EMAT30008 Scientific Computing Report

Jack Parr - 1925164

https://github.com/jack-parr/emat30008

April 2023

1 - Summary of Software

This section outlines the capabilities of the software package, by detailing the methods incorporated into each file and providing examples of applying the functions and comparing their accuracies against known true solutions. The code itself is not included, but all code used to produce figures and values can be found in the *report_figs.py* file.

1.1 - ode_solver.py

The goal is to produce a software capable of solving single and multi-dimensional systems of ODEs, utilising the Euler and 4th Order Runga-Kutta (RK4) integration methods to solve the solution at timesteps. A maximum timestep can be specified which impacts the number of steps the algorithm must take. The first example shows the *solve_to* function evaluating the ODE equation $\dot{x} = x$, with initial condition $x(0) = 1, t \in [0, 1]$, and $\Delta t_{max} = 0.05$. Solutions to $x(1)$ are found using Euler and RK4 integration. These are compared to the known solution $x(t) = e^t$ as shown in Figure 1. This plot shows the solution generated using RK4 are accurate enough to be obscured on the graph, while Euler method results in a small under-approximation.

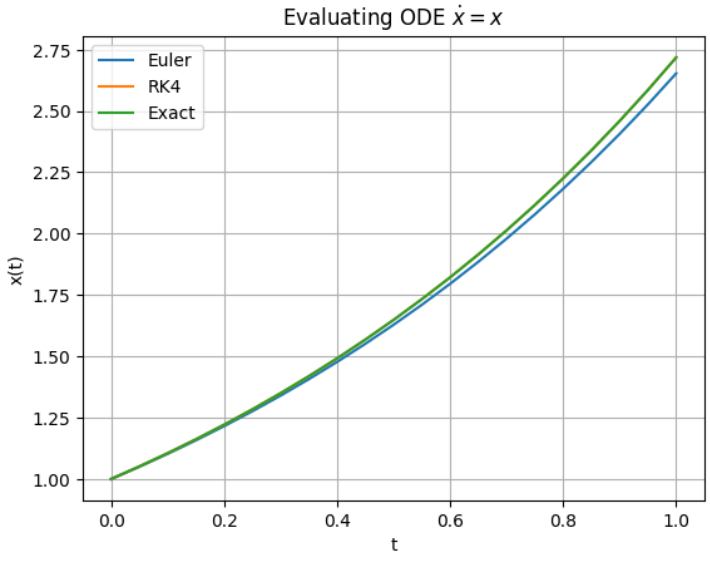

Figure 1: Using the *solve_to* function with Euler and RK4 integration.

The *solve_to* function is also capable of evaluating systems of ODEs. To demonstrate this, the next example solves the ODE $\ddot{x} = x$, which is equivalent to the system $\dot{x} = y, \dot{y} = -x$. The same initial conditions are used, with $x(0) = [1, 1], t \in [0, 10]$, and $\Delta t_{max} = 0.01$. Solutions to $x(5)$ are found and compared to the known solutions $x(t) = cos(t) + sin(t), y(t) = cos(t) - sin(t)$, as shown in Figure 2. For this problem, Euler and RK4 appear to perform similarly with the reduced Δt_{max}, both closely approximating the exact solution.

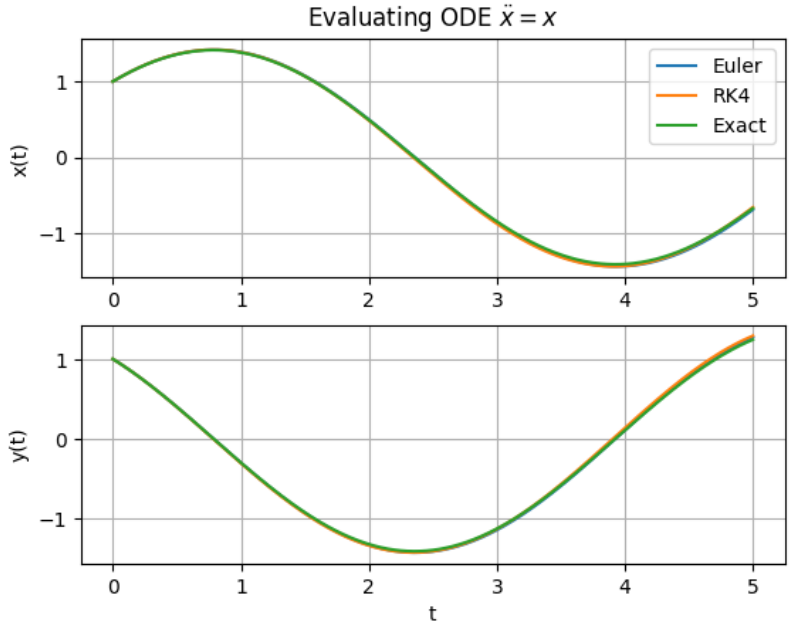

Figure 2: Using the *solve_to* function to solve a system of ODEs.

