Scientific Computing

Nathan Browne ge18723
https://github.com/NateBrowne/Sci-Comp

May 2021

1 Usage Standards

In order to use the created ODE Utilities on a differential system, the user must define the differential system as a single function that takes in a time value, a state vector as a Numpy [1] array, and keyword parameter arguments. An example is shown in Figure 1:

```
def lotka_volterra(t, vect, a=1, b=.1, d=.1):
    x, y = vect
    dxdt = x*(1-x) - a*x*y/(d+x)
    dydt = b*y*(1- (y/x))
    return np.array([dxdt, dydt])
```

Figure 1: *A standard setup of a differential system function. The vector is unpacked then manipulated to create the gradient returned as a Numpy array. This particular system is independent of time.*

To use any of the solving functions defined in this software, the user must define a dictionary named sys_params, which is a dictionary of parameters to feed into the system. That is, unless they wish to use the defaults they specified in the function definition. For example, in the function shown in 1, the user may wish to fix their 'a' and 'd' parameters to non-default values. They would do this as follows:

mypars = {}
mypars['a'] = 2.
mypars['d'] = .2

Then they could pass this information in as a keyword argument to any process that takes it. For example:

tl, vl = solve_to(lotka_volterra, 0, 100, [0.1, 0.1], sys_params=mypars)

is a valid function call.

2 Design Philosophy

The main goal in this project is make all functions as general as possible. The pseudo-arclength continuation code will utilise shooting code, which will use numerical integration code, which will use numerical step code. If keyword arguments are handled correctly, adjustments to the arguments

in higher-up functions such as psuedo-arclength continuation will be correctly passed all the way through to the correct dependency functions lower down without needing explicit redefinition.

Where possible, we will also to use class wrapping to better organise our function returns. For example, in the numerical shooting method, the outputs are a set of initial conditions and a measurement of the system's period. If these are returned normally, the user would have to pass something along the lines of:

period, ics = shoot_root(args)

which relies upon them knowing the order of the returns. Thus, a better method would be along the lines of:

shot_solution = shoot_root(args)

allowing the user to now call shot_solution.period and/or shot_solution.ics, which is more intuitive.

Where appropriate, our user will have the ability to preview information graphically by simply activating a plot boolean variable. In a similar way, where possible, we will incorporate measures to help our user assess where errors are occurring by turning on a print_progress variable or a warning variable which, when activated, outputs specific warnings.

The final product is called 'ODE_Utils3.py'. This, along with functions and test files used in this document can be found in 'Test Files' in the created repository [2].

3 Basics

3.1 'euler_step' and 'rk4_step' (4th order Runge Kutta)

The Euler and RK4 step functions require just three arguments. The differential system function, a value for t, and the current state vector v. 'deltat' is an optional argument at default 0.01. Also, should several keyword arguments be inputted, the structure of the final arguments

..., sys_params={}, **kwargs)

means that the system parameters will be automatically unpacked from other keyword arguments. Then, the unpacked version of the system parameters can be passed into the function without any risk of receiving unexpected arguments.

The outputs of both functions are the next time value incremented by deltat, and the next state vector as a numpy array.

3.2 'solve_to' function

Function 'solve_to' is built in ODE_Utils.py. It requires four arguments: a function 'func' to integrate, two time bounds 't1' and 't2' to integrate between, and an initial state vector 'v0' as a Numpy array.

Optionally, a method argument can be specified as rk4_step (default) or euler_step. The user can also set plot=True (default False) if they wish to see a graphical representation of the solve which they can save.

In accordance with the standards we set out in Section 2, the function does not explicitly require any system parameters 'sys_params', or a step size 'deltat', but if these are passed into the function as keyword arguments they will be correctly unpacked into the internal processes or defaults will be used in their place.

The function then returns a Numpy array 'tl', the timesteps for the solve, and a Numpy array 'vl', the state vectors through time.

It is also worth mentioning that two steps have been taken within this function to handle manipulation of the step size. Firstly, it is determined to be logical to re-scale all steps in the integration to equal sizes. So, before any solving occurs, the variable 'deltat' is manipulated according to the following example: We wish to solve between time 1 and time 11 in steps no greater than 0.6. First, we divide the size of the time bound by our step size: (11 - 1)/0.6 gives 16.666... We then take the ceiling function of this value to get 17. Our new step size is equal to (11 - 1)/17 which gives ∼0.58824. Of course, due to rounding errors in Python, this will still not exactly hit 11 at the final step, but it will be very close. This method is exacted in function 'rescale_dt'.

