EMAT 30008 – Scientific Computing Software

Ryu Bromley
Email: ua19949@bristol.ac.uk

11th May 2022

INTRODUCTION

This report outlines the summary of the software developed as part of EMAT30008 – Scientific Computing. Within this summary, each file's purpose, format, and usage are briefly explained. Additionally, the decisions made throughout the development of this software are outlined with explanations detailing specific design choices where relevant. Finally, the reflective learning log discusses and analyses the process that was undertaken to develop the software.

1. SOFTWARE SUMMARY

1.1 Initial Value Problems

The file integrate_ode.py implements the Euler and Runge-Kutta 4th order (RK4) methods to solve an ODE or system of ODEs with specified initial conditions. Each step of the process has a respective function to carry out the required processes:

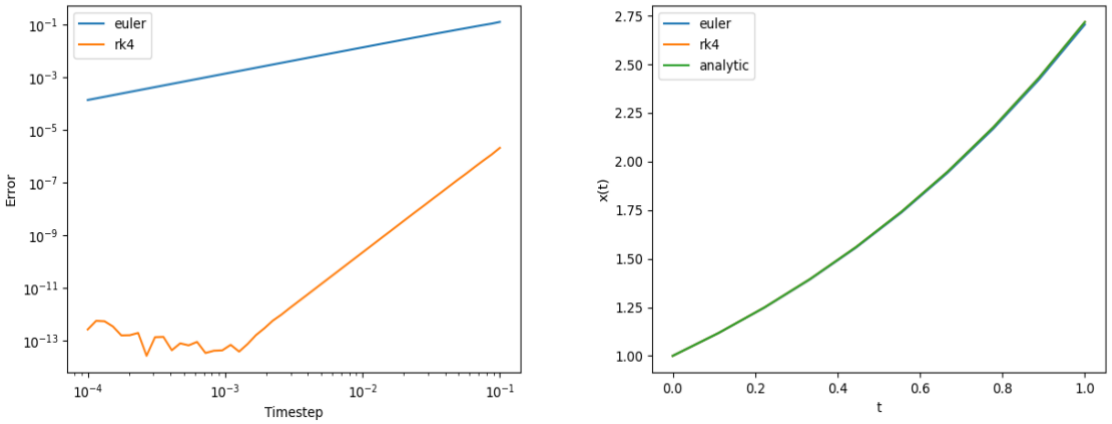
- euler_step – calculates a single step of the Euler method and returns the resulting x and t values
- rk4_step – calculates a single step of the RK4 method and returns the resulting x and t values
- solve_to – calls the required 1-step integration method (euler_step or rk4_step) and solves between x1, t1 and x2, t2 ensuring each step does not exceed dt_max

- solve_ode – calls solve_to for each time step and returns an array of numerical solution estimates to the user.

To run the program, a user must specify:

- f – the required ODE to be solved as a function
- x0 – the initial conditions of the ODE as a float or array
- t – the time frame for the ODE to be solved as an array
- dt_max – the maximum step size to be implemented as a float

The file euler_rk4_plot.py calculates the error of the Euler and RK4 methods by finding the difference between the generated solution estimates and the analytical solution. For the case of the first simulated simple ODE, these errors are plotted for both methods as seen in the figure below

