1 Summary

The purpose of this software is to provide the user with a suite of different approaches to Boundary Value Problems (BVP's). BVP's are essentially Ordinary Differential Equations which boundary conditions imposed as ranges. So there can be many different solutions to these problems with a variety of different initial conditions[3]. These ODE's are essential to any attempt to model the real world with mathematics, as ODE's describe literally everything that changes with respect to another parameter, for instance, time.

The example this software was trained on was the Duffing equation, used to model damped and driven oscillators [4]. The code has been split up into many different functions in order to make the code better constructed. The first function is the Duffing equation itself, however, it has been converted to First Order Form. This is because it contains a second derivative $\frac{d^2u}{dt^2}$ which must be translated into a first derivative, creating a system of equations instead.

$$\begin{gathered}\ddot{u} + 2\zeta\dot{u} + u + u^3 = \Gamma\sin(\omega t)\\ \text{becomes}\\ \dot{u_1} = u_2 \text{ and } \dot{u_2} = \Gamma\sin(\omega t) - 2\zeta x_2 - x1 - x1^3\end{gathered}$$

1.1 Shooting

The Duffing equation only serves as an example, and any equation could be substituted into its place provided it's in First Order Form.

From there, the equation can be converted into a *zeroproblem()* where all the components of the equation are integrated using *scipy.integrate.odeint()*, then all moved to the same side of the equation to give something of the form $x - y = 0$ so that this equation can be solved for x in a process called Numerical Shooting. This is implemented in the *shooting()* function. Here, the code searches for a periodic orbit of the ODE, using the *zeroproblem()* and, if it is found, returns the starting conditions for which there is a periodic orbit.

1.2 Collocation

The next section of the code focuses on Chebyshev approximation[2]. The Stone Weierstrass theorem states that every continuous function can be approximated by a polynomial [1]. This means that the function we are attempting to perform numerical methods on can be approximated by a function that is much simpler to integrate. Chebyshev noted this and derived a method that allows us to utilise specific points that, if we evaluate our function at, can be used to approximate a solution to the equation.

To do this we use the *cheb()* function in the code, which returns the Chebyshev matrix, D, as well as the Chebyshev points, t. Since neither D nor t depend on anything they are easy to check as these matrices are known for various dimensions. This means that we can write tests to check their performance. This is an essential part of any code but especially numerical methods as it can often be hard to tell where you code may be going wrong from just a graph, for instance.

The collocation code is within the *run_cheb()* function. This function computes D and t, then reshapes the y values that are passed in, converts them to the correct size, then converts the equation to a form that can be solved by an inbuilt solver. Generally equation takes the form:

$$\underline{\underline{D}} \cdot \underline{\mathbf{x}} - \underline{\mathbf{f}} = 0 \qquad (1)$$

1.3 Continuation

The final function of this software is the *Results()* function[5]. This function governs all the rest. It allows the user to select an arbitrary ODE (provided it is in first order form) and a set of parameters, and then it outputs a plot of the behaviour of the ODE as you vary its parameters. It offers you a choice of approaches here as, for simple enough equations that need not be discretised, it will simply solve them and extract the maximum value of the solved ODE and plot them against the values of the varied parameter(Figures 1 and 2).

