Scientific Computing Report

Benjamin Le Coz
Email: go19111@bristol.ac.uk
Github: benlecoz

EMAT30008 Scientific Computing

Department of Engineering Mathematics, University of Bristol

November 16, 2022

1 Summary of the software

The aim of this section is to explain the methods used in the software, as well as its capabilities. Examples and results of it running will be provided to illustrate how to use it, as well as to show what results the user can expect. The section will explain each file within the project, and how they all piece together to built the software.

1.1 Solving Ordinary Differential Equations (ODEs)

The first aim of the report is to create a piece of software that is capable of solving ODEs, whether it be single or a system of equations, similar to the scipy package odeint. The algorithm is designed to solve the ODE, by generating a series of numerical estimates between $x(0)$ and $x(1)$, with timesteps no bigger than a specific Δt_{max} value.

First, the ODE solver calculates the amount of steps that it needs to take between the inputted t_0 and t_{end}. The number of steps is defined as the sum of these two time values, divided by the maximum size of the timestep mentioned previously. The solver then runs a single step between each t and $t + \Delta t_{max}$, with $t \in [t_0, t_{end}]$. In this algorithm, we will be using two different 1-step integration methods, the Euler and 4th order Runge Kutta methods.

To illustrate the results found when running this code, we solve two different ODEs, using both the Euler and RK4 methods. The first is a first order ODE $\dot{x} = x$, which we plot compared to its true solution: $x(t) = e^t$. The parameters are set as having an initial condition $x(0) = 1$, $t \in [0, 1]$, and $\Delta t_{max} = 0.01$. Since our ODE solver can also solve systems of ODE, we also run the code to solve the second order ODE $\ddot{x} = -x$ using both methods. In this case, the true solution corresponds to the two equations: $x(t) = cos(t) + sin(t)$ and $y(t) = cos(t) - sin(t)$. The initial conditions are defined as $x(0) = [1, 1]$, $t \in [0, 10]$, and $\Delta t_{max} = 0.01$.

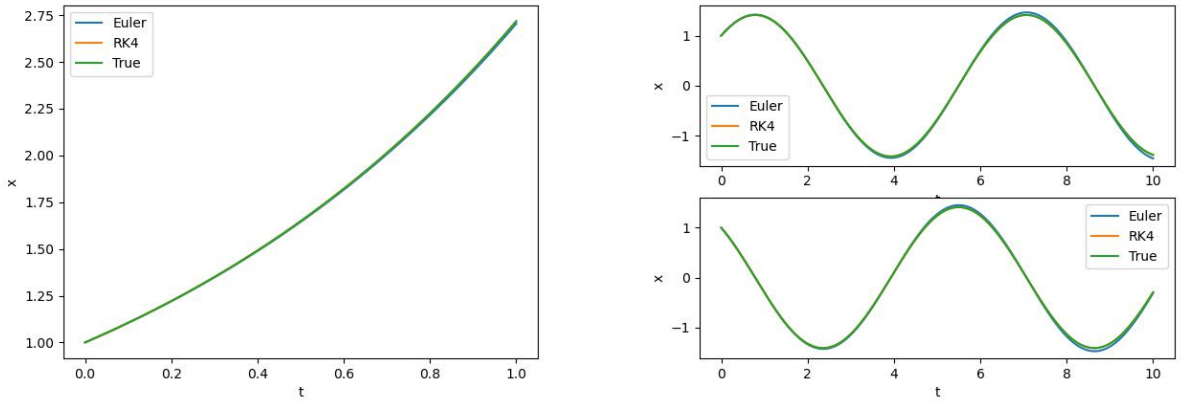

Figure 1: Plot of ODE Solver results for single ODE $\dot{x} = x$ (left), and system ODE $\ddot{x} = -x$ (right), using Euler and RK4 methods compared to true solutions.

Although it is very faint, it is possible to see a small part of the blue line on all three graphs, indicating that the Euler method results are slightly offset from the true solutions. The orange line

however cannot be distinguished, meaning that it is very accurate compared to the true solutions. This can be seen in the results of both integration methods found in Table 1 below.

	Euler Method	**4th Order Runge-Kutta**	**True Solution**
First Order ODE	2.7048	2.7183	2.7183
Second Order ODE	-1.4539	-1.3831	-1.3818

Table 1: ODE Solver results found at the final timestep, for different 1-step integration methods, compared to the true solution

1.2 Calculating the errors of the ODE Solver

Once the ODE Solver had been successfully coded, it was necessary to measure its performance versus the true solutions of the ODE, in a more precise manner than just looking at Figure 1. This is done using the ODE solver error plot code, which calculates the error between the Euler/RK4 approximations and the true solution, for various timesteps.