In order to better understand the difference in performance between Euler and RK4, an error plot is shown in Figure 3 for different Δt_{max} values when solving example 1. This shows that RK4 consistently produces more accurate solutions than Euler with the same timestep, enabling RK4 to accurately evaluate the problem faster using larger timesteps. Points are marked on Figure 3 showing Δt_{max} values where the absolute accuracies are the same, which is 0.000115 for Euler and 0.312572 for RK4. Timing these shows Euler taking ~127ms, and RK4 taking ~1ms on my hardware, demonstrating the benefit of RK4.

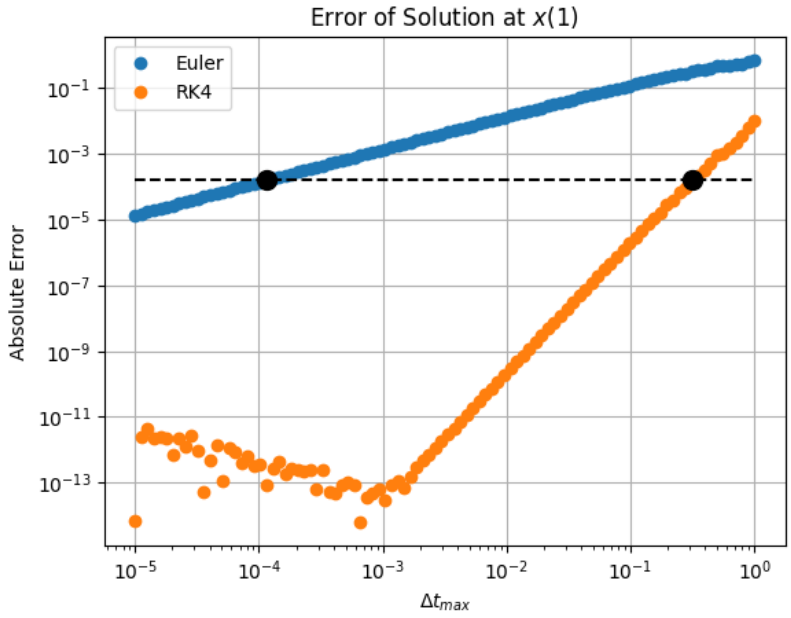

Figure 3: Absolute errors using the Euler and RK4 integration methods.

1.2 - numerical_shooting.py

This file provides the capability of finding orbits within a system. When calling the *orbit_shoot* function, it constructs and solves the shooting root-finding problem for the given system of ODEs. A phase condition can be provided in the form of a function, which is important for ensuring a solution is found for autonomous problems. The solving function used to solve for the roots can be specified, however this function is designed specifically for use with *scipy.optimize.fsolve* for reasons explained in section 2. To demonstrate the use of *orbit_shoot*, the predator-prey equations are used, in the form:

$$\dot{x} = x(1-x) - \frac{axy}{d+x}$$

$$\dot{y} = by(1-\frac{y}{x})$$

$$where: a = 1.0, b = 0.2, d = 0.1$$

An initial guess of $[0.5, 0.5, 20]$ and phase condition $\dot{y} = 0$ are used to identify an orbit, which is found to begin at $x = y = 0.383$ with a period of 20.72 seconds. Figure 4a illustrates the solution to the system of ODEs found using *solve_to* with those starting conditions. Figure 4b shows the same solution with the solved variables on both the axes, with the closed loop indicating an true orbit has in fact been found.