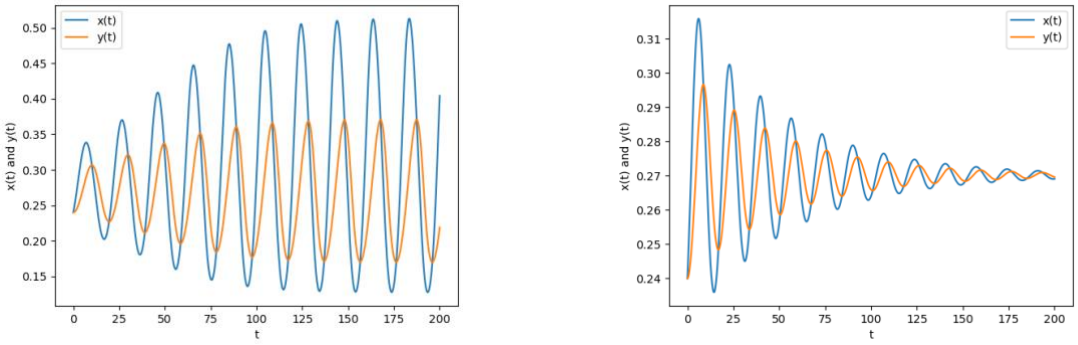


From this graph, it is clear that RK4 produces more accurate solutions within the software compared to Euler. The RK4 error behaviour between 10-4 and 10-3 can be attributed to rounding errors, with some slight anomalous behaviour around 5.5 x 10-4. After roughly 10-3, the RK4 error behaviour is due to truncation errors.

The generated solution estimates for the ODE case above are plotted against the analytic solution, in the figure (right). This graph shows that the methods implemented by integrate_ode.py perform correctly due to the proximity of each curve to one another. It is worth noting that there is a miniscule difference in solutions at the final time 1.0, indicating a slight discrepancy in convergence between the Euler and RK4 methods.

1.2 Numerical Shooting

The function predator_prey in ODE_functions simulates the Lokta-Volterra (Predator-Prey) equations in code form. A time-series plot of the Lokta-Volterra x and y equations is produced when running predator_prey.py, to analyse the behaviour of these equations below and above the point b = 0.26. These time-series plots can be seen below



Time-series plots of Lokta-Volterra equations for b<0.26 (left) and b>0.26 (right)

From these plots, it is clear that the equations demonstrate a divergence from the starting point to a uniform frequency for b<0.26. By contrast, the equations demonstrate convergence as time increases for b>0.26. Through visual inspection, the time period for these equations is seen to be roughly 22 seconds, with initial conditions of roughly 0.2 and 0.23 for x and y.

The file numerical_shooting.py implements the shooting root-finding problem for arbitrary ODEs of arbitrary dimensions. This file is modularised with functions:

- orbit_calc – solves the root-finding problem with SciPy's fsolve function and calls the shooting function 'num_shoot' within the fsolve execution to solve root-finding problem
- num_shoot – calculates the periodic boundary value problem by finding the difference between the initial conditions and the result of the numerical integrator F, simulating the equation

$$\mathbf{G}(\mathbf{u}_0, T) = \begin{bmatrix} \mathbf{u}_0 - \mathbf{F}(\mathbf{u}_0, T) \\ \phi(u_0) \end{bmatrix}$$

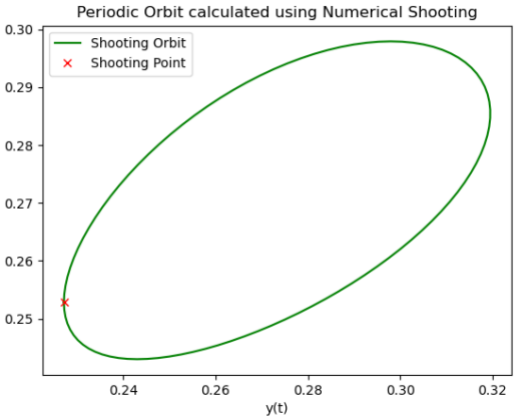
- conditions – implements the phase condition to enable shooting of autonomous ODEs.

To run the program, a user must specify:

- f – the required system of ODEs to be solved as a function
- u0 – an estimate for the shooting problem, passed as an array in the format [x, y, T] where x and y are the estimated points and T is the estimated time period

- var – the index of the equation within f required to return the phase condition for shooting, passed as an integer

Using the initial conditions and time period found from the time-series plots in predator_prey.py, numerical shooting is carried out to find the same aforementioned period, as seen below



1.3 Code Testing

Docstrings are implemented throughout the software with the purpose of defining and documenting the key functions used in the various scientific algorithms. These docstrings define a function's input parameters based on their variable type and usage, as well as the variables returned by the function.