The error plot code displays the errors of the two integration methods when approximating a first order ODE. As seen in Figure 2, the RK4 method largely outperforms the Euler error. The error values are calculated for timesteps ranging from $\Delta t_{max} = 10^{-4}$ to $\Delta t_{max} = 1$.

To compare the efficiency of running each method, timesteps are randomly chosen that find very similar error values for both method. An example of this can be seen in the right hand graph of Figure 2, where the black dots denote the two timestep values that produce similar errors. In most cases

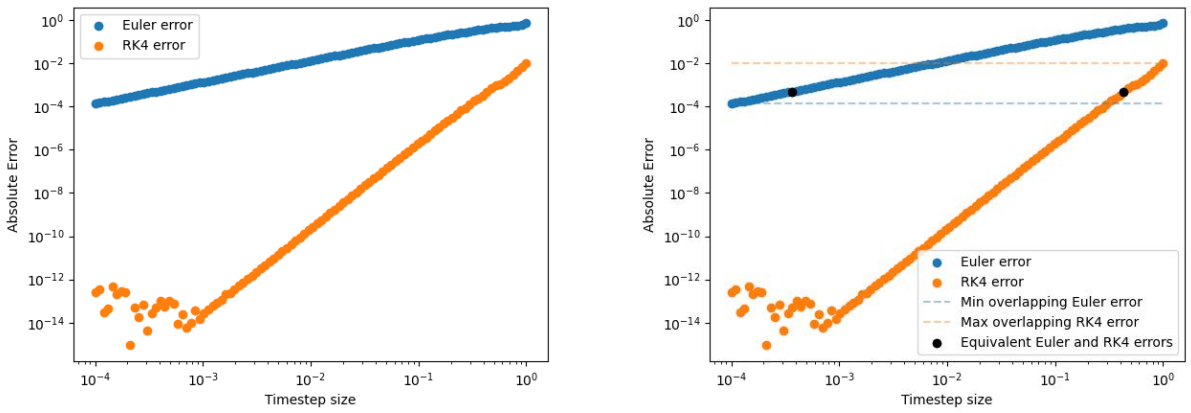

Figure 2: Scatter plot of the Euler and RK4 method errors when approximating $\dot{x} = x$, for different sized timesteps (left). Timesteps that produce similar Euler and RK4 errors are highlighted (right).

1.3 Shooting Root-Finding algorithm

The aim of shooting root-finding algorithm is to be able to isolated periodic orbits of any ODE, by solving boundary value problems.

Most of the ODEs that will be tested on this algorithm will lack any time dependency, similarly to the equations used in subsection 1.1. This means that solving the root-finding problem would mean solving two separate problems, involving both the unknown period and time shift. While the algorithm is capable of solving the root finding algorithm without the presence of a phase condition, the addition of this function will be instrumental in isolating periodic orbits for autonomous ODEs. Armed with this phase condition, the shooting algorithm then solves the boundary value problem to find the periodic orbit.

Putting the shooting algorithm into practice gives quite satisfactory results. First, we try to solve the predator-prey equations (expression of the exact equations can be found in Appendix A). The running of the root finding code allows us to plot the isolated periodic orbit of this function, as seen in Figure 3 below. We have also plotted the solutions to the two equations against each other, in order to visualise the periodicity of the solution found. This latter plot shows a closed ovoid shape, indicating that the root finding algorithm has successfully found an periodic orbit.