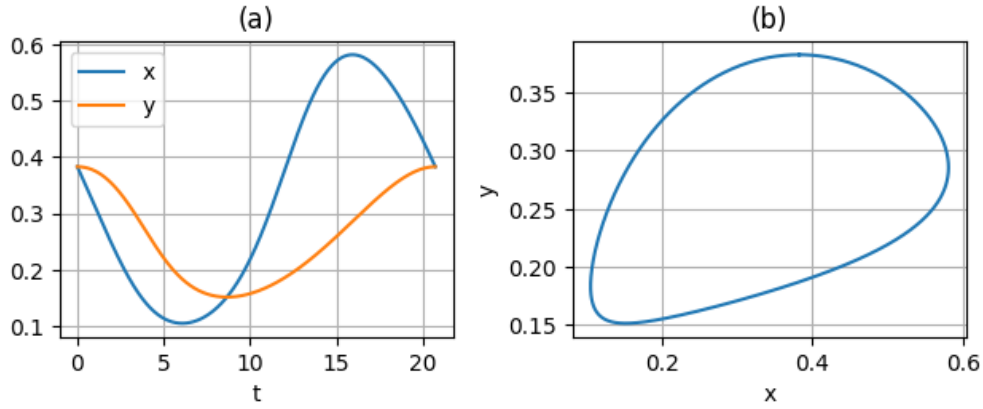

Figure 4: Using the *orbit_shoot* function on the predator-prey model.

1.3 - numerical_continuation.py

This file contains functions to perform numerical continuation, specifically natural parameter continuation and pseudo-arclength continuation. These allow us to see the change in behaviour of a system as a parameter changes, with *pseudo_arclength* also incorporating the pseudo-arclength equation into the problem. Both functions operate based on a predicted value, calculated from the two previous soluions. First, they are both used to solve the algebraic cubic equation $x^3 - x + c = 0$ as parameter c varies between [-2, 2]. For this problem, a discretisation is not needed, and *scipy.optimize.fsolve* is used as the solver. Figure 5 shows that *natural_continuation* fails at the fold bifurcation point as the search line fails to intersect the solution, whereas *pseudo_arclength* is able to continue capturing the behaviour past this point.

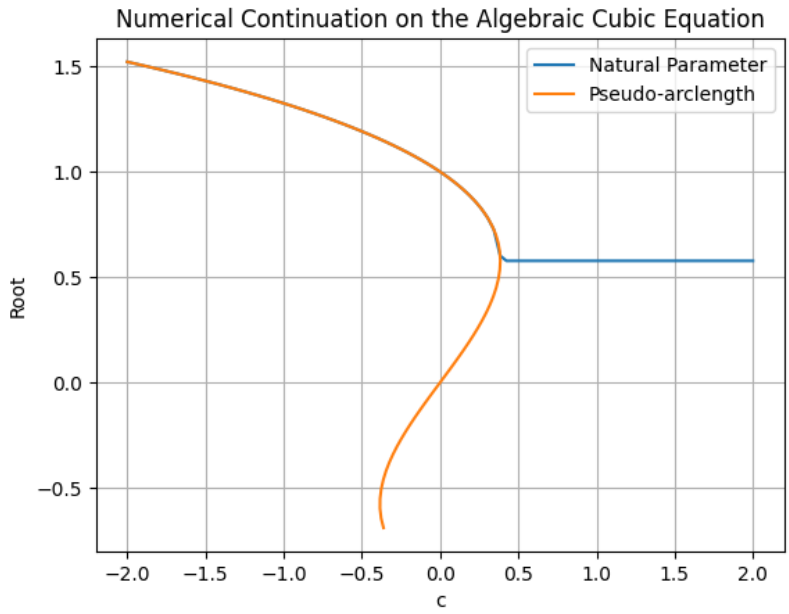

Figure 5: Using my numerical continuation functions on the algebraic cubic equation.

The next problem does require discretisation, which is the *shooting_problem* function within *numerical_shooting.py*. The continuation functions are both used to solve the Hopf bifurcation normal form equations, which take the form:

$$\dot{x} = \beta x - y - x(x^2 + y^2)$$

$$\dot{y} = x + \beta y - y(x^2 + y^2)$$

Parameter β varies between [0, 2]. Figure 6 illustrates how *natural_continuation* has the same issue of failing at the fold bifurcation point.