A final conditional statement means that if another step can't fit exactly (due to the mentioned step error), the new step size is set to the exact value that brings our solve to the final bound.

3.3 'solve_ODE' function

The function 'solve_ODE' completes the 'solve_to' routine for a set of specified step sizes. It now has an optional argument 'stepsizes', which must be passed in as a list or tuple object, and contains the step sizes the user wishes to solve for. The defaults are (1., .1, .01). The function can also plot a comparison of the solves by setting plot=True, which is set by default to False. Keyword arguments **params can also be passed in including system parameters, numerical method, etc.

'solve_ODE's returns are then wrapped by a class called 'Solved_ODE', which has two items: estimates and tracings. Estimates is a dictionary the final value of the solved ODE with its step size as the key. Tracings is a dictionary of the full solve between bounds for each step size with the step size as the key, and is required if the user wishes to make their own plots.

```
$ python Solver.py
Final sol by  Euler  with stepsize  1 :  2
Final sol by  Euler  with stepsize  0.1 :  2.5937424601
Final sol by  Euler  with stepsize  0.01 :  2.6780334944767583
Final sol by  Euler  with stepsize  0.001 :  2.7142097225133828
Final sol by  Euler  with stepsize  0.0001 :  2.7181459268252266
```

Figure 2: *Euler method values at t = 1 for function dxdt=x with starting condition x=1. A return from Solved_ODE.estimates*

```
Final sol by  RK4  with stepsize  1 :  2.708333333333333
Final sol by  RK4  with stepsize  0.1 :  2.7182797441351663
Final sol by  RK4  with stepsize  0.01 :  2.7182818282344017
Final sol by  RK4  with stepsize  0.001 :  2.7182818284590224
Final sol by  RK4  with stepsize  0.0001 :  2.7182818284593115
```

Figure 3: *RK4 method values at t = 1 for function dxdt=x with starting condition x=1, as we can see it converges much faster, as expected*

Running 'Solver.py' produces Figures 4 and 5:

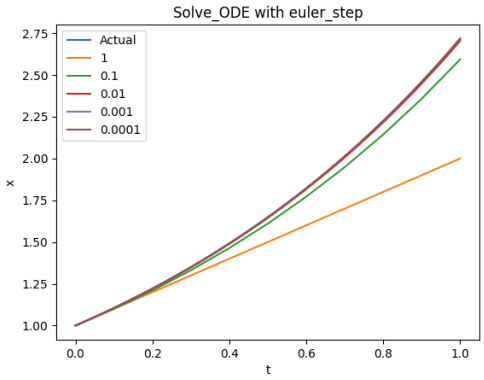

Figure 4: *Comparison of solve for step size using the Euler Method on function dxdt=x*

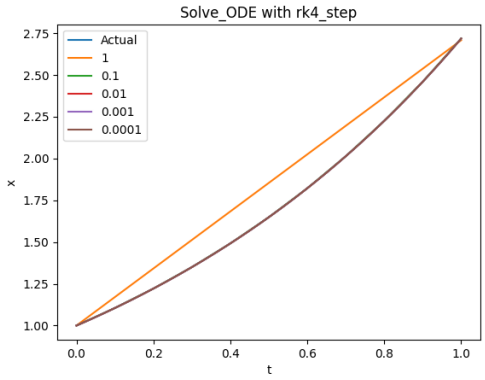

Figure 5: *Comparison of solve for step size using the RK4 Method on function dxdt=x*

3.4 Error for step size

For this comparison, we have chosen step sizes varying by orders of magnitude. As we can see in Figure 6, the 4th Order Runge Kutta routine consistently produces lower error than Euler, but suffers truncation error at the lowest step sizes.