The file shooting_tests.py verifies the results from numerical_shooting.py against known, explicit solutions. This is carried out for the Hopf bifurcation normal form and an extended, 3-dimensional variation of this bifurcation. Tests for the time period, initial points and final orbit points are computed using the functions:

- hopf_test – calls the shooting function orbit_calc and tests calculated solutions against the explicit solutions
- du3_shoot_test – performs identically to hopf_test with an additional initial condition to test the 3D Hopf Bifurcation

Input checks are used throughout the software to ensure that any passed arguments are in the correct variable type, preventing errors from occurring during execution. These checks are also implemented to ensure that a user selects a valid method for a given algorithm.

1.4 Parameter Continuation

The file numerical_continuation.py implements numerical shooting to capture parameter dependent behaviour for a given system of ODEs. This process is computed using two functions:

- natural_continuation – solves the root finding problem using SciPy's fsolve at each point in a parameter range,
- plot_bifurcation – plots each tested parameter value against its continuation result, demonstrating the bifurcation behaviour for a given function.

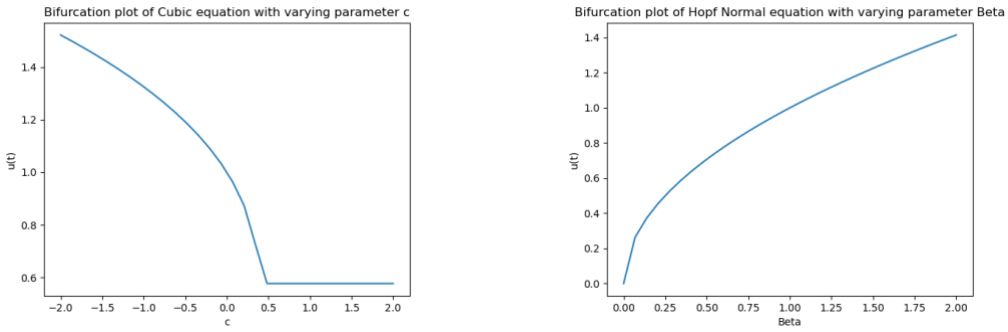
To run natural_continuation successfully, the user must specify:

- f – the function containing the parameter to be varied
- u0 – the initial conditions of continuation for the function f, of the start points and period (where relevant for a shooting discretisation)
- range – the range of values to be tested for the parameter
- space – the increment size for each iteration of continuation
- discretisation – specifies the solver to be used for each iteration as a lambda function
- System – a string of the function f's name for automation in plot_bifurcation
- Parameter – a string of the function's parameter to be varied for automation in plot_bifurcation

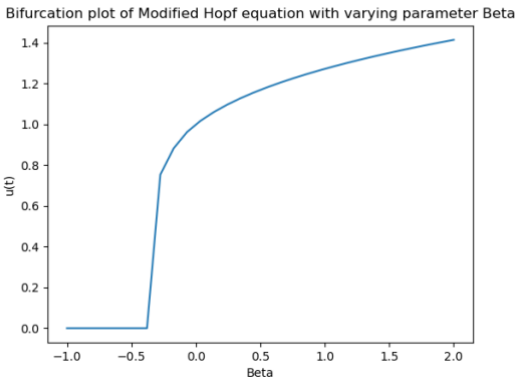
Three natural parameter continuation examples can be found in the file continuation_test.py:

- cubic_natural – varies the parameter 'c' in the equation x^3 – x + c = 0 between -2 and 2 and initial conditions of u0 = 1.5 with a space increment of 30
- hopf_norm – varies the 'Beta' parameter in the Hopf bifurcation normal form between 0 and initial conditions of u0 = 0.1, 0.5 and T =6 with a space increment of 30
- hopf_mod – extends the hopf_norm example for the modified Hopf bifurcation normal form for 'Beta' between -1 and 2, starting at 2 with the same initial parameters

The resulting bifurcation diagram for each example can be seen below



Left: Cubic equation bifurcation plot with varying parameter 'c' between -2 and 2, a fold can be seen at c ≈ 0.5. Right: Hopf normal form bifurcation plot with varying parameter 'Beta' between 0 and 2, a fold can be seen at Beta ≈ 0



