting the data
p=4
Tnt=[[] for i in range(p)]
Tat=[[] for i in range(p)]

for j in range (p):
```

```
    for i in range(N):
        Tnt[j].append(T[i][60+j*5])

    for i in range(1000):
        Tat[j].append(Ta[i][60+j*5])



plt.plot(xn,Tnt[0], label='t=0.49', linestyle='-', marker='o')
plt.plot(xn,Tnt[1], label='t=0.53', linestyle='-', marker='o')
plt.plot(xn,Tnt[2], label='t=0.57', linestyle='-', marker='o')
plt.plot(xn,Tnt[3], label='t=0.61', linestyle='-', marker='o')


plt.xlabel('x')
plt.ylabel('$\\theta}$')
plt.legend()
plt.show()
```

Crank-Nicolson:

```
import matplotlib.pyplot as plt
import math
import numpy as np

L=1
K=1
N=10
C=2/3

Dx=L/(N-1)

tstart=0
tend=1
Dt=C*Dx**2


#set initial conditions
t=tstart
tn=[]
T = [[] for i in range(N)]


for i in range(N):
    T[i].append(math.sin(i*Dx*math.pi))


#numerical calculation for time steps
A=[[1+C,-C/2,0,0,0,0,0,0],
```

```
   [-C/2,1+C,-C/2,0,0,0,0,0],
   [0,-C/2, 1+C,-C/2,0,0,0,0],
   [0,0,-C/2,1+C,-C/2,0,0,0],
   [0,0,0,-C/2,1+C,-C/2,0,0],
   [0,0,0,0,-C/2,1+C,-C/2,0],
   [0,0,0,0,0,-C/2,1+C,-C/2],
   [0,0,0,0,0,0,-C/2,1+C]]


t=t+Dt
indx=0;
while t<tend:
    T[0].append(0)
    T[N-1].append(0)


    b=[]
    for i in range(1,N-1):
        b.append(0.5*C*T[i-1][indx]+(1-C)*T[i][indx]+0.5*C*T[i+1][indx])



    for i in range (1,N-1):
        T[i].append(np.linalg.solve(A,b)[i-1])



    tn.append(t)


    indx=indx+1
    t=t+Dt



xn=[]
x=0
for i in range(N):
    xn.append(x)
    x=x+Dx



#analytical calculation for time step
Ta = [[] for i in range(1000)]
t=tstart
ta=[]
xa=[]
```

```
x=0
Dx=L/(1000-1)

while t<tend:
    for i in range(1000):
        Ta[i].append(math.sin(math.pi*i*Dx)*math.e**(-math.pi**2*t))

    t=t+Dt


for i in range(1000):
    xa.append(x)
    x=x+Dx



#collecting the data
Tnt=[]
Tat=[]

for i in range(N):
    Tnt.append(T[i][121])

for i in range(1000):
    Tat.append(Ta[i][121])



plt.plot(xn,Tnt, label='Numerical', linestyle='', marker='o')
plt.plot(xa,Tat, label='Analytical')
plt.xlabel('x')
plt.ylabel('T')
plt.legend()
plt.show()
```

C Random numbers integration Appendix

Monte-Carlo and Trapezium rule, (N,E) plots:

```
import matplotlib.pyplot as plt
import math
import numpy as np
import random

ac=0.5*(math.sin(4)/4+1)


NN=[i for i in np.arange(3,200)]
MaEN=[]
MnEN=[]
TaEN=[]
TnEN=[]

for i in NN:

    n=i
    xmin=0
    xmax=2
    t=0


    #MC


    #getting sample x values
    xvalues=[]
    while t<n:
        xvalues.append(random.uniform(xmin, xmax))
        t=t+1

    #finding y values for corresponding sample x values
    yvalues=[]
    for i in xvalues:
        yvalues.append(0.5*(math.cos(i))**2)