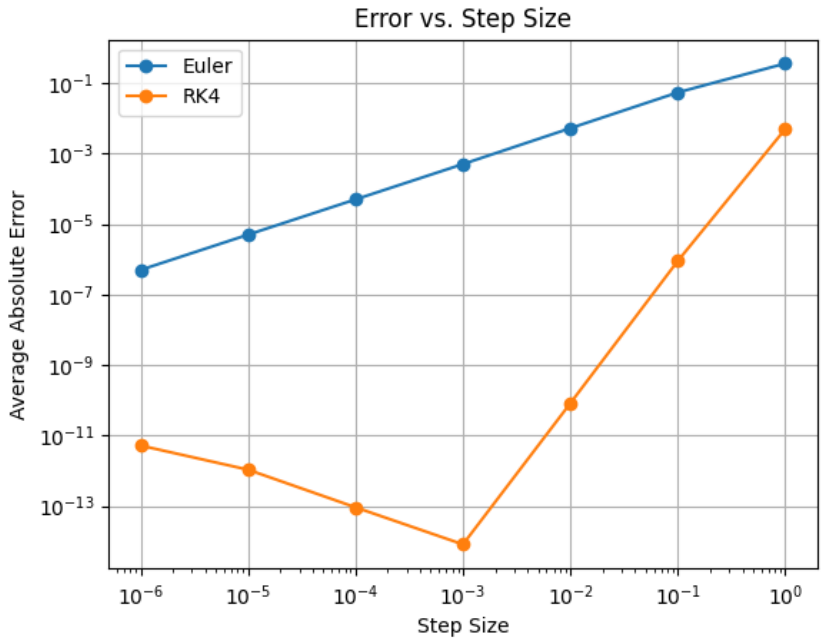

Figure 6: *Comparison of error for step size using each method*

4 ODE Problems

4.1 Numerical Shooting Discretisation

Function 'shooting' is a discretisation method that returns a root-finding problem as a function G. It requires a function, 'func' to discretise and a guess at the initial conditions 'u0t' which also contains a guess at the period. For example, in a 2D system, this u0t would be [u0, u1, t_guess].

Optional arguments include the phase condition equation 'condeq', which defaults to 0, pointing to the first expression in the system. If a user wanted to apply the phase condition to the nth dimension of the system, they would pass n-1. The phase condition 'cond' is the value to set for the phase condition. So, condeq=1 and cond=.5 will set the second dimension of the system's shot initial conditions to 0.5. The default though, is 'extrema', which will set the condition equation to the nearest extremum.

Our integral phi function within shooting requires only system parameters because we are making a call to the user's custom defined function. Any additional arguments will lead to an error. So, a conditional is used to separate the dictionaries and establish 'sys_params' as its own variable.

4.2 Solving Shooting

4.2.1 Preparation - Orbit Detection

With the ability to numerically solve ODE's over a time domain, we now have an opportunity to analyse the solved vectors to find an oscillation period, and the heights of minima and maxima.

Function 'isolate_orbit' identifies peaks and troughs in a Numpy array, and can subtract their time stamps to deduce a period. This is then returned to the user as an 'Orbit' object, with three attributes: Orbit.period, Orbit.max_height and Orbit.min_height.

The function also conducts a number of checks that may give information to the user. If less than 2 peaks are found, the function exits with an error, as no period can be calculated from this. If the relative difference in period measured in the troughs and peaks differs by more than 'peak_tol', an optional argument supplied by the user, they will be warned about it. Also, if the heights of the final two peaks differ by more than 'peak_tol', this will create a warning. These warnings can be turned on or off by setting 'warnings' to True or False respectively.

4.2.2 Shoot Root

Run 'shoot_demo.py' to replicate results.

Function 'shoot-root' is created to solve the shooting problem and identify the period and initial conditions for a particular system by performing a solving step with Scipy's fsolve [3]. Its required arguments are 'func', the function to apply shooting to, and u0, a guess at the initial conditions.

Optional arguments include 'solver' which defaults to Scipy's 'fsolve' function, and 'xtol', the convergence tolerance to be used in the solve. Our shooting discretisation requires a guess at the oscillation period, but this t_guess cannot be entered by the user. Instead, a 'solve_to' call is run between zero and optional argument 'period_find_len'. After solving over this range, function 'isolate_orbit' is used to identify a good approximation of the time period and the initial t_guess becomes Orbit.period.