Modified Hopf normal form bifurcation plot with varying parameter 'Beta' between -1 and 2. A fold can be seen at Beta ≈ -0.35

1.5 Finite Differences (PDEs)

The file PDE_solver.py utilises three different finite difference schemes, Forward Euler, Backward Euler and Crank-Nicholson, to find solutions of PDEs for a specified domain in x and t. The file is modularised using ten functions:

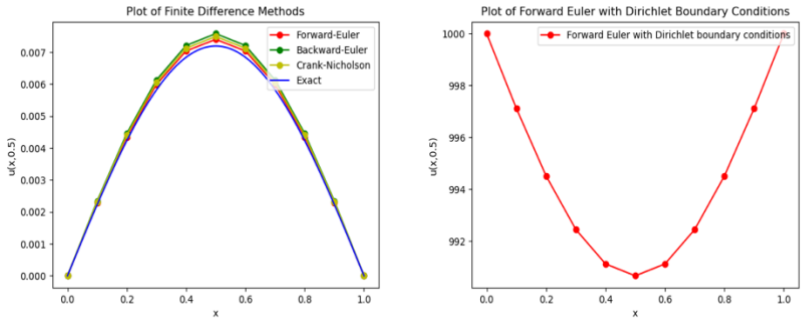
- calc_PDE – called by the user to solve a PDE and returns the solutions
- init_params – initialises the numerical parameters for the difference schemes and the numerical environment variables
- sol_vars – sets up the solution variables as placeholder arrays
- solve_PDE – solves the PDE by iterating across all points in time when called by calc_PDE
- fw – computes the Forward Euler scheme when called from solve_PDE
- dirichlet – implements homogenous Dirichlet boundary conditions for the Forward Euler scheme, called in the function fw
- none – implements 'zero' boundary conditions for the Forward Euler scheme, called by the function fw
- neumann – placeholder function for Neumann boundary conditions for the Forward Euler scheme, not implemented
- bw – computes the Backward Euler scheme when called from solve_PDE
- ck – computes the Crank-Nicholson scheme when called from solve_PDE

Examples of PDE_solver.py execution can be found in PDE_test.py. This file is structured with six functions:

- u_I – simulates the initial temperature distribution
- u_exact – simulates the exact solution

- p_func – simulates the Dirichlet boundary condition ‘p’
- q_func – simulates the Dirichlet boundary function ‘q’
- finite_diff_run – solves the function u_I with parameters kappa = 1, L = 1 and T = 0.5 for Forward Euler with zero conditions, Backward Euler and Crank-Nicholson and plots the results
- dirichlet_run – extends the Forward Euler example for Dirichlet boundary conditions and plots the returned results for x against u(x, 0.5).

The generated plots indicate the performance of each difference method, as seen below



Left: plot of x against solution u(x, 0.5) for all three difference schemes and the exact solution. Right: plot of Forward Euler with Dirichlet boundary condition solutions.

2. SOFTWARE DECISIONS

2.1 Simulating Equations and ODES as Functions

To minimise hard-coding and reduce clutter throughout the software, each simulated equation, and ODE is declared and stored in ode_functions.py, a dedicated file. This reduces the runtime of scripts within the software and saves computing memory, as these functions are only imported and called when necessary. Moreover, a dedicated file for all equations makes accessing, altering, and debugging these functions easier than having various functions scattered across different files.

