Computer simulation: Numerical Methods, assignment 3

Name: Rohan Kadam

Student number: 20334092

Introduction:

In this assignment we are tasked with studying the numerical solutions of an ordinary differential equation. The differential equation in question is $\frac{dx}{dt} = (1+t)x + 1 - 3t + t^2$

We first must remind ourselves of what exactly a differential equation is, and when they are used. In mathematics we define a differential equation as an equation which contains one or more derivatives of a function, where the first derivative of the function is given by dy/dx. For example a first order differential equation is one which contains this first derivative as its highest order derivative. Differential equations are used in a wide variety and range of disciplines, such as physics, engineering, chemistry and economics, specifically in the area of physics differential equations are often used to describe the state of a system i.e. in terms of heat transfer, fluid flow, and particle motion etc. Differential equations are usually classed into groups such as ordinary, linear, nonlinear, partial, homogenous and non-homogeneous, however the differential equation we will be dealing with is one of the first order linear ordinary differential equation. It is an ordinary differential equation as the derivative it taken with respect to one variable. We will try to computationally solve this differential equation using different methods in this assignment, and pseudocode will be provided to describe how I carried out each task. The two python packages required for this assignment were numpy and matplotlib.

Task 1

We are asked to plot a direction field for t, in certain parameters that are given to us. This is a concept which needed further research in order to understand and grasp its complexities. The book *Calculus to Chaos by D Acheson* was used to achieve this. After reading the relevant chapter/s I had an understanding for what the direction field is and made my own evaluation. I recognized that the directional field is a way of graphically representing the solutions of a first order differential equation without actually solving the equation analytically. The general idea was to first choose a set of grid points for the graph (specified for us in this task), then at each points compute the slope given by the differential equation using the x and y values of that point, and finally to draw a short line segment that has the slope computed. After further research I discovered it is possible to sketch a directional field for a differential equation using simply pen and paper. I decided to carry out this extra task to be aware of how my direction field graph should look like when plotting it through python. Following this an attempt was made to sketch the directional field as shown below.

$$\frac{dx}{dt} = (1+t)x + 1 - 3t + t^2$$

x	0	0	0	0	0	0
t	0	1	2	3	4	5
t'	1	-1	-1	1	5	9

x	1	1	1	1	1	1
t	0	1	2	3	4	5
t'	2	1	2	5	10	17

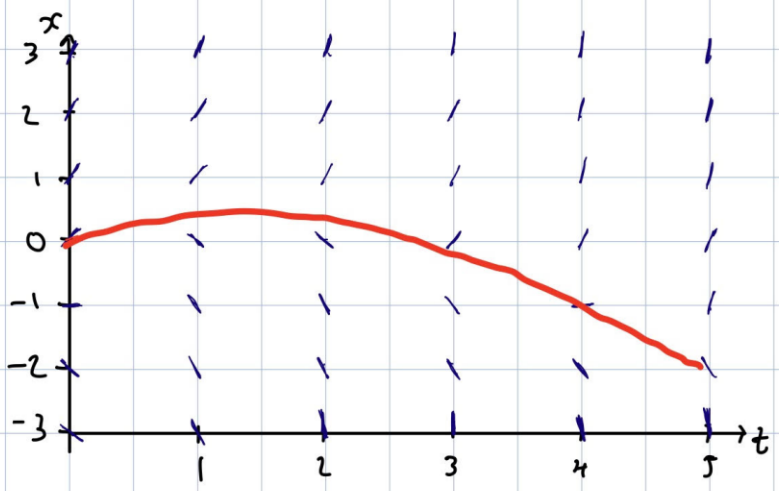
x	2	2	2	2	2	2
t	0	1	2	3	4	5
t'	3	3	5	9	15	21