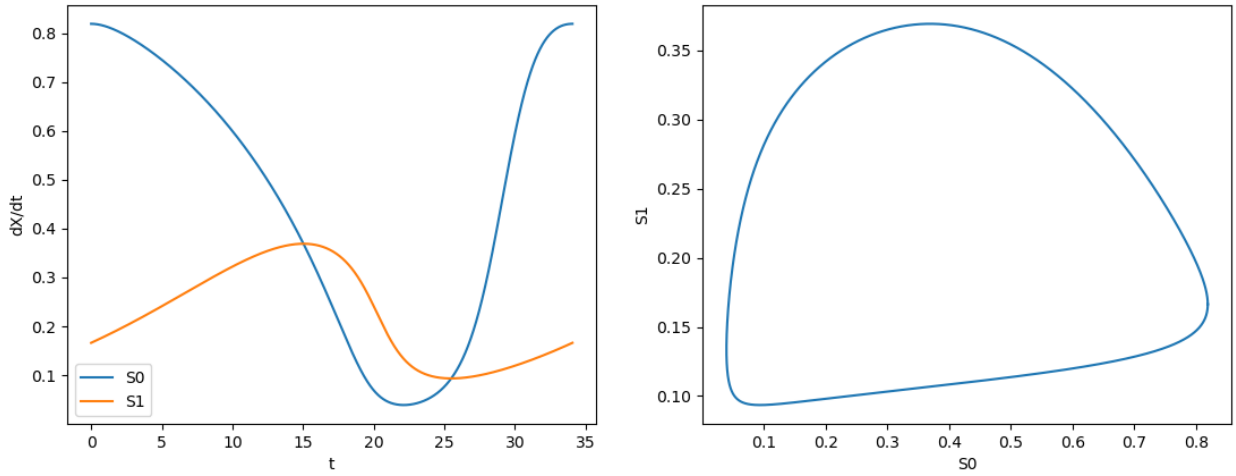

Figure 3: Plot of the isolated periodic orbit of the predator-prey function (left), and the two solutions plotted against each other (right).

The shooting algorithm can also solve ODEs that produce solutions of equal period. This is evidenced by the solving of the Hopf bifurcation normal form (expression of the exact equations can be found in Appendix A), as seen in Figure 4 below. The first plot illustrates the identical period of the two solutions, offset only by a couple timesteps. When plotting the two solutions against each other, a perfect circle can be seen, corroborating the identical period.

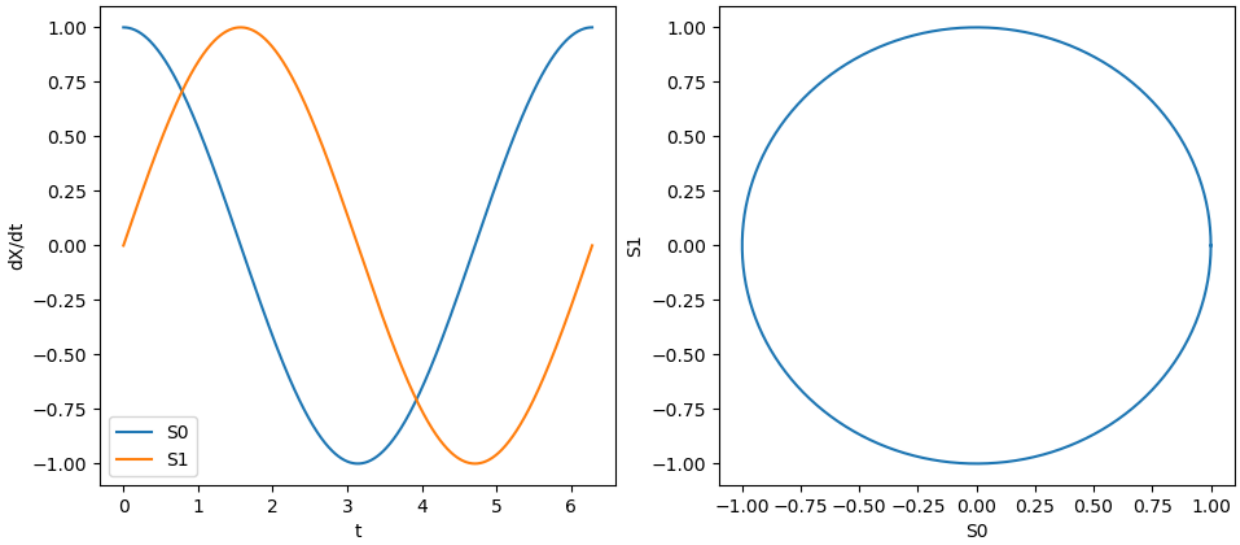

Figure 4: Plot of the isolated periodic orbit of the Hopf bifurcation normal form (left), and the two solutions plotted against each other (right).

1.4 Numerical Continuation

We have looked at ODE solving and boundary value problems so far, and now want to quantify parameter dependent behaviour found within problems. In order to achieve this, we develop two methods of numerical continuation: natural parameter and pseudo-arclength.

The code for both of these methods is relatively similar. For natural parameter continuation, the first solution is calculated using the shooting code defined in subsection 1.3, which takes a user inputted set of initial conditions as its first guess. The code then iteratively runs through a set of parameters, using the solution found at the previous parameter as its initial guess.