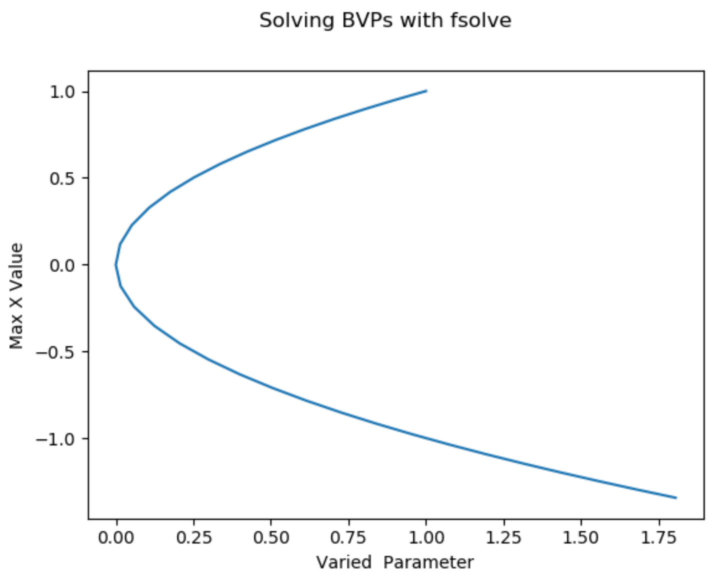

Figure 1: A Graph to show the effects on the maximum value of y while varying p in a simple quadratic equation $x^2 - p$

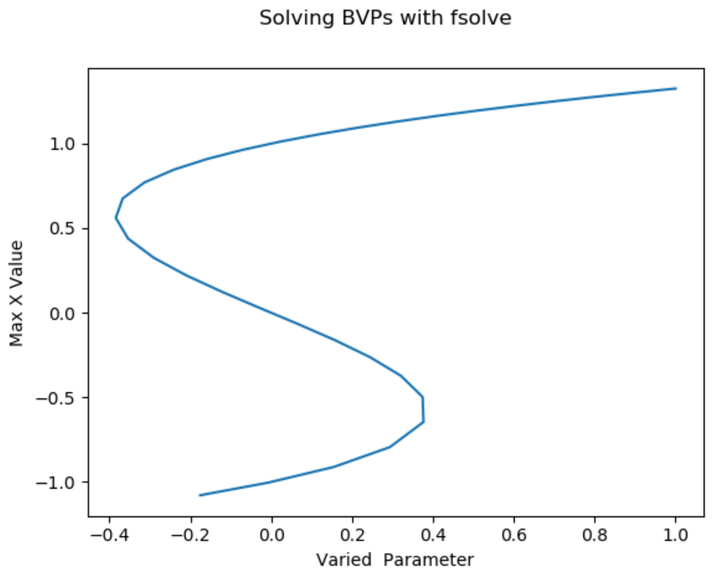

Figure 2: A Graph to show the effects on the maximum value of y while varying p in a simple cubic equation $x^3 - x - p$

However, for complex equations like the Duffing equation you can implement Collocation in order to discretise the ODE, and then investigate its behaviour as above. The results of this can be seen in Figure 3

2 Design

This is a complex piece of code that involved a great many design decisions that, while they work, only impact the style, efficiency and potential length of the code. The first one to come up was which method to use in order to integrate an ODE. The most simple methods for this are Euler and Runge-Kutta. I coded up both of these, however there seemed to be no advantage to using these methods as they are not the most accurate approximations of integrals, and may contain bugs that I failed to spot. This lead me to use *odeint()* to integrate them. It is efficient and accurate, as well as being much more robust than a method that I could code and test in the timeframe allowed.

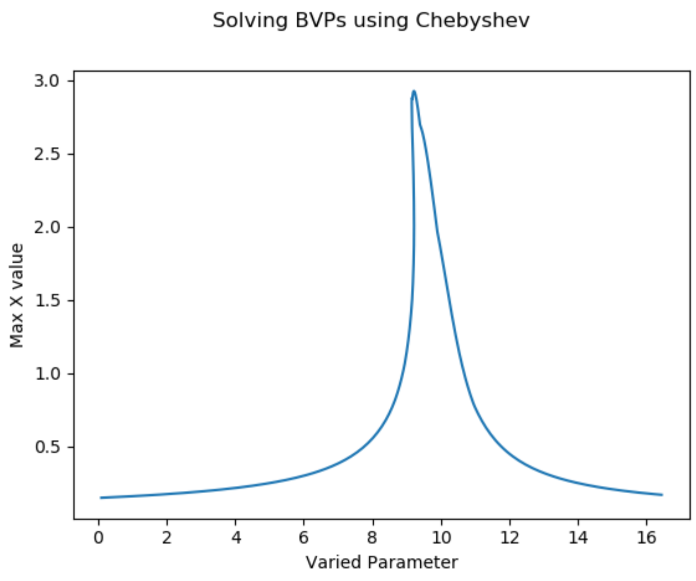

Figure 3: A Graph to show the effects on the maximum value of y while varying k in the Duffing Equation

