Scientific Computing Report

Christopher White

EMAT30008

Department of Engineering Mathematics, University of Bristol

April 26, 2022

1 Software Summary

The software contains methods for numerically solving ODEs of arbitrary dimensions with an option of four different time stepping methods of different orders. These methods are then used in the implementation of numerical shooting, which can be used to find periodic orbits of ODEs. Furthermore, this method is used in the implementation of numerical continuation where orbits and equilibria are found for various values of some parameter. Two methods of numerical continuation are implemented, natural parameter continuation and pseudo-arclength continuation.

Three numerical methods for solving 2nd order diffusive PDEs are also implemented. These are the forward Euler, the backwards Euler and the Crank-Nicholson methods. All three of these methods can be used to solve PDE initial boundary value problems with Dirichlet, Neumann, Robin and periodic boundary conditions. The methods can also be used to solve problems with a reaction term on the right hand side that can be either linear or non-linear as well as a variable diffusion coefficient (although this must be linear).

As well as these methods, some functions are included for the easy plotting and visualisation of the generated solutions.

The software described in this report can be accessed in the GitHub repository [1]. Examples of the implemented methods can be seen below, with more detailed examples available in the examples folder in the GitHub repository and in the `Python` scripts for each method.

1.1 Solving ODEs

To solve an ODE, it must first be defined as a `Python` function. For example, the simple ODE,

$$\frac{dx}{dt} = x, \tag{1}$$

can be written as a `Python` function as below.

```
def f(x, t, params):
    return x
```

The ODE can then be solved using the forward Euler method by calling the `solve_ode` function as follows.

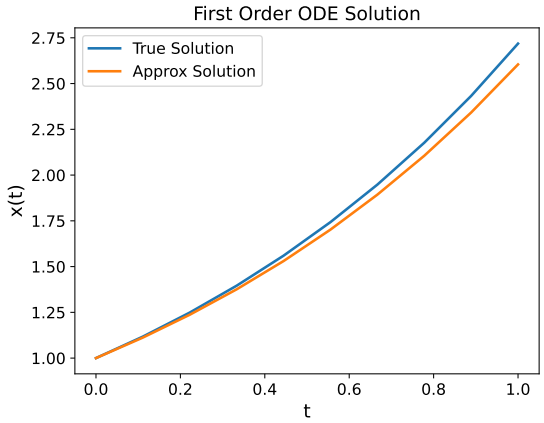
```
X = solve_ode('euler', f, t=np.linspace(0,1,10), [1], h_max=0.1)
```

This returns an array with the numerical solution at each time given in the provided array, `t`.

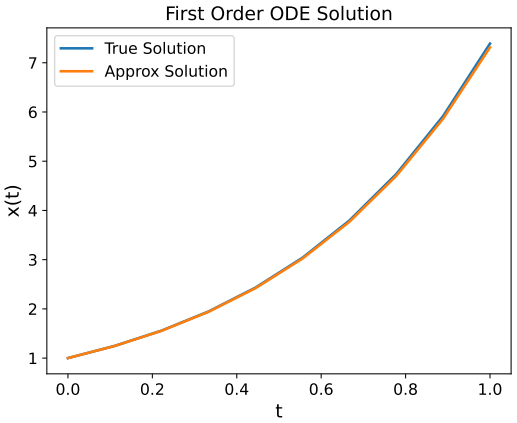
This solution can be plotted using the included `plot_solution` function, alongside the analytical solution, $x(t) = e^t$.

```
X_exact = lambda t : np.e**t
plot_solution(t, X, 't', 'x(t)', 'First Order ODE Solution', X_exact)
```

The output of this function can be seen in Figure 1a.



(a) Numerical and exact solutions to the differential equation defined in Equation 1 using the forward Euler method.



(b) Numerical and exact solutions to the differential equation defined in Equation 2 using the midpoint method.

Figure 1

The same process can be done using a different method and with a parameter included in the ODE. For example, the 2nd order midpoint method can be used to solve

$$\frac{dx}{dt} = ax, \tag{2}$$

using the following code.

```
def f2(x,t, params):
    a = params['a']
    return x*a
X = solve_ode('midpoint', f2, t, [1], h_max=0.1, a=2)
X_exact = lambda t : np.e**(2*t)
plot_solution(t, X, 't', 'x(t)', 'First Order ODE Solution', X_exact)
```

The output of which is shown in Figure 1b.

