Assignment 3 Report

David Hor, 21703067

Introduction:

This assignment focused on numerically evaluating different methods of taking the derivatives of functions and moved up to evaluating Ordinary Differential Equations numerically. The first exercise involved numerical differentiation of sin(x) using finite differences, specifically forward, backward, and central differences. The second exercise utilised the Forward and Leap frog Euler methods and the third and final exercise involved a second order Runge-Kutta method.

Exercises:

Ex1.1: define the functions for forward, backward, and central.

Done in coding script. Coded it into one function.

Ex1.2: consider step sizes from 0.1, 0.01, 0.001…….10^{-15} for each method.

Also done in coding script. Since it is a list of 15 values, I am not copying the answers into here. Running my code should output the whole list.

Ex1.3: Plots in loglog

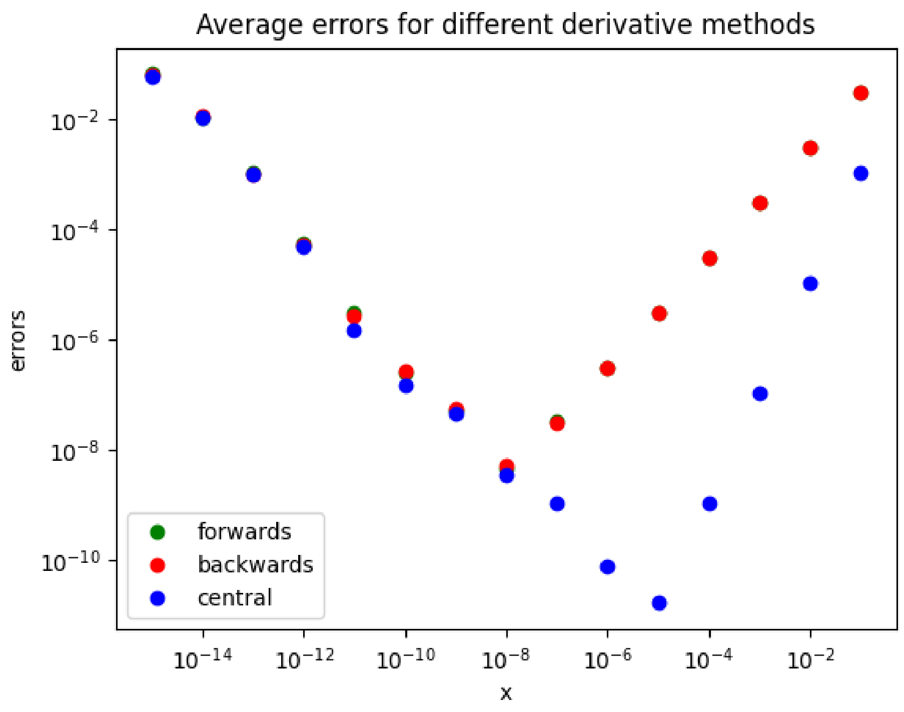

Figure 1: Average errors using the forwards, backwards and central difference methods for finding the deriative at different step sizes.

Ex1.4: Relate the slopes of the lines (in the region where error is decreasing with decreasing h) for the three methods to their known h error scaling.

The slopes in the lines show that for the forwards and backwards difference methods, that they both reach an error of approximately 5×10^{-8} at a step size of 10^{-8}. The forwards and backwards difference methods are so similar that the points are essentially on top of each other and cannot be differentiated. The central difference method reaches a lower error of 10^{-10} at a larger step size of 10^{-5} then the errors start to increase and reaches a similar error at 10^{-8}. Overall, the central difference method seems to have the least error for all step sizes.

Ex1.5: Why does error start to increase as step size decrease below certain point.
The error starts to increase after a certain point due to the inability of a computer to return the correct number of decimals correctly due to the memory. The reason why forwards/backwards differs in this from the central is that the central method modifies both x and y values by step size making it have a lower error at a larger step size as it is a better method for optimisation with less computing power.

Ex2.1: write function for forward Euler method.
The function is defined in the python script and the following is a plot of the forward Euler method in comparison to the final output of the decay of excited atoms with a time interval of 25 (I am not sure if this is minutes or seconds). It shows that the forward Euler results in a very close approximation of the actual value.