This method does have a quite glaring fault. The natural parameter code fails at folds, since there is not intersection between the search line and a nearby solution. This can be seen in Figure 5, where the natural parameter solution stagnate at certain values for both sets of equations. We therefore need to vary both the parameter values and the state vector when trying to solve the continuation problem. This additional value to solve for means that we need to add another equation to solve for, which is the pseudo arclength equation:

$$\delta\vec{u} \cdot (\vec{u} - \tilde{\vec{u}}) + \delta p \cdot (p - \tilde{p}) = 0 \tag{1}$$

This equation is incorporated into the code relevant to the pseudo arclength method. The initial guess for each parameter value is the predicted state, calculated using the previous two solutions. This method shows evident progress compared to the natural parameter method, as evidenced in Figure 5, as the pseudo arclength method manages to pass the folds. The pseudo arclength functions could also be run from one end of the parameter range to the other, but in order to visualise the natural parameter plots more clearly, the number of iterations of the pseudo arc length have been limited to 50.

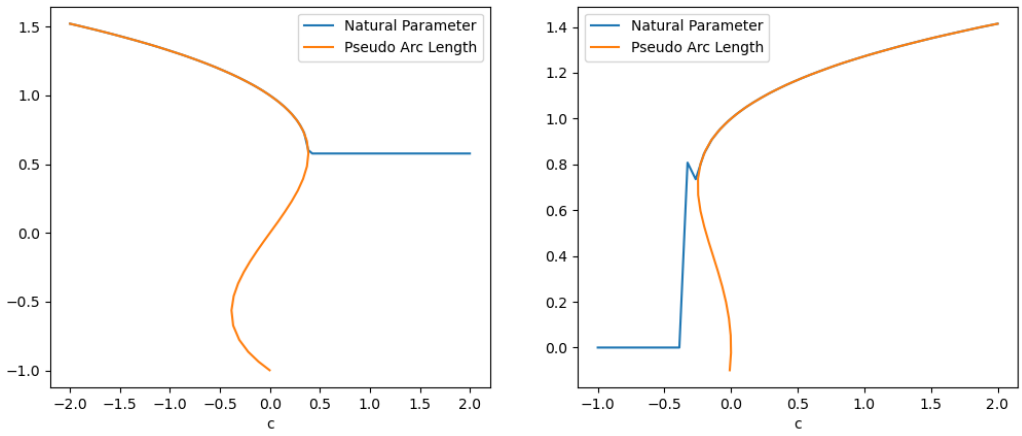

Figure 5: Plot of the Natural Parameter and Pseudo Arclength solutions, for a cubic equation (left) and the modified Hopf bifurcation equations (right).
These function are represented by Equation 4 and Equation 5 respectively, in Appendix A.

1.5 PDE Problems

The use of our PDE code is relatively straight forward. Two initial condition functions are defined: $sin(\pi x)/L$, which is the standard used initial condition for the rest of the code, as well as $sin^p(\pi x)$. The latter initial condition is used in order to understand the effect of this condition on the output of the function. To visualise this, we plot the PDEs while varying p.

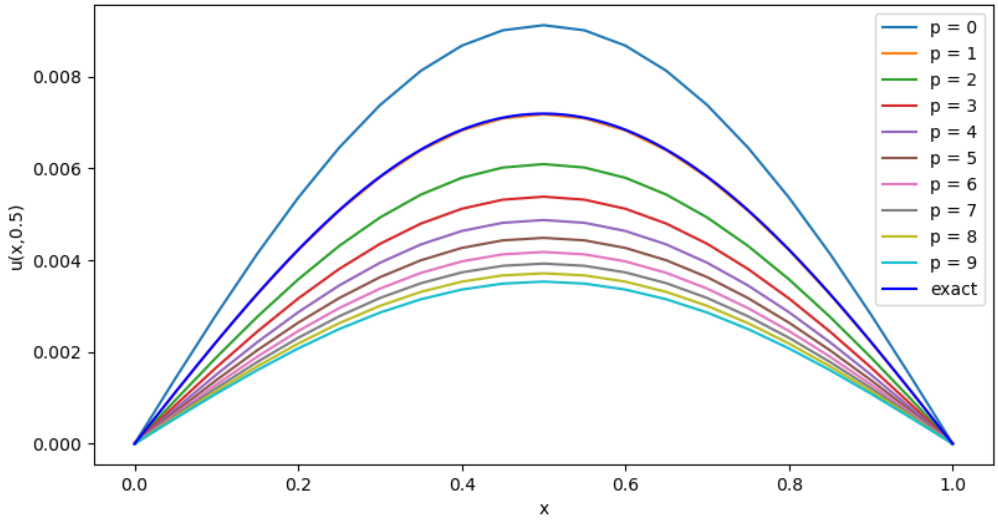

Figure 6: Plot of the PDEs, while varying p in initial condition $sin^p(\pi x)$

This shows us that the value of the initial condition has a large impact on the plotting of the PDE: when $p = 1$, the solution is almost perfectly true, yet the larger p gets the more it strays away from this exact solution.