It is possible to solve higher order ODEs or systems of first order ODEs using the same methods. The 2nd order ODE,

$$\frac{d^2x}{dt^2} = -x, \tag{3}$$

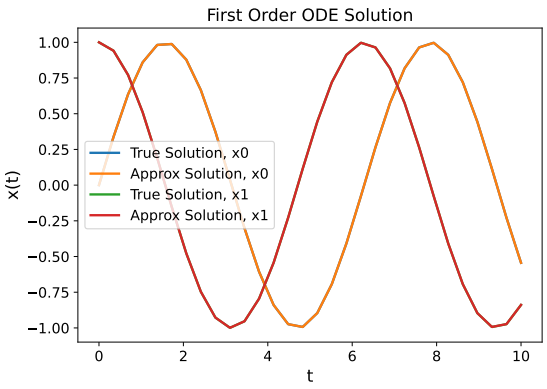
can be written as two first order ODEs,

$$\frac{dx}{dt} = y, \quad \frac{dy}{dt} = -x. \tag{4}$$

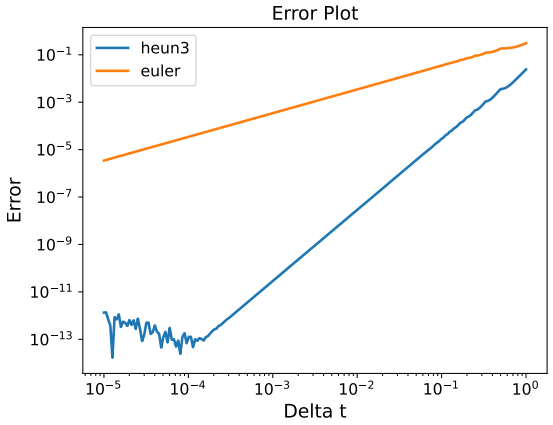
Once encoded as a `Python` function, it can be solved and plotted, using the Heun 3rd order method, with the following function calls.

```
X = solve_ode('heun3', g, t, [0,1])
plot_solution(t, X, 't', 'x(t)', '2nd Order ODE Solution', X_exact)
```

The output of the plotting function can be seen in Figure 2a.



(a) Numerical and exact solutions to the differential equation defined in Equation 3 using the Heun 3rd order method.



(b) Error plot showing comparison between Heun 3rd order method and the forward Euler.

Figure 2

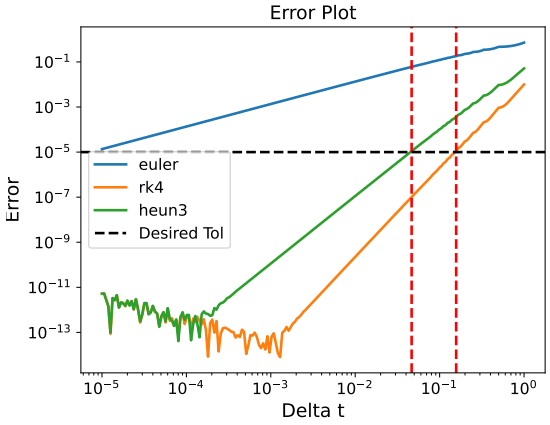
Also included are two functions to evaluate the different methods. The first one, `plot_error` plots the error of the specified methods against the step size h. An example can be seen in Figure 2b. The second function is `evaluate_methods` which produces a similar error plot but also evaluates the ability of the specified methods to meet a certain tolerance. Each method is timed and the largest possible h value is given to meet the specified tolerance.

Below is an example usage and output from `evaluate_methods`.

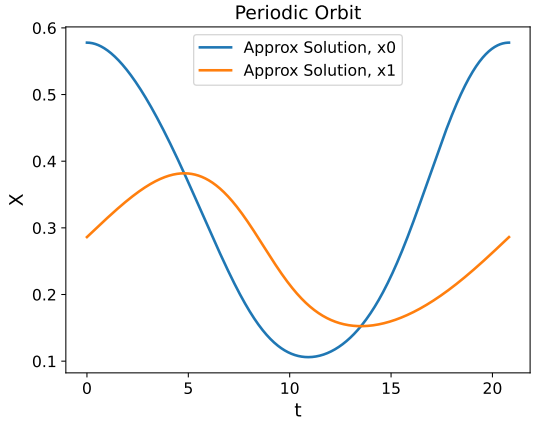
```
evaluate_methods(['euler', 'rk4', 'heun3'], f, 10**-5, 0, 1, [1], np.e)
```

```
Method: euler
Method cannot reach desired tol

Method: rk4
h to meet desired tol: [0.15702901]
Time taken to solve to desired tol: 0.0s

Method: heun3
h to meet desired tol: [0.04659526]
Time taken to solve to desired tol: 0.0019974708557128906s
```

The produced plot can be seen in Figure 3a.



(a) Plot produced by `evaluate_methods` function showing error against step size.



(b) Plot showing the periodic orbit found by the numerical shooting method.

Figure 3

1.2 Numerical Shooting

To find periodic orbits in the solutions of ODEs, numerical shooting can be used. An example of an ODE system that contains a periodic orbit is the predator-prey model,

$$\frac{dx}{dt} = x(1-x) - \frac{axy}{d+x}, \qquad \frac{dy}{dt} = by(1-\frac{y}{x}). \tag{5}$$

Details of encoding this system as a `Python` function can be seen in the example notebook in the repository. Once the ODE system is defined, a phase condition is necessary to ensure that an orbit can be found. The phase condition,

$$\frac{dx(t=0)}{dt} = 0, \tag{6}$$

will be used. This must also be encoded as a `Python` function.