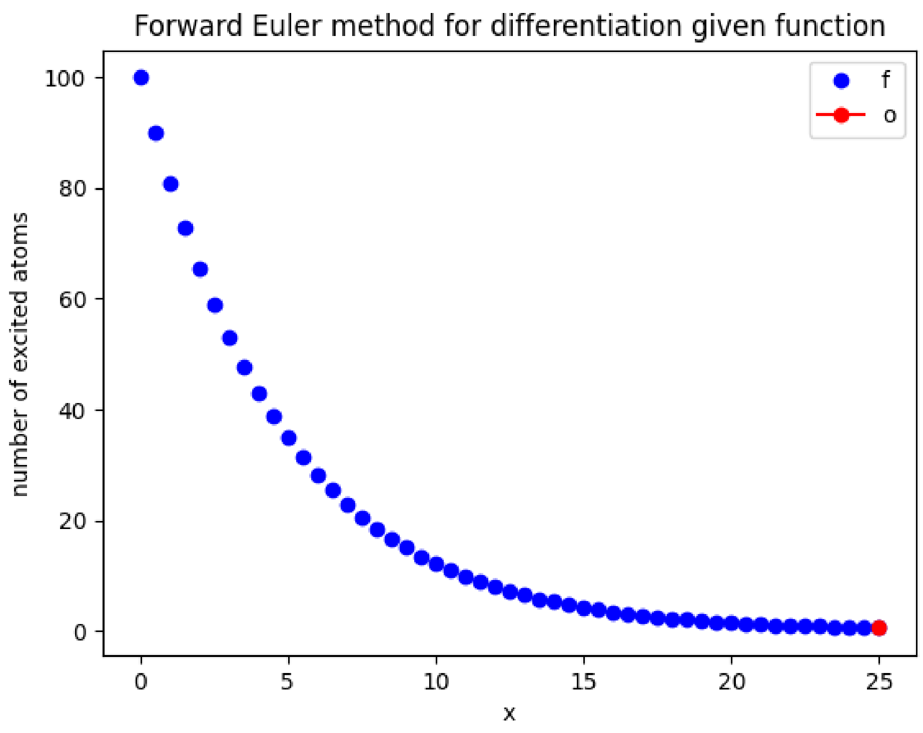

Figure 2: The Forward Euler method showing the decay of excited atoms compared with the final value of the actual equation shown in red.

Ex2.2: do the same for leapfrog method.
The leapfrog method is defined in the python script and a plot is featured below in figure 3. It is shown to be the same as the forward Euler method for this problem from graph comparisons.

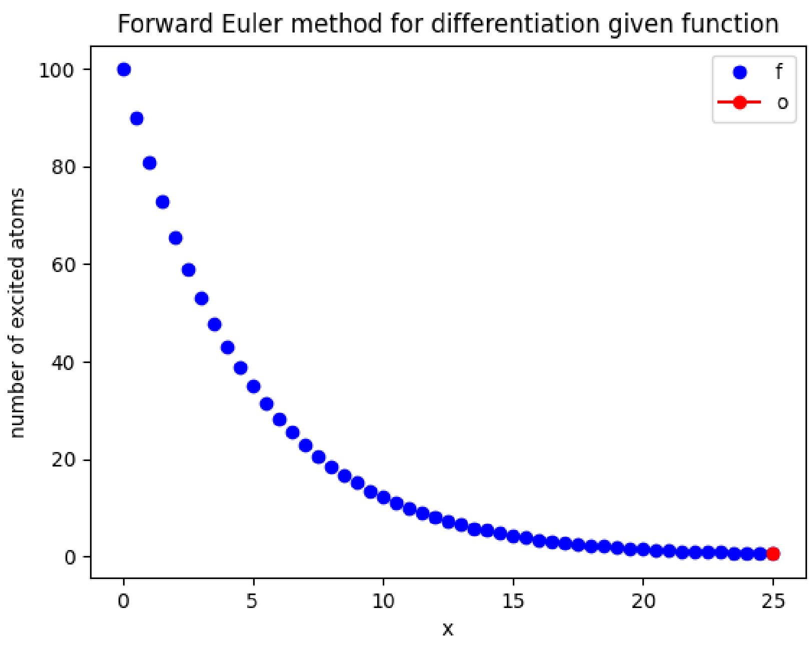

Figure 3: The leapfrog method in a similar situation to that of the Ex 2.1.

Ex2.3: calculate fractional error on log-log scale.
The fractional error is shown in figure 4.