The PDE code allows us to vary the method that we use. Up till now, the default method was Forward Euler, but the code allows us to also use Backwards Euler and Crank Nicholson. In order to get a comparative idea of the three methods, we plot them relative to the true solution.

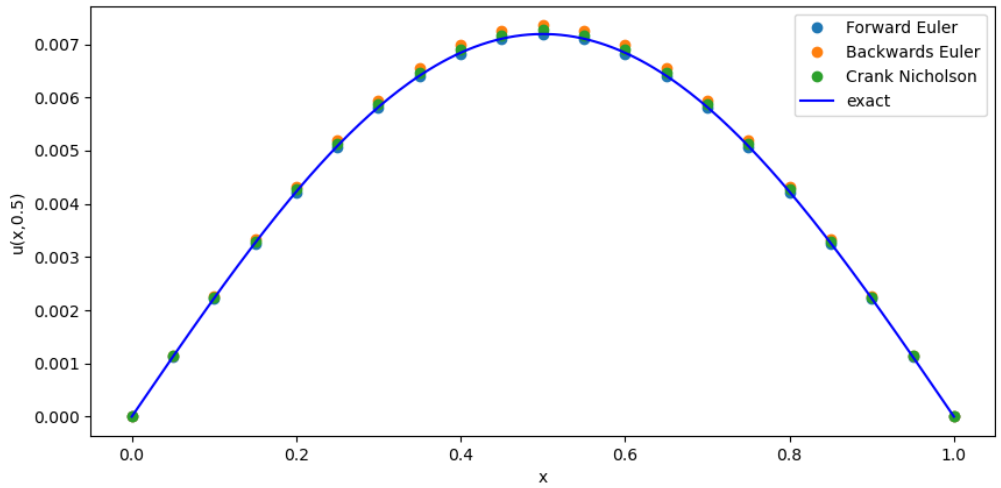

Figure 7: Plot of the PDEs, using the Forward Euler, Backwards Euler and Crank Nicholsom methods.

Finally, the code allows the user to define the boundary conditions, which can follow the Dirichlet, Neumann or periodic boundary conditions. However, it is unclear how effective the implementation of the latter two boundary conditions has been.

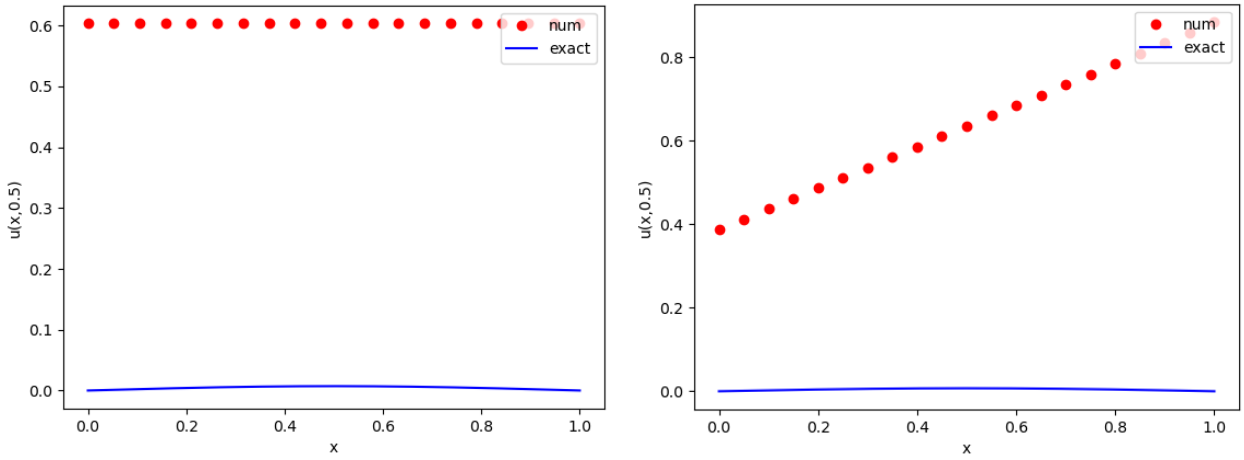

Figure 8: Plot of the PDEs, using the periodic (left) and Neumann (right) boundary conditions on the Forward Euler method.

2 Key Software designs

2.1 ODE Solver

One of the first decisions that was made when creating the ODE solver code was to not use the Numpy linear spacing function to create the timesteps. The initial idea for the code was to subtract the initial t_0 and final t_1 time values, and then choose the closest inferior value to Δt_{max} that would give a whole division of $t_1 - t_0$. Instead it was deemed that using the Δt_{max} as the difference between timesteps was the optimal way of operating. If this Δt_{max} does not provide a whole division of the time values, then the last timestep is equal to difference between the before last timestep and t_1. While this might add to the computing demands of the code, it was deemed to remain truest to the value of Δt_{max} inputted by the user.