Numerical shooting can be carried out using the following function call.

```
X0, T = numerical_shooting([1.3, 1.3], 10, predator_prey,
                           pc_predator_prey, a=1, b=0.2, d=0.1)
```

Here, the returned variables are the initial conditions of the orbit and its period respectively.

Also included is a function for plotting the resulting orbit, it is called as below.

```
plot_orbit(X0, T, predator_prey, 'Periodic Orbit', a=1, b=0.2, d=0.1)
```

The resulting plot can be seen in Figure 3b.

1.3 Numerical Continuation

Numerical continuation is the process of finding the location of periodic orbits or equilibria as a parameter in the equation or system is varied. Two methods of numerical continuation are implemented in this package, natural parameter continuation and pseudo-arclength continuation.

Firstly, numerical continuation can be applied to equations, for example, polynomials, to find the location of the roots as a parameter varies. For example, the equation,

$$x^3 - x + c = 0, \tag{7}$$

will have one or three roots, depending on the value of c.

The natural parameter continuation method can be carried out using the following function call.

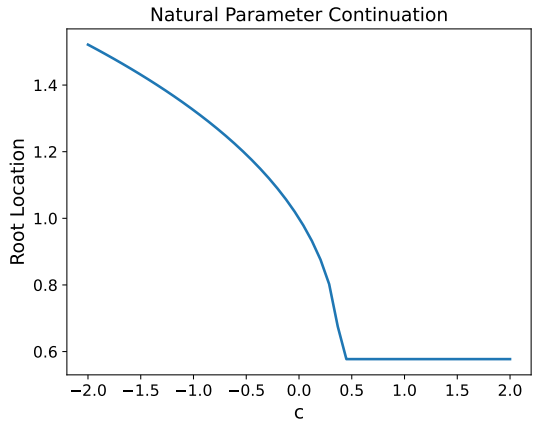
```
u, p  = continuation([1], 'c', [-2,2], 50, cubic,
                     method='natural-parameter', c=-2)
```

This returns `u`, containing the location of the roots and `p`, which contains the values of c corresponding to the roots in `u`. Similarly, pseudo-arclength continuation can be used with the following function call,

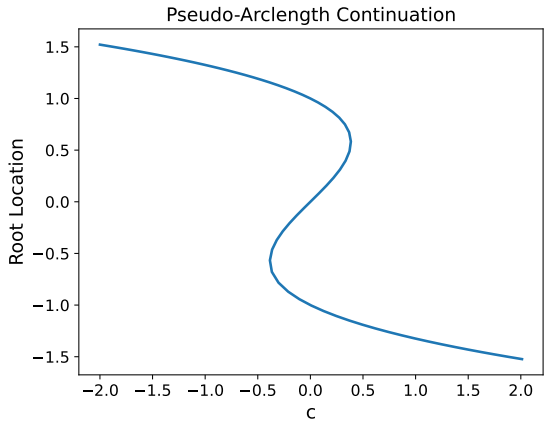
```
u, p  = continuation([1], 'c', [-2,2], 50, cubic,
                     method='pseudo-arclength', c=-2)
```

This method returns the same variables.

By plotting `u` against `p`, the plots in Figure 4 can be obtained.



(a) Result of applying natural parameter continuation to Equation 7.



(b) Result of applying pseudo-arclength continuation to Equation 7

Figure 4

As well as finding the roots of equations, numerical continuation is used for plotting the locations of periodic orbits that result from the variation of some parameter. As an example, a modified version of the Hopf bifurcation normal form is used,

$$\begin{aligned}\frac{du_1}{dt} &= \beta u_1 - u_2 + u_1(u_1^2 + u_2^2) - u_1(u_1^2 + u_2^2)^2,\\ \frac{du_2}{dt} &= u_1 + \beta u_2 + u_1(u_1^2 + u_2^2) - u_2(u_1^2 + u_2^2)^2.\end{aligned} \tag{8}$$

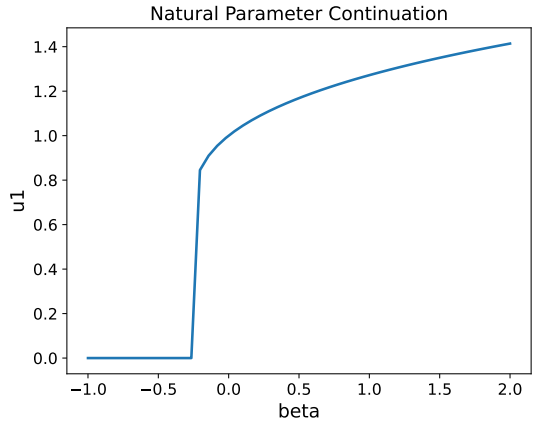
It is also necessary to define a phase condition for the orbits to be found, just as done in subsection 1.2. The phase condition, in this case, is