x	3	3	3	3	3	3
t	0	1	2	3	4	5
t'	4	5	8	15	20	29

x	-1	-1	-1	-1	-1	-1
t	0	1	2	3	4	5
t'	0	-2	-4	-3	0	5

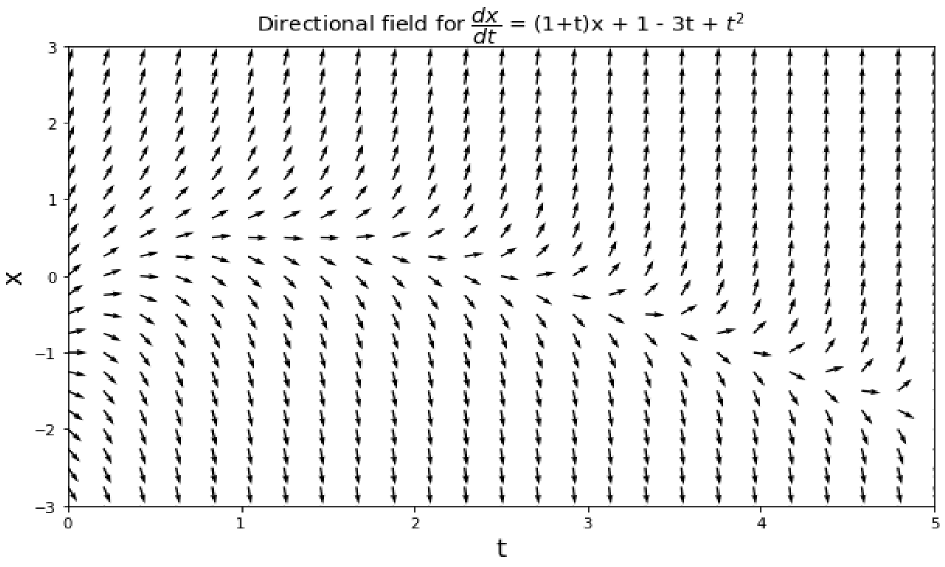
x	-2	-2	-2	-2	-2	-2
t	0	1	2	3	4	5
t'	-1	-5	-7	-7	-5	-1

x	-3	-3	-3	-3	-3	-3
t	0	1	2	3	4	5
t'	-2	-7	-10	-11	-10	-7



The direction field was sketched using the axis of t from 0 to 5 and x from -3 to 3. The slope was calculated for each point and a rough approximation of the slope was drawn. It was difficult to draw a difference between a slope of 15 and a slope of 20 for example due to the size of the axes used. In hindsight the slope of half points should also have been calculated to give a more accurate directional field. A very rough solution curve was also sketched with relation to the point (0,0) by following the direction of the slopes at each point following (0,0). The sketch was a bit of an imperfect science however; by having rough estimates of slopes made after approaching each point I wanted to go through. However this was only done to achieve an idea of what a computed directional field should look like.

To plot this directional field we first give the range and interval values we want of x and t by using the [numpy.linspace] function like [xrange = np.linspace(-3,3,25)]. To set a meshgrid which is used to create a rectangular grid out of two given one-dimensional arrays, we use the [numpy.meshgrid] function for the specified variables from the [numpy.linspace] already defined function like so [T,X = np.meshgrid(trange,xrange)]. We then normalise the arrows we want to use to give a more presentable graph by the numpy.ones function like [dx = (np.ones(f(T,X).shape))]. Now we simply plot the figure we want while also giving ranges on the x and y axis we want by using the matplotlib library.

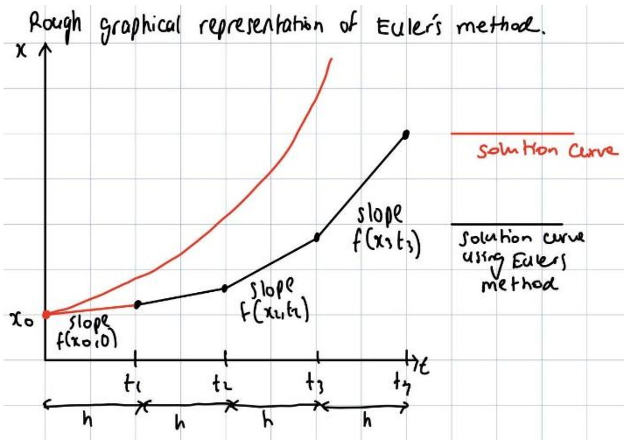


'Figure 1 above shows the direction field for the differential equation given'

As we can see the solution curve from our sketch is almost identical to the solution curve seen from the plotted directional field. We know we have plotted the correct directional field within the specified parameters and given constraints.

Task 2

From understanding the initial conditions and the complexities of the differential equations we are dealing with, an understanding was made that it cannot be solved analytically. We are tasked to solve/approximately solve the differential equation using Euler's method. We know from our reading from the book *'calculus to chaos'* that Euler's method is a very crude way to approach solving a differential equation. It is based on the assumption that the tangent line to the integral curve of our differential equation at $(x_i, t(x_i))$ approximates over the integral curve over an interval of $[x_i, x_{i+1}]$. As we know the slope of the integral curve using our initial conditions the equation of the corresponding tangent curve can also be expressed. We introduce the concept of a step size 'h' which is the 'distance' or 'length of the forward step' that we will take from going from approximation to approximation. We can already deduce that a smaller step size will most probably induce a more accurate approximation of a solution curve for Euler's method. By introducing the step size 'h' and allowing $x = x_{i+1} = x_i + h$ and by knowing x_0 we can yield an initial approximate equation $x_1 = x_0 + hf(x_0, 0)$. Using the same method by taking another 'step forward' and by using the slope of the line just calculated, we can derive an expression for the next line which is given by $x_2 = x_1 + hf(x_1, t_1)$. We advance in time steps of 'h' and increasing 'i' like so $x_{n+1} = x_n + hf(x_n, t_n)$ to obtain a solution curve.