2.2 Initial Value Problems

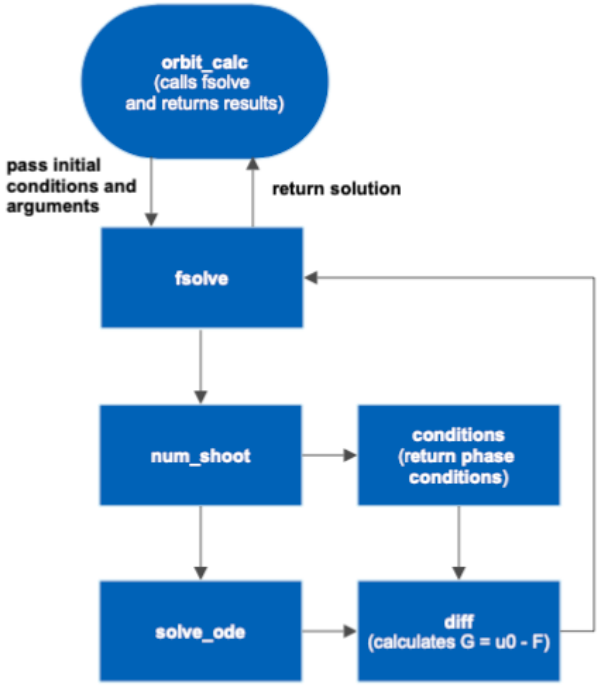
Each part of the integration estimation process in integrate_ode.py was modularised using distinct functions. This enabled me to debug my code with greater ease as the code was clear and concise throughout. Modularisation of the code also reduced the runtime of the software and made my code future-proof for any additions I made later in the project. In particular, returning to integrate_ode.py to include args parameters for natural continuation was possible by simply adding this new parameter to each required function.

To extend the Euler and RK4 methods to handle systems of ODEs, I implemented NumPy arrays to store and return variables. This enables multi-dimensional arrays of inputs to be passed to each function. Compared to my initial use of standard lists, floats, and integers, these NumPy arrays proved to be far more robust and kept code runtimes to a minimum. In addition, the wide range of NumPy functions that can be used enabled me to perform more actions in fewer lines of code and reduced the number of unnecessary loops used.

I attempted to automate the process of plotting the error and solution plots in integrate_plots using a dictionary of methods, with error plots of both methods called once from a single 'for' loop. This allowed me to tweak my plotting functions without having to check excessive lines of code and duplicate variables for errors. In addition, the inclusion of this automation feature future-proofs the code so that any further 1-step integration methods can be included and their error vs timestep investigated relatively simply.

2.3 Numerical Shooting

I experienced difficulties understanding how to code the numerical shooting algorithm at the start of its development. To overcome this and formulate the structure used within the code, I wrote out a flowchart for the entire process. This flowchart is seen below



This structure is simple, yet effective and allowed me to fully understand each step of the process. By modularising each part of the algorithm with a dedicated function wherever possible, I could

call certain parts without having to run the entire file and thus reducing computing power and memory. This proved extremely useful during the natural parameter continuation development, as I did not need to call the 'orbit_ode' function to implement parameter continuation properly. A suitable alternative structure could be created using functions. I would like to have explored this option during development but opted not to due to time constraints.

From my initial experience of developing the ODE solver, I opted to use NumPy arrays once more to ensure that variables were as robust and efficient as possible. This also ensured that my shooting was fully compatible with my ODE solver from the start of development.

I chose to use a lambda function to call 'num_shoot' from inside fsolve's arugments to keep this call concise. I tried to distribute the methods carried out in 'num_shoot' into as few lines of code as possible. For instance, the phase condition of shooting is calculated in the dedicated function 'conditions', where an ODE's relevant equation is indexed based on the input parameter 'var' and passed the initial conditions u0. I initially considered implementing the phase condition as an input argument to the 'orbit_ode'. However, this method would have required a function to be coded for each ODE's respective phase condition, making it less efficient than the method I ultimately opted for.

I chose to implement fsolve for the root-finding problem as it is extremely easy to implement and use, given the extensive documentation available for it. However, different options for the root-finding problem could have been implemented such as the Newton-Raphson method to compare their effectiveness.

2.4 Code Testing

To test my shooting code, I devised a simple method of comparing the shooting results against explicit analytical solutions using the NumPy function '.allclose'. From this, I was able to determine whether my shooting code performed as expected whilst considering the slight discrepancy between results. The start points, end points and orbit are tested for both the Hopf bifurcation and 3D extended version of the Hopf bifurcation. To keep track of the number of tests that would pass or fail for both examples, I included a simple pass and fail counter that would add to a list based on the outcome of each test.