$$\frac{du_1(t=0)}{dt} = 0. \tag{9}$$

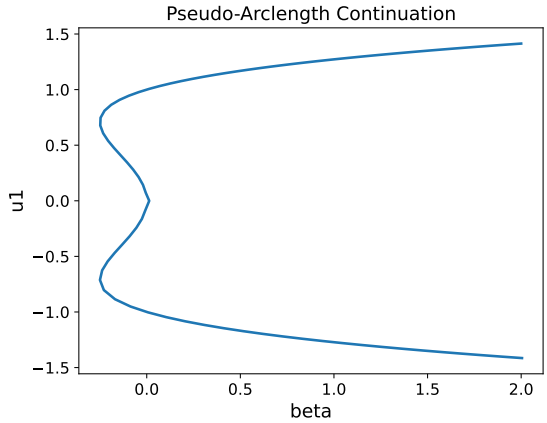
Numerical continuation, using both methods can be carried out using the following function calls.

```
u, p =  continuation([1,1], 'beta', [2,-1], 50, modified_hopf,
                     'natural-parameter', 'numerical-shooting',
                     pc_modified_hopf, T_guess=6, beta=-1)

u, p =  continuation([1,1], 'beta', [2,-1], 50, modified_hopf,
                     'pseudo-arclength', 'numerical-shooting',
                     pc_modified_hopf, T_guess=6, beta=-1)
```

The results of the continuation can be seen in Figure 5.



(a) Result of applying natural parameter continuation to Equation 8.



(b) Result of applying pseudo-arclength continuation to Equation 8

Figure 5

Note how, in both cases, natural parameter continuation fails to follow the curve around the 'corner' whilst pseudo-arclength continuation handles it gracefully. This is the main advantage of the latter technique.

1.4 Solving PDEs

This package contains methods for numerically solving 2nd order diffusive PDEs, of the form

$$\frac{\partial u}{\partial t} = \kappa(x)\frac{\partial^2 u}{\partial x^2} + F(u, x, t), \tag{10}$$

where the diffusion coefficient κ can be a function of space and the right-hand side function (or reaction term) can be either non-linear or linear. Solutions can be found for Dirichlet, Neumann, Robin and periodic boundary conditions.

Firstly, homogeneous boundary conditions, sinusoidal initial conditions and a function for κ must be defined as follows.

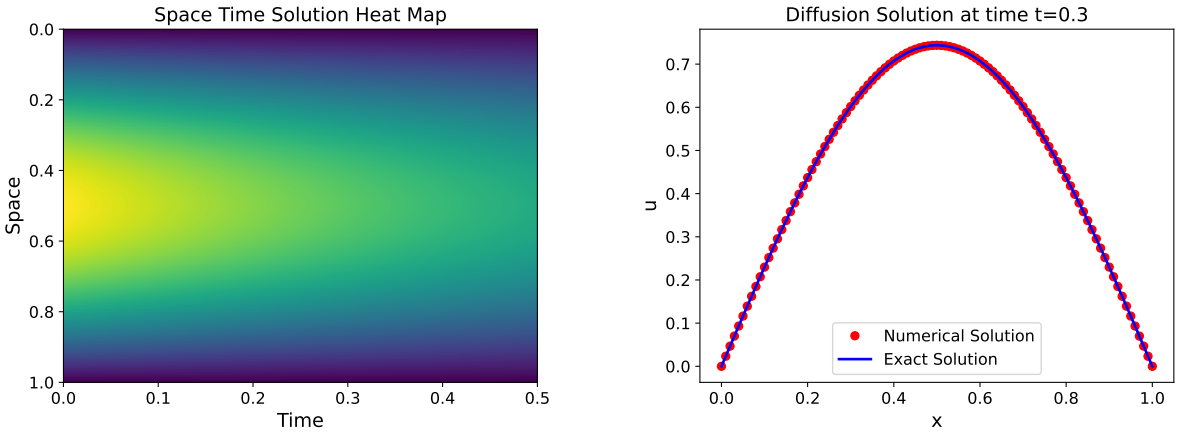
```
homo_BC = lambda x,t : 0
IC = lambda x,L : np.sin(np.pi*x/L)
kappa = lambda x : x/(x*10)
```

Then, the PDE can be solved with Dirichlet boundary conditions, using the forward Euler method. Note, the default is for the reaction term to be homogeneous, therefore, it is not necessary to specify it here.

```
u, t = solve_pde(1, 0.5, 100, 1000, 'dirichlet', homo_BC, IC,
                 solver='feuler', kappa)
```

This returns `u`, an array containing the solution for all times specified in the other returned array, `t`. There are two plotting functions for solutions to PDEs included, `plot_pde_space_time_solution` and `plot_pde_specific_time`. The former produces a heatmap, showing the solution in both space and time and the latter plots a curve that corresponds to the solution across the spatial domain at a specific time. These plots for the above example can be seen in figure Figure 6. The function calls to generate these plots can be seen below.

```
plot_pde_space_time_solution(u, 1, 0.5, 'Space Time Solution Heat Map')
plot_pde_specific_time(u, t, 0.3, 1, 'Diffusion Solution',
                       exact_solution)
```



(a) Solution in space and time of heat equation with homogeneous Dirichlet boundary conditions.