Rough graphical representation of Euler's method.
x
Solution Curve
slope f(x₃,t₃)
Solution curve using Eulers method
slope f(x₂,t₂)
x₀
slope f(x₀,0)
t
t₁
t₂
t₃
t₄
h
h
h
h

'Figure 2 shows a rough graphical representation of Euler's method to find a solution curve our differential equation.'

The idea of Euler's method is to obtain a solution curve taking a large number of step sizes, but 'small' in nature to achieve an accurate approximation to the solution at any 't' for our differential equation. In figure 2 we can clearly see how a large step size of h will not yield an applicable solution curve. Computationally this problem is fixed however as described below.

To carry out Euler's method computationally we define a function with 3 variables with the equation we found above by [def SimpleEuler(t,x,step): return x+ step*f(t,x)] Notice the change of f(x,t) to f(t,x) for grid points on the t-x plane. We define the step size we want, x_0 and the start and end values of t we want. The crucial part of this code is to ensure we create a 'loop' to ensure the continuous updating of x and t, we do this by [for i in range(1,n): SimpleEulerl[i] = SimpleEuler(arranging[i-1], SimpleEulerl[i-1], step)]. We then simply plot the direction field and this curve on the same plot using the well-established techniques from the matplotlib library.

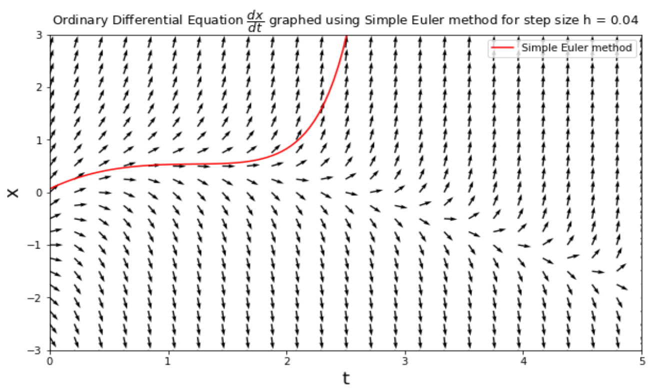


Figure 3 on the previous page shows the solution curve from the simple Euler method plotted with the direction field.

The solution curve plotted is one with 125 step sizes (5/0.04). As discussed earlier we know that decreasing the step size will yield a more accurate solution curve; this indicated to us the graph is not 100% accurate and there must be an element of error included. There is a name given to the major fault from Euler's method and this is called the truncation error which is usually given as 'O(h)'. In our situation this truncation error arises from the error from approximating the integral curve by the tangent line over the interval $[x_i, x_{i+1}]$ and the error when we replace $x(t_i)$ with x_i when computing x_{i+1}. The expression for the error will not be derived in this report however its effect on our solution curve will be discussed. For larger values of 'i' this error becomes a serious issue for obvious reasons. Over several steps the overestimates build up leading to the x_i's being significantly above the actual values of the $x(t_i)'s$. Euler's method is based on lines tangent to solution curves passing through all (x_i, t_i) points and not the tangent line approximations to the true solution which leads to this huge error. We overestimate the values and we end up with the points above the equilibrium and in a region where our solution curve diverges away from equilibrium. As we know our true solution curve is concave down in nature we expect Euler's solution curve to lie above this curve and this is exactly what we see.