I made use of the '.isinstance' and 'callable' functions throughout the software to ensure that parameters are passed to each function in the correct format. I implemented these functions throughout the development of each part of the software, which helped identify errors in the code. However, the dynamical nature of this software's development meant that variables would often change type in a short space of time. This caused me to experience issues when I had forgotten to

amend the variable checks, which was proved to be an easy mistake. An alternative approach would have been to implement variable checks at the end of successfully completing each component of the software. This way, time looking through code for incorrect variable checks would be saved and tests could be implemented without hindering a component's integrity.
I investigated the use of unit tests but struggled to develop them for my software. Had I managed to implement unit tests, these would have been carried out for the core part of the software and integrated with my use of Git to ensure all code was stable.

2.5 Parameter Continuation

From my experience developing numerical_shooting.py, I aimed to replicate the same concise format for my natural parameter continuation component. Within the function 'natural_continuation', a for loop iterating across each parameter value carries out natural parameter continuation algorithm. To solve the root-finding problem, I opted to use fsolve as I had done previously. Similarly, I made use of NumPy arrays throughout the file. By this point in the software's development I was highly familiar with how to use these functions correctly, and so it was logical to continue to do so. In addition, the consistency in methods used throughout the software made it easier for me to debug, make amendments and integrate each component together.
To pass the varied parameter between each component of the software, I used the *args and args syntax in all relevant functions. I had difficulties understanding the concept initially but after extensive trial and error, I was able to successfully utilise the syntax to code a working natural parameter continuation module. Before researching and testing the args syntax, I considered passing parameters using a dictionary method. Although this method was easier to grasp, I felt that it would be unsuitable for the style of coding that I had begun using throughout my software. However, I would like to have explored this method in further depth to compare its effectiveness against the method I opted for.
The 'natural_continuation' function makes use of lambda functions to determine a suitable discretisation for a given function. As mentioned previously, my familiarity with lambda functions by this stage of the development made it logical to implement them again. After some unsuccessful attempts to integrate 'natural_continuation' with numerical_shooting.py, I realised that I was calling the wrong shooting function. To rectify this, I amended the discretisation for the Hopf bifurcations to call the 'num_shoot' function instead of the main 'orbit_calc' function.
I experienced difficulties plotting the bifurcations for each continuation case as the Matplotlib plot functions were calculated within 'natural_continuation', after the result was returned. To

amend this and extend further modularisation to the software, I placed the Hopf plotting functions in a separate dedicated function, called at the end of iteration within 'natural_continuation'.

2.6 PDEs

One of my key design decisions in PDE_solver.py was to separate each difference scheme, relevant boundary condition and solution plotter into distinct functions. I was able to cumulate the techniques used in earlier stages of the development to code this component effectively. In addition to the core PDE functions, I created functions to initialise parameters and set up solution variables. Splitting up each difference scheme into a dedicated function made debugging and testing much easier. To call a respective scheme given a user's input and make use of input tests, I created a dictionary of methods in 'solve_PDE'. Using an 'if' statement, a user's method input is checked against the dictionary items and either called or an 'Exception' is returned.
I struggled to understand some of the PDEs content to begin with, but after tweaking with the parameters and variables extensively in the myForwardEuler1DDiffusion.py file I understood the techniques and methods in greater detail. Once I had a better understanding of the content and implemented the additional schemes and the matrices methods, I was able to extend the Forward Euler scheme for Dirichlet boundary conditions. To do so, I implemented a boundary condition dictionary within the 'fw' function. As with the methods dictionary mentioned previously, the user's boundary conditions are checked against the dictionary items to either call the respective function or raise an 'Exception'. The two implemented Forward Euler conditions, Dirichlet and zero, are computed in functions 'dirichlet' and 'none' respectively. Although I was unable to implement the Neumann conditions, I kept the function and dictionary key and created a placeholder 'neumann' function to future-proof my code.
The PDE plots are integrated directly into the examples found in PDE_test.py. I made this choice after experiencing considerable difficulty with passing each respective scheme or 'sub-scheme' into a dedicated plotting function within PDE_solver.py. Instead, my choice ensures that the PDE computations are carried out effectively without being a detriment to the plotting tasks.