(b) Solution to heat equation with homogeneous Dirichlet boundary conditions at $t = 0.3$.

Figure 6

Figure 7 shows the solution of the heat equation with periodic boundary conditions and initial conditions of $u(x, 0) = x + 2$.

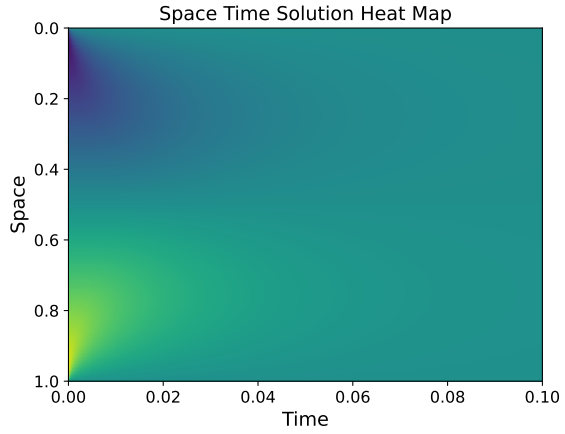
As well as the plots outlined above, a function, `animate_solution`, has been included to create an animation of the solution of the PDE as it evolves in time. This function can be used as follows.

```
animate_solution(u, t, L)
```

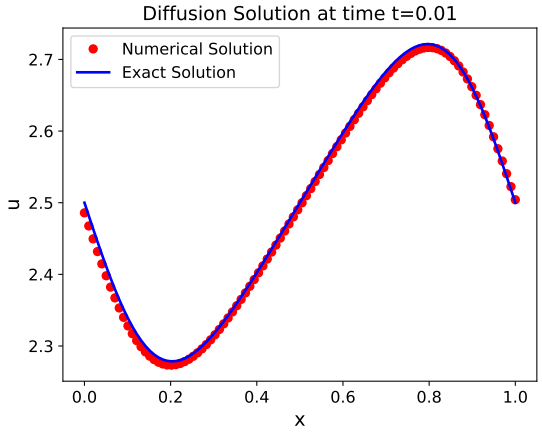
An example of an animation can be seen by running the `pde_solvers.py` script.

1.5 Examples

As previously mentioned, more detailed examples of all of the implemented methods are available both as scripts and as notebooks in the repository. This includes more detail on defining equations and systems of equations for use in the methods as well as more detail on the use of different types of boundary conditions for solving PDEs.



(a) Solution in space and time of heat equation with periodic boundary conditions.



(b) Solution to heat equation with periodic boundary conditions at $t = 0.01$.

Figure 7

1.6 Testing

The methods implemented have all been tested against analytical solutions. In order to run these tests, the `tests.py` script must be run. This also contains tests for the graceful handling of incorrect arguments and the graceful handling of root finding methods not converging. This was created using the built-in unit test functionality in `Python`.

2 Design Decisions

The overall goal with the code was to produce clean code that is easy to build upon and easy to reuse for higher level methods, for example, making the ODE solving code robust enough to work with both numerical shooting and continuation.

One design decision implemented throughout is the checking of the provided function arguments and the raising of custom error messages when the arguments are incorrect. This makes the code more user-friendly and makes it easier to debug.

In the following block diagrams, (Figure 8, Figure 9, Figure 10 and Figure 11) rectangular borders signify `Python` functions, circular borders signify a description of the code functionality and the double circular borders signify the end point of the function.

2.1 ODE Solving

The overall structure of the ODE solving code can be seen in Figure 8. The key design decisions can be found in the functionality of the `solve_to` function which includes the process of checking the step size, h. The times to evaluate the ODE at are provided in the `t` variable and it is important to ensure that the ODE is evaluated at exactly these times, even if the selected step size does not exactly meet these values. This is done by taking as many steps as possible using the provided step size and then completing one final step, to reach the value of t_{i+1}.

Also, there is a default value of $h = 0.1$ provided, therefore the function only needs to be provided the initial and final times for an accurate solution to be produced.

Evaluating the ODE at exactly the specified times is important because this will form a key part of the numerical shooting code which will use this method to solve the ODEs. Evaluating the ODE at specific times will be crucial for finding accurate periodic orbits in the ODE systems.

Another key design decision is making the `solve_to` function modular with respect to the time stepping algorithm chosen. This means that it is easy to implement new stepping algorithms and select the best one for the ODE system to be solved.

This function is also designed to be able to solve ODE systems with an arbitrary number of dimensions. Therefore the code is vectorised to facilitate this. The number of dimensions should be auotmatically detected and accounted for. This is true for all of the implemented methods.