Task 3

We are now going to plot a solution curve for the differential equation using the improved Euler method which is based on using the trapezoid rule for the integral evaluation. The improved Euler method takes two evaluations of f(x,t) at each step, a note was taken to remember how the with the simple Euler method we took only one evaluation of f(x,t) at each step. Immediately we can recognize that this method will be more accurate. The improved Euler method is based on approximating the integral curve of the differential equation at $(x_i, t(x_i))$ by the line through $(x_i, t(x_i))$ with a slope of m_i, where m_i is the average of the slopes of the tangents to the integral curve at the endpoints of $[x_i, x_{i+1}]$. Now just like in the simple Euler method we allow $x = x_{i+1} = x_i + h$ and allow $x_{n+1} = x_n + hf(x_n, t_n)$. Now we have a $x(t_{i+1})$ term so we replace it with x_i $+hf(x_i, t_i)$. Just like the simple Euler method we start with the known value of $x(t_0)$ =(x_0). This is a brief but accurate methodology of how the improved Euler method is formed, which is finally given by $x_{n+1} = x_n + \frac{h}{2}(f(x_n, t_n) + f\left(x_{n+1}, y_i + h(f(x_n, t_n))\right)$. This is computed in python in the exact same way as done with the Euler method in task 1 by first defining the function by [def ImprovedEuler(t,x,step): xnew = x + 0.5*step*(f(t,x) + f(t+step, x + step*f(t, x)))]. The following code does not need to be mentioned again as it is the same as done in task 1.

Now we need to define and plot the Runge-Kutta solution curve on the same graph. We will be implementing the 4th order Runge-Kutta method which is derived from the 2nd order method. In brief the 2nd order Runge-Kutta method consists of taking slopes at the beginning and midpoint of the time step, which yields a more accurate results than just using single slopes. To implement the fourth order Runge-Kutta method we take more slope approximations given by k1, k2, k3, and k4 to estimate the slope at different times. *k1* is given by the slope at the beginning of the time step, and k2 is the estimate of the slope at the midpoint if we use k1 to 'step halfway through the time step'. Similarly k3 is found from k2 as k2 was found from k1. To find k4 we use k3 to step all the way across the timestep $x_0 = x_0 + h$ so k4 is the estimate of the slope at the endpoint. The method is now computed by defining the function and the k(1,2,3,4) values by [def RungeKutta(t,x,step): k1 = f(t,x) k2 = f(t + 0.5*step, x + 0.5*step*k1) k3 = f(t + 0.5*step, x + 0.5*step*k2) k4 = f(t + step, x + step*k3) x_new = x + step/6.0*(k1 + 2.0*k2 + 2.0*k3 + k4]. We also do [ImprovedEulerl[i] = ImprovedEuler(arranging[i-1], ImprovedEulerl[i-1], step)] and [RungeKuttal[i] = RungeKutta(arranging[i-1], RungeKuttal[i-1], step)] just like we did for the simple Euler method for the same reasons. We know define the step size as 0.04 like the task requires us to for both the improved Euler method and Runge-Kutta method; we also plot the simple Euler method on the same graph for comparisons sake.

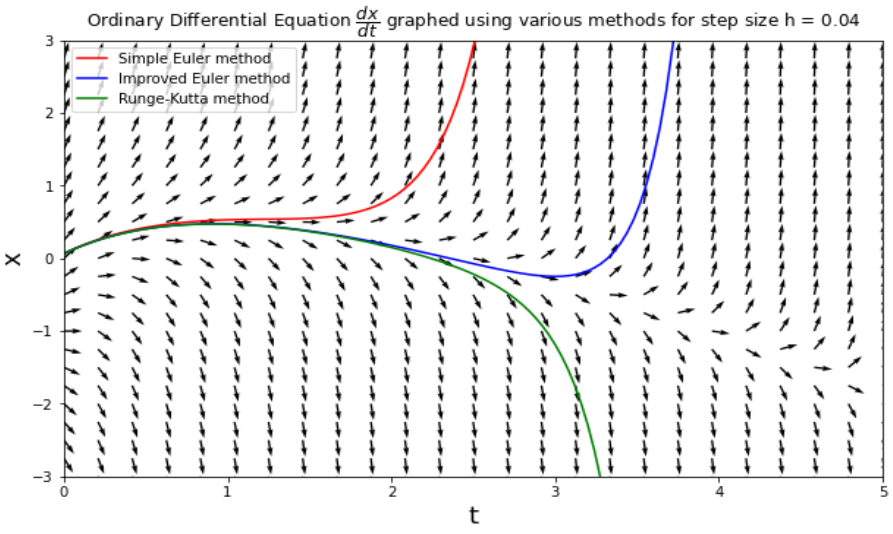


Figure 4 above shows the simple Euler, Improved Euler, and Runge-Kutta solution curves with step size h = 0.04, plotted along with the direction field for the differential equation.

From our results we can immediately see that the Runge-Kutta method is gives a more accurate solution curve, as expected, and the improved Euler solution curve tends towards the Runge-Kutta solution curve until a critical point before tending to +∞ like the simple Euler solution curve did. This shows how the improved Euler is more accurate than the simple Euler but not enough to give a true representation of the actual solution curve. We will try see if reducing the step size will lead to more accurate solution curves for all 3 methods.