Also, the user has the option of activating boolean variable 'improve_guess' which will change the initial conditions guess to the last state vector found in the first solve_to step. This point is more likely to be close to the limit cycle.

Then, the user can input a set of keyword arguments such as their system parameters dictionary or phase condition.

By solving the Lotka Volterra system using the shooting method, with an initial u_0 guess of [.1, .1], with parameters set to a=1.0, b=0.1, and d=0.1 the shooting method returns a u_0 of [.0391, .1336] and a period of 34.067. As we can see in Figure 7, the shooting method has set the first equation to a default phase condition, and as such, prey is at a minimum.

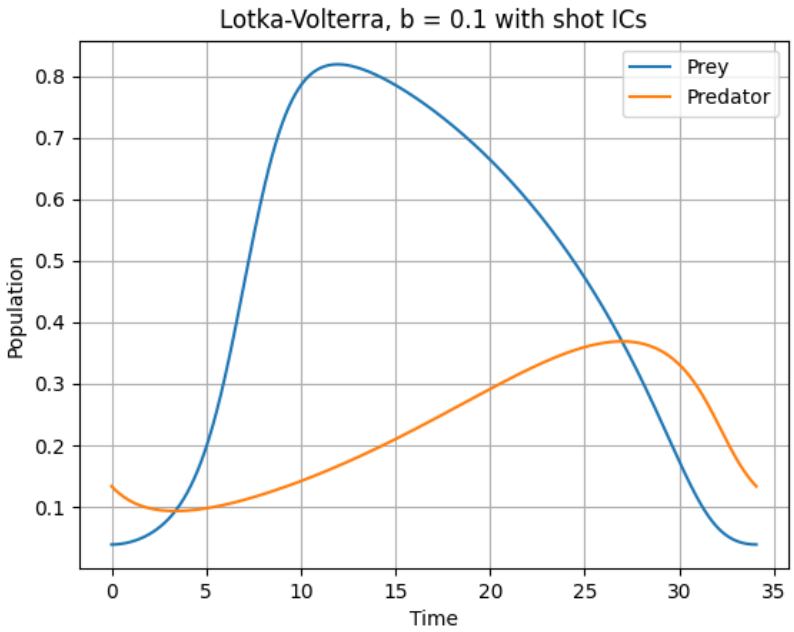

Figure 7: *Lotka Volterra with respect to time from shot initial conditions over its period*

If we plot predator and prey populations against each other now, for the solve over its period, we see that it does in fact form a complete limit cycle, as shown in Figure 8:

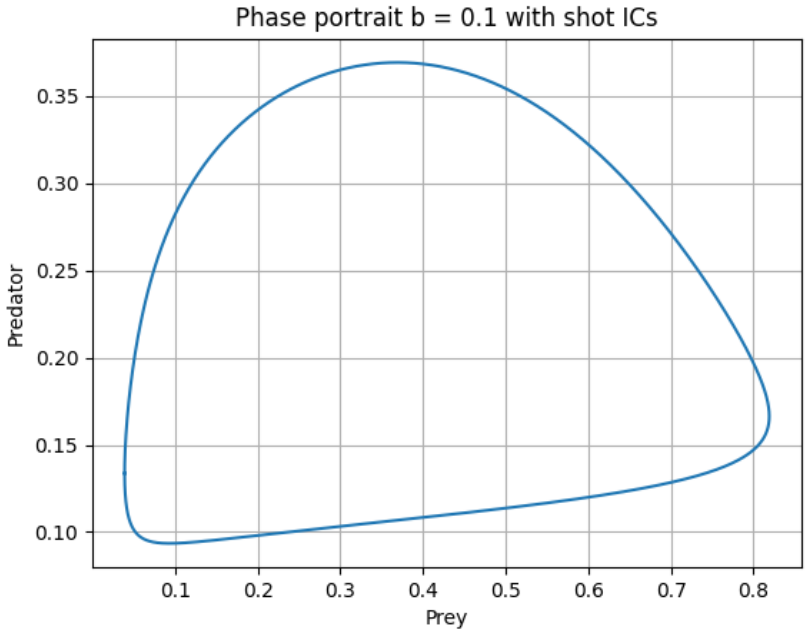

Figure 8: *Lotka Volterra system phase portrait with shot initial conditions over its period*