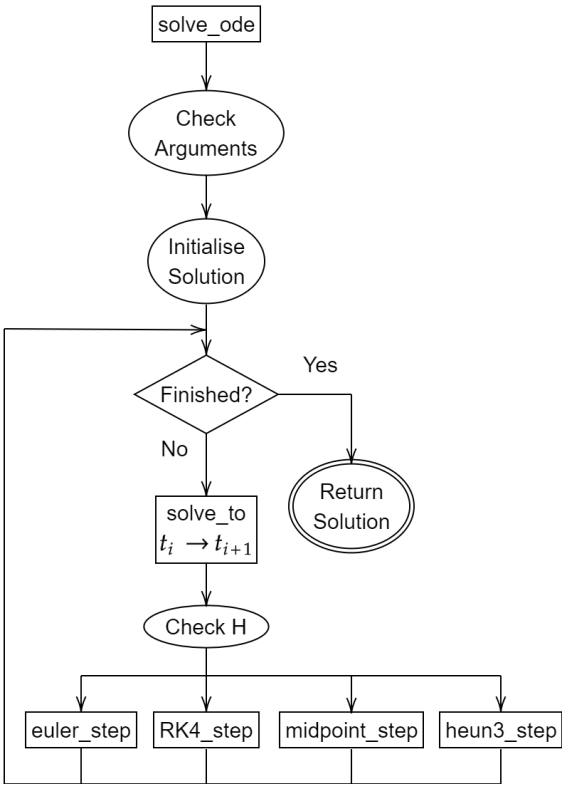


Figure 8: Block diagram showing the `solve_ode` function.

2.2 Numerical Shooting

The main design decisions in the numerical shooting part of the code include the use of the aforementioned ODE solver to allow solutions to be generated to exact time values to facilitate the finding of periodic orbits.

The core of the method is the construction of a root finding problem which takes the form of a `Python` function that is created programmatically using the supplied ODE system and phase condition. This root finding problem can then be solved using the `Scipy` function, `fsolve`. This was chosen for its reliability and accuracy. The 4th order Runge-Kutte method was chosen to carry out the ODE solving because it is the highest order method that is implemented and therefore should provide the greatest speed and accuracy. If the root finder fails to converge, a warning is printed and an empty array is returned. The passing of the phase condition as a function allows easy changing of the condition and even using more complex conditions than those used in the provided examples.

The passing of variable numbers of arguments to all the functions required for the shooting is important since this will form a key part of the next layer to be added to the code, the numerical continuation, where a specific parameter must be selected to be varied.

2.3 Numerical Continuation

The key design decisions in the code responsible for numerical continuation include another example of reusing and layering code. The `numerical_shooting` function is used when the continuation

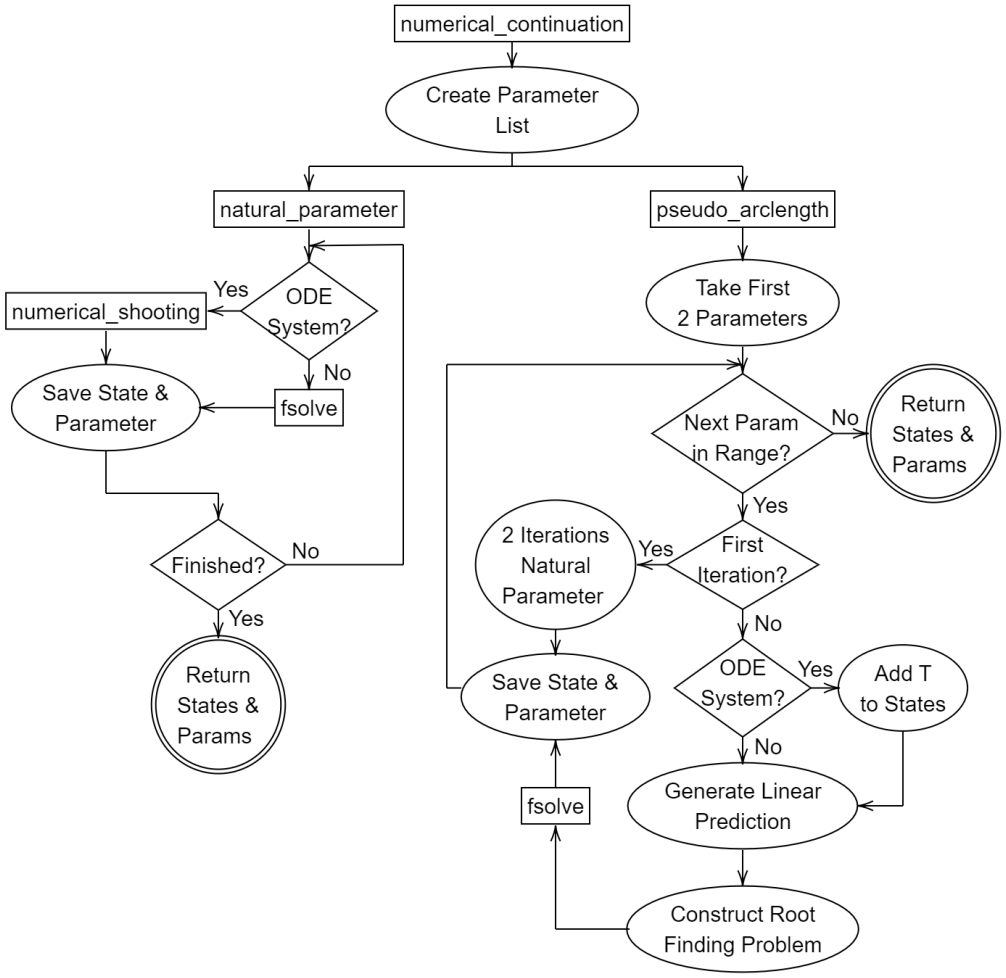


Figure 9: Block diagram showing the `numerical_continuation` function.