    #Estimation using monte-Carlo sampling
    fint=((xmax-xmin)/n)*sum(yvalues)

    MaEN.append((xmax-xmin)*(np.std(yvalues))/n**0.5)
    MnEN.append(abs(fint-ac))


    #Trap
```

```
    step=(xmax-xmin)/(n-1)

    #getting x values
    xvalues=[xmin]
    t=1
    while t<n:
        xvalues.append(xvalues[-1]+step)
        t=t+1


    #finding y values for corresponding sample x values
    yvalues=[]
    for i in xvalues:
        yvalues.append(0.5*(math.cos(i))**2)

    #calucating area of n-1 trapeziums
    trapa=[]
    t=1

    while t<n:
        trapa.append(0.5*(xvalues[t]-xvalues[t-1])*(yvalues[t]+yvalues[t-1]))
        t=t+1

    #Estimation using trapezium rule
    fint=sum(trapa)

    #errors
    TaEN.append(8/(12*n**2))
    TnEN.append(abs(fint-ac))


#plot
plt.plot(NN,MnEN, label='Numerical error')
plt.plot(NN,MaEN, label='Analytical error')
plt.xlabel('N')
plt.ylabel('E')

plt.ylim(ymin=0)
plt.legend()
plt.show

print(np.polyfit(np.log10(NN),np.log10(MnEN),1))
print(np.polyfit(np.log10(NN),np.log10(TnEN),1))
```

Monte-Carlo, area of a circle, with (N,E) plot:

```
import matplotlib.pyplot as plt
import math
```

```
import numpy as np
import random

NN=[i for i in np.arange(3,200)]
aEN=[[] for i in NN]
nEN=[[] for i in NN]

for j in NN:
    M=100
    m=0

    while m<M:
        n=j
        x1min=-1
        x1max=1
        x2min=-1
        x2max=1
        V=(x1max-x1min)*(x2max-x2min)
        t=0

        #getting sample x values
        coords=[[random.uniform(x1min, x1max),random.uniform(x2min, x2max)] for i in range(n)]


        #finding y values for corresponding sample x values
        fx=[]
        for i in coords:
            if i[0]**2+i[1]**2<1:
                fx.append(1)

            else:
                fx.append(0)




        #Estimation using monte-Carlo sampling
        fint=(V/n)*sum(fx)

        #analytical area
        ac=math.pi

        #finding error of the estimation
        error=abs(fint-ac)


        aEN[int(j-3)].append((x1max-x1min)*(x2max-x2min)*(np.std(fx))/n**0.5)
        nEN[int(j-3)].append(error)
```

```
        m=m+1


maEN=[np.mean(i) for i in aEN]
mnEN=[np.mean(i) for i in nEN]



#plot
f=np.polyfit(np.log10(NN),np.log10(mnEN),1)
x= np.array(range(2 ,200))
y= x**-0.5 +0.03



plt.plot(NN,maEN, label='Analytical')
plt.plot(NN,mnEN, label='Numerical')
plt.plot(x,y, label='E=1/N^0.5')

plt.xlabel('N')
plt.ylabel('E')
plt.legend()
plt.xlim(xmin=0)



plt.show
```

Circle in D dimensions:

```
import matplotlib.pyplot as plt
import math
import numpy as np
import random


n=10000
D=7
xmin=-1
xmax=1
V=(xmax-xmin)**D
t=0

#getting sample x values
coords=[[random.uniform(xmin, xmax) for i in range(D) ] for i in range(n)]
```

```
#finding y values for corresponding sample x values
fx=[]
for i in coords:
    sqs=[]
    for j in i:
        sqs.append(j**2)


    if sum(sqs)<1:
        fx.append(1)

    else:
        fx.append(0)




#Estimation using monte-Carlo sampling
fint=(V/n)*sum(fx)


#finding error of the estimation
ac=(16/105)*(math.pi**3)
error=abs(ac-fint)

print(ac)
print(fint)
print(error)
```