Moreover, any function defined in this file returns a Numpy array as its output. This was done for multiple reasons, the first being that this helped significantly with handling the outputs. Indexing and manipulation is made much easier with the use of Numpy arrays, as it allows element-wise operations. Additionally, using this method allows for the code to run quicker, as they can be manipulated in a computationally quicker way than lists or other forms of storing multiple variables.

Another key decision in the ODE Solver code, and one which will be used in many other functions, is the recurrent use of the system boolean. In all of the programmed functions, the code needs to run equally well for a single ODE, or a system of ODEs. Yet, this proves challenging given the complex and often conflicting nature of both the inputs and outputs. In order to circumnavigate this, the system boolean is introduced and used extensively across the repository, This allows an easy and seamless distinction between the operations that need to be ran depending on the number of differential equations.

An example of this in use in the ODE solver is when defining the size of the solution array:

```
if system:
    X = np.zeros((number_steps + 1, len(x0)))
else:
    X = np.zeros((number_steps + 1, 1))
```

To determine the number of rows, the inbuilt length function is used to count the number of initial conditions x0 that the function requires, as this should match the number of equations (if there is no disparity between the two). Yet, if the ODE only holds one equation, then the initial conditions can be expressed as an integer. Hence, the system boolean is introduced to make sure that the correct dimension can be passed to the solution array.

One way around this would be to force the user to pass either lists or arrays for the initial conditions, but using the system boolean seemed a better alternative, providing less limitations on the user inputs.

2.2 ODE Solver error plots

The first decision that was made regarding this file was to determine whether or not this function needed to be isolated in its own file. While the plotting of the errors could have easily been done in the ODE solver file, the size of the code needed to do so, as well as the necessity to implement timing functions to the error calculations of the Euler and RK4 methods meant that creating a separate file made more sense. This was also not an addition that was essential to the ODE solver, it was merely an interesting addition to test the accuracy of our code. The decision was therefore made to separate it from the main ODE solver file, so as to not confuse the user with an incredibly long ODE solver file.

2.3 Numerical Shooting

The first key decision that was made for the numerical shooting code was choosing which solver to use. The choice made was to use the scipy.optimise function fsolve. This provided an efficient way of solving the root finding problem. The slight drawback with this function is that it sometimes does not converge efficiently towards a solution, which brings up different types of warning. The only addition to the code that was implemented in order to get around this inability to converge was to access the full output of the fsolve function. This raises an error (once the solver has stopped running) if the solver has not reached a converging solution, and so provides a lot of information to the user, but cannot handle a situation where the solver runs endlessly trying to converge.

Next, the layout of the shooting code played a key role in its functionality. Initially, the shooting code set up the initial conditions needed to be solved, and called fsolve to solve them within the same function. While this proved effective when running the shooting code by itself, some issues occurred when running the continuation code later on. This was due to the continuation code needing to call the shooting function, in order to solve it simultaneously to the pseudo arclength equation. Yet, if the shooting function was using fsolve to solve the root finding problem within itself, then the continuation code could not run properly. Once this was found out, the shooting code was changed accordingly: a separate function called shooting orbit now solves the shooting code, which itself is just a function of the phase and period conditions. The continuation code can therefore run as it calls the shooting function and not the shooting orbit one.

2.4 Numerical Continuation

The numerical continuation file is the most complex program in the repository, as it ties all the other code together. Hence, many of the most important decisions were made regarding this file.

Firstly, it was found quite early that the depending on what parameter the continuation code started running on, the code would have more or less success running. This makes sense in the case of the natural parameter continuation for example: at a certain parameter value, the code will fail to pass the fold. If the code is started on the wrong side, the program might not be able to identify where this fold is located. Hence, the use of the vary_par was instrumental. This allowed the user to decide which parameter value to start the code on. Once this had been implemented in the code, solutions were able to be found a lot easier. The slight negative point to this situation is that the code currently has no way of identifying if it started on the "wrong" side. It is up to the user to determine what they think the correct parameter to start on is. In the limited use of

the code in this repository, the single ODEs seemed to work best when started on the smallest parameter, while the system of ODEs seemed to work best in the opposite scenario. Given the very small sample size of this observation, this is not an instruction passed onto the user, as it is most likely not applicable to other ODEs.