4. REFLECTIVE LEARNING LOG

- What did I learn about the mathematical algorithms? I.e., solving boundary value problems, numerical ill-conditioning, etc.
    - I have strengthened my knowledge of 1-step integrators such as the Euler and RK4 methods through the analysis conducted in this software (e.g. the effect of the timestep on the error)

- Learned how to analyse and investigate the behaviour in time for ODEs that are new to me such as the Lokta-Volterra equations
- Learned various new algorithms and techniques for solving boundary value problems and numerical continuation problems, namely using numerical shooting and natural parameter continuation
- Learned how each algorithm can be used in conjunction with one another to solve a single problem, such as continuation by using shooting and 1-step integration
- Having previously learnt about bifurcations in a different unit, I have reinforced my understanding of the concept through the development of numerical continuation
- Expanded my general knowledge on PDEs and reinforced my understanding of different finite difference schemes
- Explored how PDE behaviour changes in time based on different parameters and constants, by visualising this behaviour

- What did I learn about software engineering? How have I progressed in my abilities?
  - How to use Git appropriately to record, track and access my code throughout the development process
  - The importance of regular Git commits to track development progress effectively
  - Sound software engineering principles such as modularising code, documenting code and using appropriate interfaces. I have implemented these principles wherever possible to ensure my code is professional and suitable for its purpose
  - The importance of code testing to ensure that a piece of software runs as intended
  - The importance of code structure and its effect on a piece of software
  - Expanded and improved my Python abilities considerably, including:
    - Using lambda functions as a placeholder
    - Modularisation of files to reduce execution times and improve code reliability
    - Correctly passing arguments between different functions
    - Using NumPy functions such as NumPy arrays to perform complex computations
    - Using *args syntax to pass additional arguments between functions
    - Using the __main__ top level code environment to prevent unnecessary code from being executed
    - New techniques such as simulating ODEs with code, unit testing, plotting time-series and phase portraits and packages such as SciPy fsolve and sparse

        - Improved my code testing and debugging skills

- What are the short-term implications of what I've learnt? (When will it be useful?)
    - The PDEs section of the software stands out as being extremely useful for my Continuum Mathematics unit. I feel much more confident in my understanding of PDEs behaviour and have explored the theory behind it in depth by developing the PDEs section of the software
    - Refreshed my knowledge of 1-step integration methods such as Euler and RK4, which is useful for potential work next year
    - Learned new theory such as numerical shooting and natural continuation which are plausible to re-appear in my future studies
    - Successfully developed a piece of software that completes (most) of the set objectives, which has boosted my programming confidence

- What are the long-term implications of what I've learnt? (When will it be useful?)
    - Improved my Python skills considerably, which will make future projects easier
    - I will be able to transfer the skills learned during the development to ensure that any future computing work I undertake is to a high standard, such as appropriate use of Git, code testing, documentation, and modularisation

- What would I have done differently if I started the unit over again?
    - Implement more 1-step integration methods into the ODE solving component
    - Use Git consistently from the very start of the project to track my progress, I had difficulties with Git at the start of the development including an error in my IDE that pushed duplicate commits
    - Distribute my time between each section more evenly, this hindered my ability to develop pseudo-arclength and further PDE extensions
    - Run more tests on my code and carry out unit tests with Git integration
    - Attempt some of the alternative methods detailed in Software Decisions such as parameter dictionaries and Newton-Raphson solver
    - Approach the more challenging tasks with a more ‘computing’ than ‘mathematical’ mindset (i.e. think in terms of code from the very start)

- What will I do differently in the future?
    - Extend my usage of Git such as by using branches
    - Implement classes to optimise code structure even further

- Manage my time more effectively to ensure that each objective is completed as much as possible
- Create a ‘roadmap’ plan at the start of a project to guide my development, I felt very overwhelmed with each task at times during the development
- Experiment with alternative methods and techniques to compare and find the optimal ones for a given task
- Automate and modularise any code I write as much as possible to ensure my work is efficient and effective
- Research and read up more on scientific or mathematical theory that I don’t fully understand