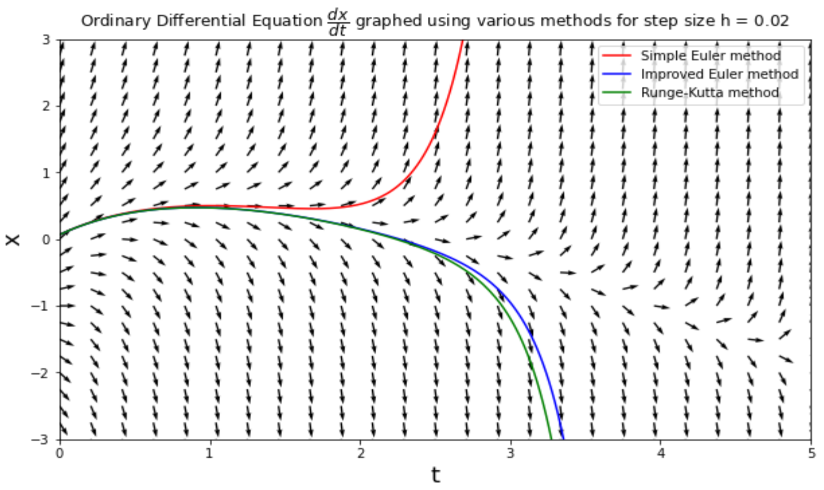


Figure 5 above shows the simple Euler, Improved Euler, and Runge-Kutta solution curves with step size h = 0.02, plotted along with the direction field for the differential equation.

As expected we can see a very slight improvement for the solution curve from the simple Euler and Runge-Kutta methods when lowering the step size. However the most obvious and notable difference comes from the improved Euler solution curve. The threshold was crossed for the solution to tend towards -∞ from the same reasons as explained in the task 2. The improved Euler follows the Runge-Kutta until $t \approx 2.2$, the error for the Runge-Kutta method is given by the magnitude of the 5th derivative of the solution of the initial value problem, and the error given for the improved Euler is given by Taylor's theorem to the differential equation, both of which we will not define; but to note of course, the defined error for the Runge-Kutta is less than the one for the Improved Euler, but marginally. Let us see what we get if we reduce the step size once more to 0.01.

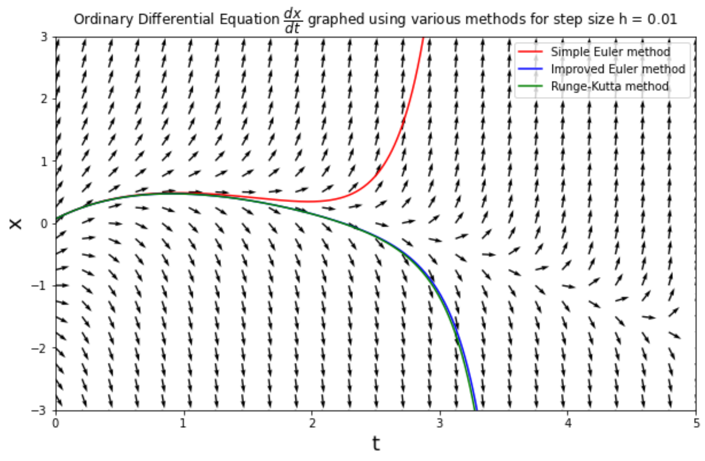


Figure 6 above shows the simple Euler, Improved Euler, and Runge-Kutta solution curves with step size h = 0.01, plotted along with the direction field for the differential equation.

Now we can see how the solution curve for the improved Euler curve becomes more accurate after reducing the step size and now is almost identical the Runge-Kutta solution curve. We also notice how regardless we reduce the step size, the solution curve from the simple Euler method does not improve; largely revealing its lack of accuracy.

<u>Conclusion:</u>

Different numerical methods for solving a first order differential equations were identified and carried out for the differential equation given by $\frac{dx}{dt} = (1+t)x + 1 - 3t + t^2$. The difference in accuracy between the different integration schemes were apparent to see when plotting the solution curves with the direction field, and how it varies with step size. The Runge-Kutta method was found to be the most accurate as expected, with the simple Euler method being very crude and inaccurate.

References:

- *Calculus to Chaos by D Acheson*
- *Lecture Notes*
- https://math.libretexts.org/
- https://www.geeksforgeeks.org/runge-kutta-4th-order-method-solve-differential-equation/