It lies within the scipy.integrate library which I imported at the top. Since this code is all built upon the layers at the beginning, I created it as one long file, with the imports at the top, which firstly, is more efficient to do this only once, but secondly, could very easily be changed and split into separate files due to the compartmentalised nature of the code.

The next, but similar decision was which root finder to use initially. Whilst it would be possible for me to code my own version of the Newton-Raphson iteration, again it would not be up to the same standard as some of the inbuilt root finders. *fsolve()* is the most commonly used solver and worked quickly and easily to solve the equations that were passed to it. It was definitely a good place to start from but other root finders, such as *brentq()* were also tempting options for their lack of need for an initial guess for a solution.

The next decisions that needed to be made were for computing the Chebyshev points to be used during collocation. The Matlab code provided was an excellent place to start from. However, due to the ease with which Matlab manipulates matrices, some tweaks were needed to recreate this method in python. Firstly we needed to catch any division by zero errors that would be caused if the user input $N = 0$. Then, in order to most closely imitate the Matlab code, *repmat()* was used instead of *tile()*, *hstack()* or *vstack()*. There were several issues with matrices such as I was forced to use *tensordot()* in place of the standard multiplication character because it would try to return a 1 dimensional array instead of an $N + 1 \times N + 1$ matrix.

Since the Chebyshev method was unfamiliar to me I decided it would be worth testing its outputs. I ran the tests provided until it passed them then wrote some additional ones to compare it to the Matlab version under different conditions, such as multiplying it by various functions such as sin and cos and checking that the output resembled the appropriate function.

I suspect that there is a more efficient method for creating the Chebyshev matrix using more vector notation as in Matlab however, I did not have time to refine every aspect of the code so I left this section to run a touch inefficiently in order to focus on getting the rest to run at all.

The most challenging section of this software to create was the collocation code. As defined above in Equation 1, the script had to take an input function, of any dimension and then convert it into a form which can be passed to a root finder. in order to do this, I found the dimensions of the ODE with

```
dim = int(round(size(x) / (N+1)))
```

This allowed me to reshape the vector into the correct dimensions to be passed to an arbitrary ODE without having a clumsy *dim* parameter needing to be passed into the function by the user.

I ran several tests on this block of code based on *runtests_ode()* however, these tests only worked properly for a 1D ODE so I ran the Matlab code equivalent and used the values it created to test my Python version.

The final challenge was to implement numerical continuation. This is conceptually simple to do as all it requires us to do is solve an input equation for a set of parameters that changes. This helps us observe the behaviour of a complex system under varying conditions by extracting a known point, here, the max y value of the function.

The simple version of this involves none of the discretisations coded earlier so this was the one I coded first. It was very simple to imitate the Matlab approach while making improvements to it as we went. The Matlab version was (deliberately) poorly written and missed out simple features that make it much more usable and versatile, but not hugely more efficient. I created a function that would be capable of taking inputs:

```
ODE - the ode to solve
U0 - initial guess for the solution
pars - the additional parameters for your ode
vary_par - the index of the paramter you wish to vary
max_steps - the number of points you would like for your continuation
```

This approach uses *fsolve()* for the reasons mentioned above, but differs from the Matlab code in that the provided code creates several empty lists which it then proceeds to populate. There is a very slight benefit in terms of efficiency in doing this, however the *extend()* method is more efficient than both. However, as stated above, this code has not been optimised for efficiency.

The second method of continuation is using the Collocation method outlined in the Section 1.2. I decided to make a function that would plot the outputs of these different methods for continuation because otherwise I would have repeated the same block of code with a different figure title each time so I made the method an input to the *plotter()* and added into each block in an attempt to make the code more resemble a finite state machine for simplicity.