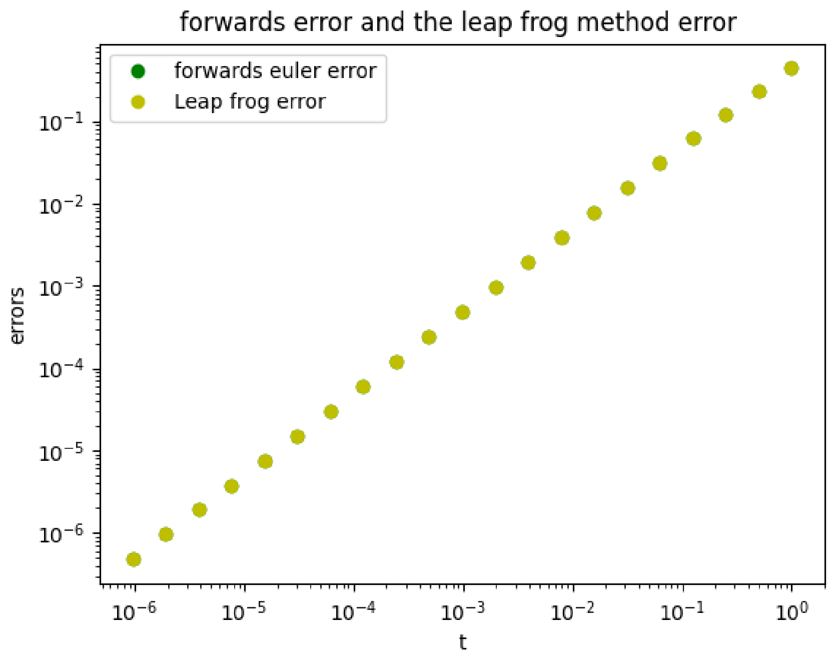

Figure 4: the forward and leapfrog Euler methods fractional error when compared to the actual value

It seems that the fractional error for both methods is the same when they should not be. A possible reason for this is because I am running this on my laptop with less processing power in comparison to say my gaming computer (probably not true but I like to flex). This is mainly an issue as we are probably running into float overflow.

Ex2.4: How do methods compare?
The methods seem to be the same when computing this ODE with the exact same errors. When I compared the outputs of the forwards Euler and leap-frog Euler, the output was an array of "True".

Ex3.1: Define the Runge-Kutta function within the range of 0 and 1.
See python script.
Ex3.2:
Using a fractional error method like that in 2.3 for t = 1, for a similar set of step sizes, A plot of the fractional errors is shown in figure 5.

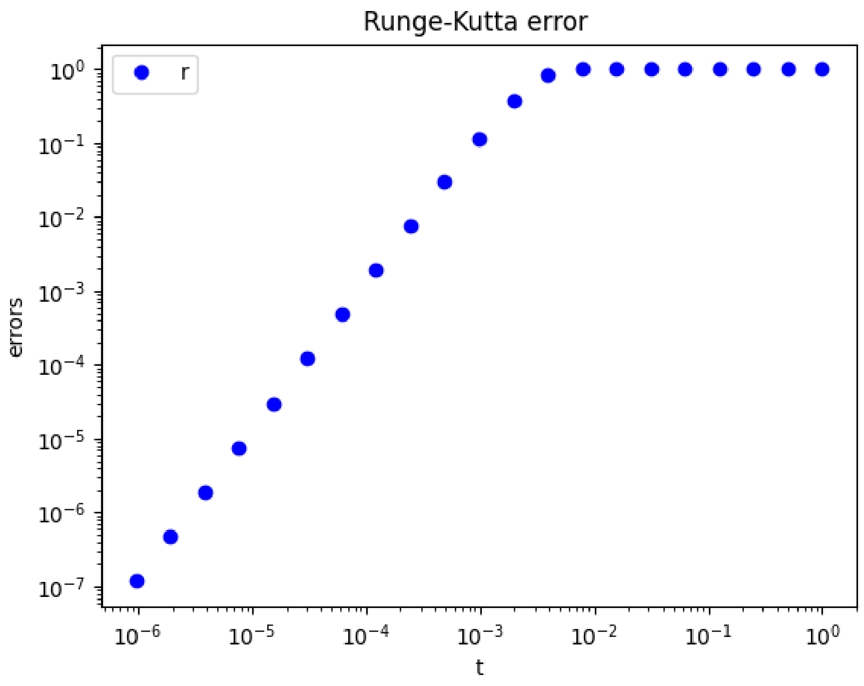

Figure 5: The fractional error of the Runge-Kutta second order modified Euler algorithm.

This shows that for lower step sizes, the fractional error decreases as the step size decreases. Hence this means that of all methods used, the Runge-Kutta method seems to be the only one with a decreasing error at the maximum step size. This could be because it is at a point just before float overflow kicks in as any smaller step sizes will result in another increase of error.
Conclusion:
This assignment shows us how to numerically solve ODE's using python via many different methods and comparing their effectiveness to each other. From the exercises conducted, the main issue encountered is float overflow which is when a computer is unable to assign the required points of precision.