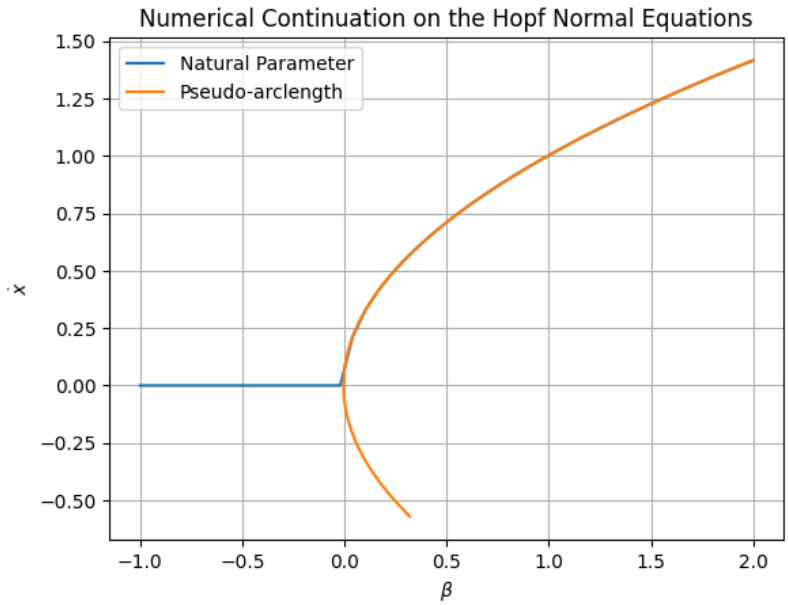

Figure 6: Using my numerical continuation functions with a discretisation function, on the Hopf normal equations.

1.4 - pde_solver.py

This file provides the ability to solve diffusive PDE equations. Different boundary conditions can be specified, which can be of Dirichlet, Neumann, or Robin form. The problem is solved using sparse matrices, and the method used to solve can also be chosen from method of lines, explicit Euler, implicit Euler, or Crank-Nicolson. The method of lines uses the RK4 option of *solve_to* from *ode_solver.py*. To demonstrate them, the linear diffusion equation is solved, which is defined by $\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2}$, with boundary conditions $u(0,t) = u(1,t) = 0$, initial condition $u(x,0) = \sin(\pi x)$, and parameters $D = 0.1, N = 100$. Note that for implicit Euler and Crank-Nicolson methods, $\delta t = 0.1$ is used, however the stability condition limits the method of lines and explicit Euler methods to $\delta t = 0.0005$ is the maximum they are capable of using for this problem. Figure 7a shows how the implicit Euler method is the most inaccurate, while the other methods are very precise. To demonstrate the other boundary conditions, Figure 7b shows the same problem again, however this time both the boundary conditions are specified as Neumann. They take the form $\frac{\partial u}{\partial x}\,|_{x=0} = 0, \frac{\partial u}{\partial x}\,|_{x=1} = 0.$

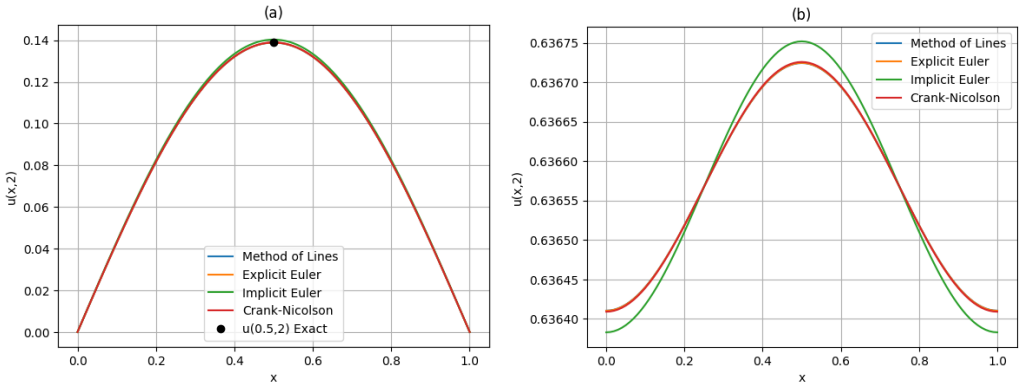

Figure 7: Using the *solve_diffusion* function with (a) Dirichlet and (b) Neumann boundary conditions.

The *solve_diffusion* function is also designed to be compatible with the natural continuation functions. Figure 8 shows the Bratu problem solved using the Crank-Nicolson method and using the *pseudo_arclength* function with parameter μ varying from [0, 4]. The graph shows how the fold is found, however the function fails to navigate around the fold in this case.