3 Learning Log

This project taught me a great deal. I learned about mathematics that was alien to me that now I feel I have a firm grasp on, namely Boundary Value Problems (BVP's). The methods used to solve them are impressive algorithms that were challenging to implement. I learned about the differences between the various different integrators in terms of accuracy. I struggled mostly with the collocation section, more specifically in the manipulation of vectors and matrices in Python. I suspect this is due to the fact that I'm used to Matlab which is quite gentle with users when they try to do matrix manipulation and will generally have little issue with the shape or size or operator being used. Python seemed to be slightly more rigid with its handling of matrices.

I have learned about the benefits of git. I started off finding it to be slightly tedious and like an added complication that I did not understand and did not see a need for but after three weeks of editing code, parts of it stopped working and it was a great help to find that I could return to the version I had created days ago with minimal effort. I have since begun using it for a variety of different group projects in MDM and Machine Learning.

I have also practiced my abilities in Python which had grown stale since I last used them halfway through first year. At the time I never learned how to perform any complex numerical methods, just game development and basic usages. This was a much more in depth test of my abilities that I was slightly familiar with in theory from Numerical Methods in Matlab, however, this went much deeper. Although, it has revealed that there is an absolutely vast amount of Python that I do not come close to understanding at this point in time but I hope that this has given me some of the tools I will need to begin to get to grips with the wider language. This should be useful for Machine Learning and MDM as both can involve modeling in Python as well as hopefully in later life depending on what I end up doing professionally and for my own enjoyment. I have always been tempted to figure out how the world around me works and it would be incredibly enjoyable to be able to perform some calculations on say an SIR model at some point if I were to have the time.

I began this unit with a sound work ethic and kept up to date with the worksheets until the collocation week. From there I struggled to make the code to cooperate and began asking Oliver for his help on the Friday lab sessions. This was immensely helpful and I cannot thank him enough. However, I was slightly too proud with my problems. I saw some other members of the cohort leap ahead of the stage I was at because they had reached out and asked for help from eachother and the lecturers. This was quite likely a very good idea. I spoke to members of the cohort frequently and they were immensely helpful for me and identified several errors in my code. Eventually, a lot of these problems became clearer as I talked about them with my peers but this could have been expedited by asking the support staff. However, I recognise that there is a line beyond which the staff cannot help for practical and fairness reasons as well as I should solve my own problems. The help I recieved from all parties was tremendous but I think I would have asked for more in order to create the highest quality code.

I think a lot of my issues stemmed from not understanding exactly what I was trying to accomplish. As in, with the Matlab code infront of me I would be able to 'translate' it and get slightly close to something that worked but, since I didn't know exactly what the Matlab was doing, I didn't know how to combat the errors that Python was throwing at me. The ways I think I will try to deal with this phenomenon in future is

1. Research the problem and understand what is happening in the background to an immense level of detail, focusing on shapes and dimensions
2. If there is code provided as a hint, check the outputs of every section

All in all this has been a challenging unit. The overview of the maths involved is not too complex, for instance, shooting is a simple enough concept, and coding it is only slightly more concept, whereas continuation is similarly simple but much more tricky to code correctly while collocation is complicated whichever way I look at it. There are always challenges and these have been tough ones but that only makes it the more satisfying to overcome them and learn from them.

References

[1] Louis De Branges. The stone-weierstrass theorem. *Proceedings of the American Mathematical Society*, 10(5):822–824, 1959.

[2] Lloyd N Trefethen. *Spectral methods in MATLAB*. SIAM, 2000.

[3] Wikipedia. Boundary value problem — wikipedia, the free encyclopedia, 2017. [Online; accessed 17-November-2017].

[4] Wikipedia. Duffing equation — wikipedia, the free encyclopedia, 2017. [Online; accessed 15-November-2017].

[5] Wikipedia. Numerical continuation — wikipedia, the free encyclopedia, 2017. [Online; accessed 17-November-2017].