Next, while testing out the algorithm, it was found that some variations of the continuation code would not work unless the solutions were rounded to a certain value. It remains unknown why this is the case, but it is hypothesised that there is an issue at some point with the handling of values with many significant values. When running the pseudo-arclength code on the modified Hopf bifurcation functions for example, the algorithm would perfectly compute the first 18 values out of 50, and then abruptly end at the 19th. This is the most blatant example that was found when creating and testing the code, but this error was persistent in most if not all of the code. Hence, in the case of the natural parameter continuation, each $n-1$ solution is rounded to 5 decimal places when used as an initial estimate to calculating solution n. Similarly, the pseudo-arclength continuation method rounds every solution to 3 decimal places when appending it the solution array, in order to be used as approximations to calculate the next solutions.

Finally, many different set ups were attempted in the pseudo-arclength code, as it was unclear for an extended period of time what the optimal coding layout was in order to consistently quantify parameter dependent behaviour. Initially, the decision was made to have two separate functions, which would each represent one of the equations to solve (these two equations are evidently the shooting and the pseudo equation). These two functions would be called and appended together inside of the fsolve brackets in order to be solved simultaneously. Yet, as plausible as this technique sounded, this did not work. Hence, a lambda function was used. This not only allowed solutions to be found, but also limited the number of exterior functions that the algorithm had to call when running, allowing the code to run quicker.

2.5 Code Testing

One of the most essential aspects of this coding project was to not only create code that could achieve one specific goal, but could be used in many other contexts and be able to handle different situations. A concrete example of this is generalising our code to not only run for specific ODEs that we want to solve, but for any ODE. In order to achieve this, it was necessary to develop arduous code testing, which would allow other users to implement our code while being guided through any mistakes that they might commit while using it. Two different types of code testing have been implemented in this repository: error traps and error trap testing.

The layout of the three main files has been set out so that only one function needs to be called for the code to run (that function then evidently calls others within the same file or across other files in order to achieve the designed task). The functions in the main files are: solve_ode in the ODE solver, shooting_orbit in the Numerical shooting code, and continuation in the Numerical continuation code. The process of developing error traps was simplified, as it was done by implementing them at the start of each main function. Two types of error traps were developed: ones that test the inputs of the main functions, and ones that test the outputs of the functions that the main function calls.

In this repository, it was deemed logical to organise and separate these error trap tests in terms of which file they were testing. Given the clear delimitations of the files (even if they do interact with each other), a test file was created for the following files: ODE_solver, numerical_shooting, numerical_continuation. These files use the "try: ... except:" loop to test out both the algorithms using wrong inputs, expecting errors to be raised.

In the case of the ODE solver and the numerical shooting code, test functions were also created which compare the results of these algorithms when compared to the true solutions. For the ODE solver, this function has been tested and proven to work, and has therefore been incorporated into the ODE solver file. On the other hand, the numerical shooting accuracy function was created too late to be tested properly, and so has not been fully implemented into the shooting code. It can instead be found inside the input and output test file, and would have been implemented into the shooting code had there been more time to do so.

2.6 PDE Problems

The main decision that was made when writing the PDE code was to separate the three methods into different functions. This meant that the code had less loops determining every condition applied to the PDE, and the user could just call the desired method with a desired boundary condition. When it came to these boundary conditions, the aim was to implement them for the Forward Euler method first, and then extend them to suit the other two methods once it was deemed that the implementation was successful for the Forward Euler. Yet, it was hard to tell when the boundary condition implementation had been successful. Hence, only the Forward Euler method has the additional boundary conditions.

3 Learning Log

Many different learning outcomes can be drawn from this coursework. Firstly, before any implementation of the methods was taken into account, there is an undeniable amount of knowledge gained pertaining to the understanding of the mathematical concepts involved. It sometimes however take quite a long time to grasp some these concepts, most notably with the continuation algorithm. In some cases, we had been exposed to these concepts in the scope of other courses. Yet, the main difference that I highlighted was that instead of just being able to solve these problems, having to code problem solvers for them meant that there was a necessity to understand more of the underlying concepts that drove how they worked.

In previous AI or MDM projects, the use of Github repositories was often necessary and also monitored. Yet, given the recurrent reminders on the clean and up to date code, there was an added emphasis on the upkeep of the repository. This meant that regular commits with detailed changes to the code were instilled into my coding routine. While this a repository that I was the only one accessing and contributing to, the practice of getting used to regular and effective repository management was greatly beneficial to my software engineering abilities.