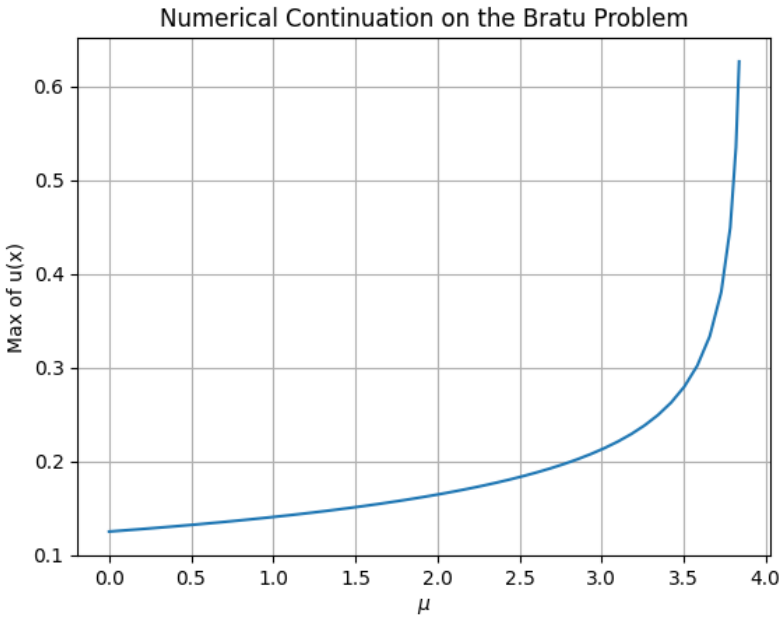

Figure 8: Using the *pseudo_arclength* function on the Bratu problem solved with the *solve_diffusion* function.

---

2 - Software Design

This section describes key software design decisions that were made throughout the development, and how they link towards the overall goal. In order to make the software easy to understand and use, early in the design process a standard format of outputs was decided upon, which is an numpy.ndarray with a row of values for each relevant element being returned, with solved variables always coming first. This was chosen because most of the outputs of the functions within the software are best visualised with a plot. Structuring the outputs like this makes indexing the different elements for the plots easy. An alternative approach could have been to return each element as its own different variable (eg: return x,

b) however this was avoided as it makes errors involving unequal length dimensions of the outputs more likely to go unnoticed. Containing everything into the same ndarray makes it easy to ensure the outputs are all consistent.

Similarly, the formats of the inputs was standardised as well, making combining the functions easier. This became especially important in cases such as the numerical continuation functions, which use the *shooting_problem* function from numerical shooting as a discretisation function. Standardising the inputs also makes it easier to combine the PDE solver with numerical continuation using a simple wrapper which the user custom makes for their problem. In cases like these, the documentation of the software becomes the primary source of understanding for the user. This example also demonstrates how certain stages of process were kept in separate functions within files, in cases where another file would use one of the stages. Keeping the functions separate enabled cleaner coding without repetitions.

I also standardised the format of all the documentation, which is contained at the beginning of each function. This enables the user to quickly scan the text for the specific information they require, which includes input formats, output formats, and a description of what each variable is. An improvement to this would be to include examples of the functions being used, however this was not done because of how poor they look in the text format within the file. This can be overcome by moving the documentation to a different medium, for example on a website or on a separate document within the repository.

Gracefully handling errors is key to user-friendly generalised software, therefore a separate file called *input_checks.py* was made, which contains simple functions for checking the format of function inputs before carrying out the function. These feature informative error messages which allow the user to efficiently locate the source of the error, instead of allowing the error to cause a random issue somewhere within the function and return a meaningless error message. Within some functions, custom error handlers are also implemented, for example with decision variables requiring a string input. These check that the string is a valid option, and if not display a list of valid options to choose from. Some inputs are not always required to be in the same format, for example with single ODE equations vs systems of ODEs. The single ODE could have an initial value given as an int or float, however the system requires it to be in a list or ndarray form. To overcome this, I made it so the function always requires it to be given in list or ndarray form, which is simpler for the code to handle however less intuitive for the user. An alternative approach could have been to include an extra check or input variable which indicates the nature of the problem and allows the user more flexibility. Unfortunately, there is not custom error message implemented for when the solver fails to converge, or runs endlessly without converging.

While having the option of various solving methods is useful, for many problems the user simply desires a solution by any means. Therefore, a lot of input parameters have been given default inputs so that code will still run without the user considering it. For example, in the *solve_to* ODE solving function, RK4 is used as the default after demonstrating more accurate outputs. In cases such as the solving function for *orbit_shoot*, the default is Scipy's *fsolve*, chosen because Scipy functions are far more reliably tested than any functions

developed for this coursework. This was also important for optional inputs, such as the various argument lists. In the *solve_pde* function, the use of sparse matrices can be specified by the user, which was only incorporated for my investigations. The default is to use them, due to the greatly enhanced computational efficiency they provide when solving the systems by reducing memory usage.