Should we now decide that we want the predator population to be set to .2 at the start for our phase condition, we can simply set condeq=1 and cond=.2, to produce the tracing shown in Figure 9:

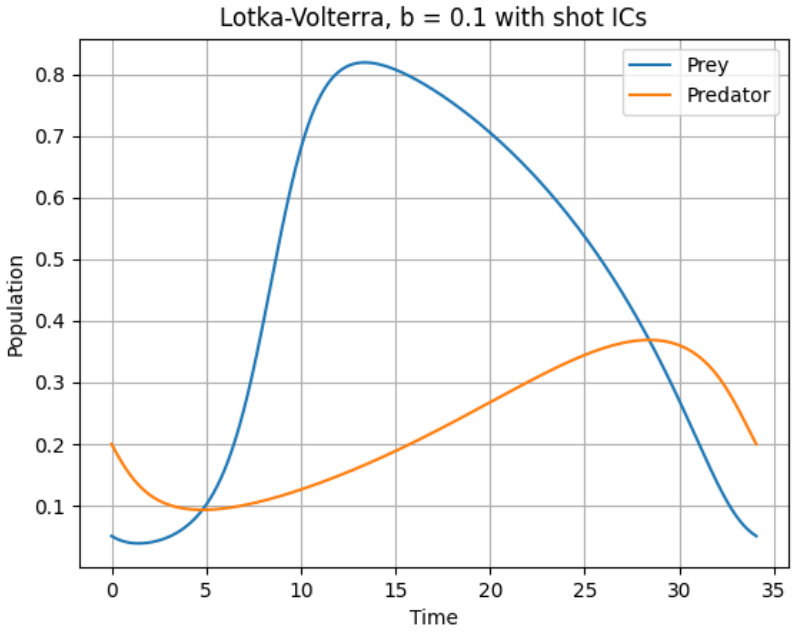

Figure 9: *Lotka Volterra with respect to time from shot initial conditions over its period, this time with a phase condition specified as predators=.2*

This also produces the same phase portrait we saw earlier in Figure 8.

In order the test the code, the following tests were conducted in 'ShootingTest.py':

- Known Solution: A simple dynamic system was solved by the numerical shooting code then compared against an analytic solution. The analytic solution was given as a Numpy array of values, and as such, the closeness of this result with that produced by the shooting code can be evaluated by the Numpy.isclose() function, which tests if each solved point was within a desired error tolerance of the solved solution. We determine 1% is a close enough tolerance but this can be varied. The code passes this test.
- 3D System: The shooting code should be able to analytically solve n-dimension systems. The code is tested on oa 3D system and correctly shoots its initial conditions and period.
- Dimension Error: The code should gracefully reject a user's input if they attempt to solve a system with initial conditions that don't match in terms of dimension. The code passes this test and outputs a warning message to the user.

4.3 Numerical Continuation

4.3.1 Natural Parameter

Run 'natcon_demo.py' to replicate results.

Function 'continuation' requires 5 variables: the function 'func', a guess at the first initial state vector 'u0', the start parameter value 'par0', the same of said parameter 'vary_par', and the maximum parameter value to continue to, 'end_par'.

'step' is an optional argument that declares how much to increment the parameter by each iteration until it reaches the 'max_steps' argument, default 100. Then, we require a discretisation which is, by default, shooting, and a 'solver' to find the root, which is by default Scipy's fsolve solved to a tolerance of 'xtol'.

When shooting the first solutions, 'first_shoot_deltat' is used, but for other solves we may wish to use a higher deltat to speed up computation, so 'contin_deltat' is defined with a higher default value. 'root_change_tol' is used to identify when a solution is the same as the previous, and therefore we should terminate the continuation.

As with other functions, the user has the option of plotting the solve after by setting 'plot' to true.

The user can also activate 'print_progress' to see each step as it is solved, allowing them to spot errors in their own usage if they would like.

The continuation function then returns a Numpy array of parameter values and a Numpy array of solved initial conditions.