procedure is carried out on an ODE system. For pseudo-arclength continuation, numerical shooting forms part of the root finding problem (alongside the pseudo-arclength condition), shown in Figure 9, and for natural parameter continuation it is directly used to find the system states for an orbit occurring for the particular parameter value. This methodology is also shown in the first iteration of pseudo-arclength continuation, where the `natural_parameter` function is used to obtain the system states required to start the pseudo-arclength continuation process.

Another design decision is the continued use of `fsolve` for the evaluation of the root finding problem. This, again, is due to its reliability and accuracy.

For natural parameter continuation, the system states are returned for all of the parameter values provided to the function. However, since pseudo-arclength continuation is able to go around the 'corners' in the bifurcation diagrams, it is difficult to specify the number of parameter values to evaluate the system at beforehand. Therefore, the function takes the first two provided parameter values and will continue to generate new parameter values, based on the linear prediction, until the newest generated parameter value lies outside of the range specified in the function call. This results in, often, having more parameter values used and returned than specified. This makes the

function slower but it ensures that the resolution remains high when the 'corners' are dealt with.

2.4 PDE Solving

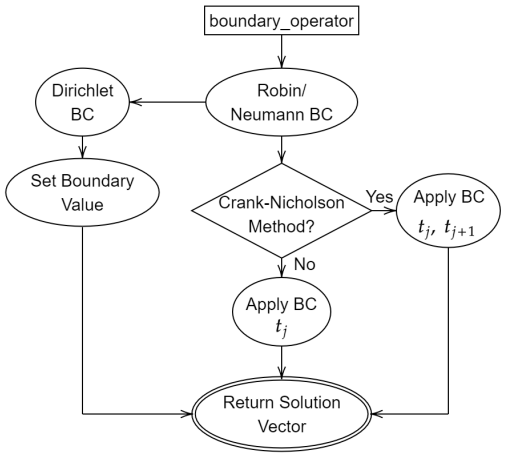


Figure 10: Block diagram showing the `boundary_operator` function.

Much of the complexity that arises when creating a PDE solving function is due to the number of different boundary conditions and right hand side functions and ensuring that all three of the finite difference methods can deal with all combinations of them.

To this end, the function, `solve_pde`, is split into smaller parts so that the minor, but important, changes between the problems to solve can be dealt with more elegantly. For example, The effects of the boundary conditions must be applied to the solution at different points in the algorithm depending on the type of conditions and the finite difference method being used. To simplify this, the `boundary_operator` function was created so that this resulting complexity can be dealt with, without cluttering the code for each of the implemented methods. A block diagram for this function can be seen in Figure 10.

As well as this sub-division, the overall structure of the `solve_pde` function is similar to that of `solve_ode` where each implemented method is comprised of a function that carries out a single step in time that gets looped over. The structure is shown in Figure 11 and it helps with making the code more modular and easy to add new methods and types of boundary conditions to.

As well as the `boundary_operator` function, the construction of the tri-diagonal matrix used for each of the methods is handled by the `construct_L` function. The matrices required for each method are similar, with slight changes, due to the boundary conditions. Separating this functionality makes these variations easier to deal with. As well as this, it made implementing a variable diffusion coefficient much easier. This function is called by each of the three finite difference methods.

In order to solve PDEs with a non-linear reaction term, a root finding problem is constructed programmatically and then solved using `fsolve` for the aforementioned reasons.

`Scipy` sparse matrices were used because the tri-diagonality of the matrices used lends itself well to a sparse format. This also increases the performance when solving systems of equations and carrying out matrix multiplication.

3 Reflective Learning Log

3.1 Mathematical Knowledge

This process has helped me to cement existing knowledge surrounding numerically solving ODEs and allowed me to apply my newer knowledge of software engineering principles to make the process easier and make the code easier to understand and extend.

The theory of numerical shooting and numerical continuation was new to me and the process of implementing the algorithms was very useful in testing my understanding of the mathematical theory. A particular aspect of implementing the numerical shooting algorithm was learning how to

use `fsolve` for a more complex root finding problem. This method came up again when working on numerical continuation and was also key in implementing non-linearity in the finite difference methods for PDE solving.

The work on PDEs has really furthered my understanding of the equations and how they can be used to model real-world phenomena. The visualisation methods that I have implemented have allowed me to easily see the effects of different boundary and initial conditions, providing useful intuition that will be helpful in the near future when I will be solving these types of equations.