The design decisions around the *pde_solver.py* file were particularly interesting. During intitial development, the various methods were separated into functions that could be called based on the desired method. However, this resulted in a large amount of repetition and difficulty when refining the code. In order to make it more user-friendly and developer orientated, the methods are now all combined into a single function. To do this, similarities between the matrices used for the solving stage were identified and base matrices are generated. These are then modified based on the boundary conditions, and the output from the solver is similarly modified. I allowed the user to be able to specify each boundary condition type individually, which unfortunately means the modification of the matrices looks clumsy within the code. An alternative way of handling the boundary conditions could have been a custom made class containing all information, with just two objects of that class being input into the *solve_diffusion* function for the boundary conditions.

Ensuring the stability of the various functions throughout developement was important, and was achieved using unit tests. All of the unit tests are contained inside one file, called *all_unittests.py*, which enabled easy running after changes were made the structures of functions. These unit tests involve testing each function with problems and checking their outputs against known solutions. Using this allowed me to identify any misfiring code upon changes, and doing so quickly prevented me from becoming dependent on new code when it was faulty.

---

3 - Reflective Learning Log

This coursework has greatly increased my depth of understanding regarding the mathematics of these processes. I often use various equations and functions without any thought towards the fundamental concept of them, but coding them like this caused me to better appreciate them in order to effectively trouble shoot. I am pursuing a career in AI, therefore understanding the mathematical concepts behind my tools will enable me to produce more tailored and efficient products.

Additionally, my understanding of software engineering practices has significantly increased. I was already familiar with using GitHub and regularly comitting changes, however my previous code has typically been messy and difficult to interpret. I have found that modularising functions into well-structured and documented files better enables me to return to the work after an extended absence. Being able to read through my notes made it easy to understand my previous decisions. Including custom error checking functions within the software also helped greatly with trouble-shooting when applying the functions. Something I was not previously aware of was unit tests, which proved to be an elegant way of checking the functionality of the whole repository at once. In later stages of the

development process, optimisation became the focus, and I learnt about useful principles such as sparse matrices and the benefits of numpy.ndarrays over lists.

The short-term implications of what I have learnt are my understanding of the mathematical processes including numerical continuation, orbit shooting, and solving boundary value problems. This specific knowledge may or may not be useful in the future, but what certainly will be are the software engineering principles I have developed. In real industry, I am more likely to be working in a team rather than solely, so proper formatting and documentation of code is important, as well as clear communication of concepts and thought processes. I found during the pair-coding exercises in the lab sessions that many people approach coding problems in different ways, therefore communication using clear and correct terminology was important for effective work. As well as this, my personal approach to coding problems has been refined signficantly. Before, I used to rush into attempting to create the entire function at once, and then encountered problems in strange orders which made development difficult. Instead, creating a flow chart of processes outside of the IDE first and then implementing them in stages led to a far smoother experience. This flow chart was also very useful when optimising the software, as it enabled me to easily identify overlaps in processes and isolate functions that could be reused. Applying the methods learnt for identifying bottlenecks regarding computational and memory efficiency will help ensure my future work is scalable.

If I were to start this module again, I would have also kept a logbook separate to the python files in the repository. While my notes and documentation made it easier to interpret after absence, these notes were still polluted by the code itself. Having my thought processes towards the overall structure of each function clearly outlined in a separate document would have helped ease this problem, and also prevent smaller tasks from getting forgotten with time. In terms of my coding, I feel my weakest area currently is streamlining how the user interacts with my functions. Especially with the *solve_diffusion* function, there is a long list of inputs required. Some of these, for example all inputs pertaining to the left boundary conditions, could be combined into a class object which the user uses as an input. Structuring functions like this would have led to an easier experience using the functions, compiling relevant information together. I would also focus on developing more custom error handlers, for example when the solver fails to converge.

When I am completing software engineering project in future, I will ensure to better refine the user experience, which is extremely important if my work is intended for long-term use. I will also include a more detailed documentation on a separate medium to the code files. This would likely be a website, which enables nicely formatted examples to be included, as well as hyperlinks to quickly navigate between relevant sections.