Some of the more short-term implications of what I've learnt are to do with the mathematical concepts learnt. While I don't doubt that many fields use Ordinary and Partial Differential Equa-

tions very often, it is not something that I see myself using for the entirety of my educational and professional career, hence the more short term benefit of understanding these concepts. Similarly, one of the main short term benefits of this coursework has been the smaller coding techniques that I have picked up. One example of this is the following code structure:

```
if __name__ == "__main__"
```

I had seen this code structure in many files and repositories recently, but could never understand what it did or why people used it. Once I finally understood, I did find it was incredibly effective at limiting the output of each file when it was called within another file. Given the overlapping nature of the many files in this repository, this allowed me to control the outputs of each file when being called, and this structure was successfully implemented in almost every file created.

In addition to these more minor coding practices, many of the techniques that I have picked up will benefit me in the long term. The most notable one for me is the implementation of code testing. Many times in recent projects or coding assignments, a task has been clearly laid out for us to achieve. Once this goal has been reached, the code does not deviate, and is not expanded upon, past this one use. Thanks to this coursework though, the idea of code testing, and mainly the concept of generalising and explaining the code in a way that any exterior user can comprehend and use it, has greatly impacted me. Setting up error traps within my functions was not something I was accustomed, but greatly helped in testing my code to see how it would react to erroneous inputs. In many cases while I was creating the code myself, errors would come up, and deciphering the exact issue at hand would take me an extended period of time. Laying out error traps meant that if other users commit the same mistakes that I have, they would be met with an error message that effectively guided them towards a solution.

One of the final main lessons that I can take out of this coursework is to not blindly rush into trying to code. Many times, I would try and code the solution before I even truly understood what the solution was supposed to represent. This is even more applicable to concepts that are not initially easy to understand, which was quite often the case here. The problems that I was confronted with were not intuitively easy to comprehend, and took quite a lot of mathematical brainstorming before a coding solution could even begin to be formulated. The fact that I did not exactly follow this way of thinking at the beginning of the coursework meant that I slightly misused a significant amount of time at the beginning of the semester. In this period, while I did get some parts of the code working, this was without really understanding the concepts that the code was based on.

Finally, one aspect of coding that I feel I still need to improve is the timing of the code pushing. While I do feel that the commits that I was making were regular and done in a timely manner, I struggled with the pushing aspect a bit more. It was explained to us that pushes should only be done once a large part of the code has been completed, which was fine at the beginning because I was working on each separate file on a week to week basis. Yet towards the end, improvement were being made across almost all of the files, meaning that it was a lot less easy to tell when it was necessary to push. The decisions to push were therefore a lot more arbitrary.

Possible areas of improvements are mainly linked to the PDE section. Given the amount of time spent on numerical shooting and continuation, as well as extenuating circumstances, I had less time

to focus on the PDE section. Given more time, Neumann and periodic boundary conditions could have been implemented to the Backwards Euler and Crank Nicholson methods, as well as being successfully implemented in the Forwards Euler method. In terms of the ODE code, more input and output tests could have been added to the continuation code. Moreover, it would have been interesting to pursue a way of signalling to the user that their continuation code is not finding a solution, instead of needing manual Interrupt to end the code. Other than that, a lot of the avenues of improvement have been pursued to a suitable degree, in the humble opinion of the report writer.

A Additional Equations

Predator-prey equations

$$\begin{aligned} \frac{\mathrm{d}x}{\mathrm{d}t} &= x(1-x) - \frac{axy}{d+x}, \\ \frac{\mathrm{d}y}{\mathrm{d}t} &= by(1-\frac{y}{x}) \end{aligned} \tag{2}$$

When running the shooting algorithm, the parameters are set as $a = 1$, $b = 0.1$ and $d = 0.1$.

Hopf bifurcation normal form equations

$$\begin{aligned} \frac{\mathrm{d}u_1}{\mathrm{d}t} &= \beta u_1 - u_2 + \sigma u_1(u_1^2 + u_2^2), \\ \frac{\mathrm{d}u_2}{\mathrm{d}t} &= u_1 + \beta u_2 + \sigma u_2(u_1^2 + u_2^2) \end{aligned} \tag{3}$$

When running the shooting algorithm, the parameters are set as $\beta = 1$ and $\sigma = -1$.

Cubic equation

$$x^3 - x + c = 0 \tag{4}$$

When running the continuation code, the parameter varied is c.

Modified Hopf bifurcation normal form equations

$$\begin{aligned} \frac{\mathrm{d}u_1}{\mathrm{d}t} &= \beta u_1 - u_2 + u_1(u_1^2 + u_2^2) - u_1(u_1^2 + u_2^2)^2, \\ \frac{\mathrm{d}u_2}{\mathrm{d}t} &= u_1 + \beta u_2 + u_2(u_1^2 + u_2^2) - u_2(u_1^2 + u_2^2)^2 \end{aligned} \tag{5}$$

When running the continuation algorithm, the parameter varied is β.