I have also been able to learn new, useful skills that can be applied to a range of mathematical scenarios. For example, to generate analytical solutions to PDEs for testing my code, I learnt how to use the PDE solving functionality in Maple which will be very useful in my further studies of PDEs.

In summary, this process has introduced me to the techniques of numerical shooting, numerical continuation and finite difference methods, as well as refining my knowledge of numerical ODE solvers. Through the experimentation and validation of my software, my intuitive understanding of ODEs, and particularly PDEs has increased.

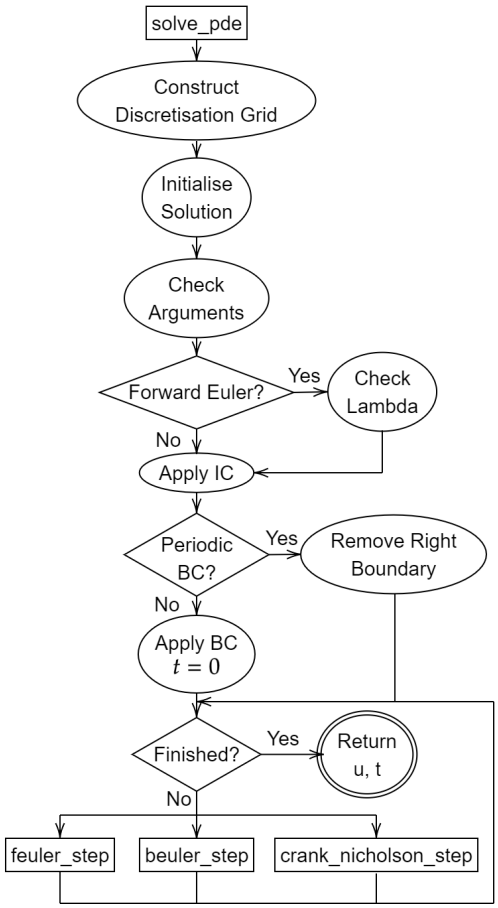


Figure 11: Block diagram showing the `solve_pde` function.

3.2 Software Engineering Skills

My software engineering skills improved greatly throughout this project. Perhaps most importantly, I learnt how to make my code more modular. This was key to making adding new functionality easier. Trying to make more modular code also forced me to learn how to make different pieces of code interface with each other well. An example of this is learning how to pass a variable number of arguments between `Python` functions. Attempting to make more modular code also made me plan out the code extensively beforehand. I had to rewrite my PDE solver almost completely because it wasn't planned out extensively enough initially and became too messy. However, after carefully planning it out, I am happy with the implementation.

Working on one, continuous piece of software also has demonstrated the great importance of proper documentation. When reusing old code to implement new methods, it is vital to have access to documentation. To this end, I learnt how to add proper doc strings to my functions to create a standardised form of documentation.

An aspect of software engineering that was introduced to me through this project is code testing. I learnt how to use `Python`'s built-in unit testing functionality. I found writing code tests very useful and it helped me to discover several bugs as I made changes to the code. In the future I would like to include more tests that test all of the functionality of the code rather than focusing strictly on the outputs. Furthermore, writing the tests in multiple Python scripts will make it easier to keep

the tests organised.

Using Git has been a crucial aspect of this project. My fluency in using Git has increased significantly throughout and I am much more confident that I will be able to apply these skills to future work. This will be especially useful in collaborative work I will be undertaking throughout the rest of my degree and further in the future, when working in industry or research. One particular aspect of Git that I have learnt is that it is better to make several commits throughout the day, after each major accomplishment. These commits can be made with a specific and helpful message and then one push can be made at the end of the work day, rather than pushing each commit independently, as I was doing before.

Writing performant code has not been a strength of mine. However, this process has taught me some easy ways to improve the performance of my code. Particularly, through the use of vectorisation to both simplify and speed up the algorithms. As well as this, the use of `Scipy` sparse matrices, and the performance benefits, was new to me.

This project also prompted me to take on auxiliary tasks such as the creation of a good quality `readme`. I learnt a lot of new skills in markdown, including linking to parts of the repository as well as including example terminal commands. The markdown skills have also been used in the creation of the example notebooks, including the use of Latex equations in the markdown cells. I have come across these example notebooks in other `Python` modules that I have used in the past and have found the interactive nature of the notebook very helpful when studying examples to use the code for a new application. Going forward, I will ensure to create examples such as this in any code I write that is intended to be used by others.

In summary, this process has taught me lots of valuable software engineering skills. Whether this be a better understanding of the `Python` language, such as unit tests, variable number of arguments and lambda functions or more auxiliary skills such as proper use of Git, learning some useful markdown or properly documenting my code. I am sure that these skills will be useful in my future endeavours, both throughout the rest of my degree and beyond.

References

[1] C. White, "Scientific Computing - Numerical Toolbox for ODEs and PDEs," 3 2022.