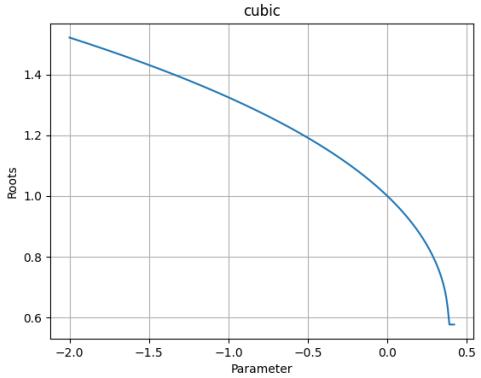

Figure 10: *Cubic function roots by natural parameter incrementing b from -2 to 2.*

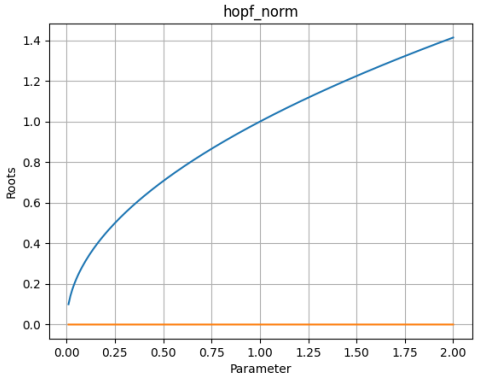

Figure 11: *Hopf normal form initial conditions by natural parameter incrementing b from 0 to 2.*

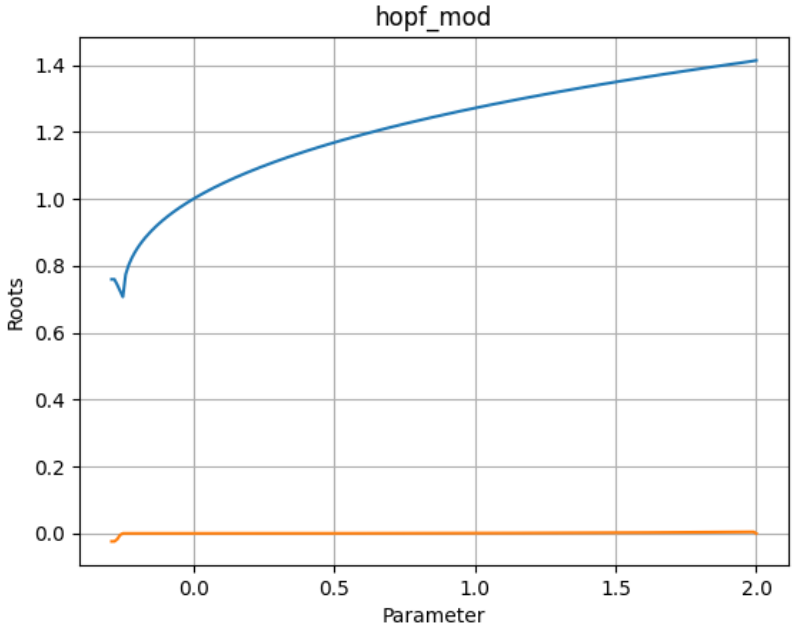

Figure 12: *Hopf modified form initial conditions by natural parameter incrementing b from 2 to -1*

As we can see from Figures 10 and 12, the natural parameter continuation fails at a certain parameter value. For the parameter values of the Hopf normal form in Figure 11, we see that natural parameter continuation is valid.

4.3.2 Pseudo-arclength Continuation

Run 'arccon_demo.py' to replicate results.

Function 'pseudo_arc_contin' requires the same variables as 'continuation', but it now has a boolean optional argument, 'check_shoot'. If set to True, the first known solution needed (found by shooting) will be plotted to check that the continuation is starting from a limit cycle.

In order to formulate the pseudo arclength root-finding problem, we require another discretisation called 'arclen_discret'. This takes in the function to discretise, 'func', the last known solution 'u0tp' which is a vector of the form [u0, ..., un, T, param] for an n-dimension periodic system (if we are not solving for a period, such as in the cubic function, we do not need the T variable), 'u0tp_guess', a guess at the next root found by addition of a found 'secant', and the identity of the parameter to vary 'vary_par'.

Optionally, the user can change the discretisation to use to construct the pseudo-arclength problem, but its default is shooting. 'param_discretise' can be used to transform functions to functions with fewer required arguments so they can be used in solving methods.

Of course, other parameters can also be entered such as the system parameters.

'arclen_discret' returns a function 'root_find_func' which, if solved to zero will give the root value of 'u0tp'.

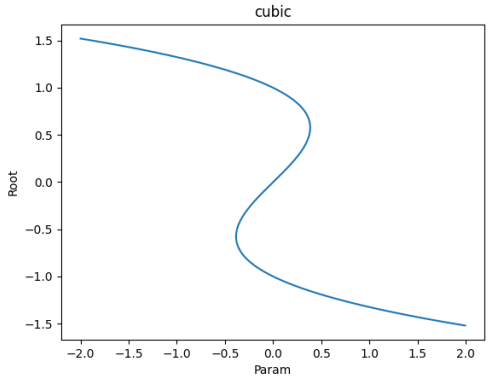

Figure 13: *Cubic function roots found by pseudo-arclength continuation between -2 and 2.*

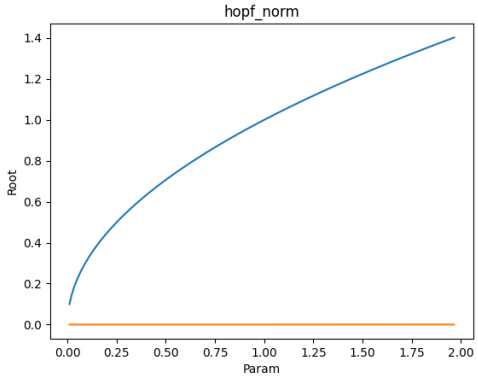

Figure 14: *Hopf normal form initial conditions by pseudo-arclength continuation incrementing b from 0 to 2.*

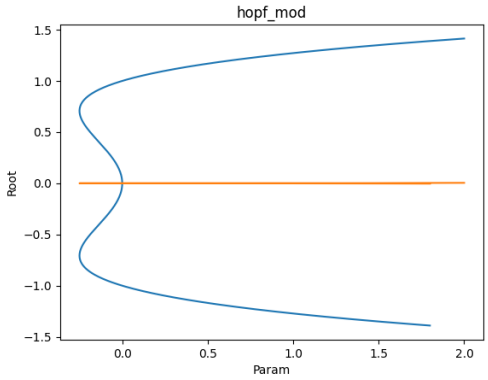

Figure 15: *Hopf modified form initial conditions found by pseudo-arclength continuation for parameter b between 2 and -1*

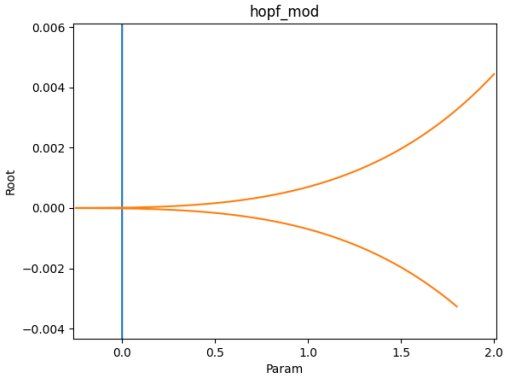

Figure 16: *Closer inspection of Hopf modified showing behaviour of second variable*

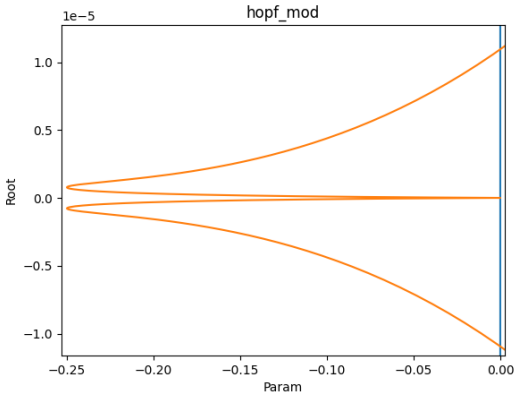

Figure 17: *Closer inspection of Hopf modified showing behaviour of second variable revealing behaviour at the bifurcations*

As expected, nothing has changed for the Hopf normal form in Figure 14. Clearly, pseudo-arclength continuation is required to solve the cubic in Figure 13 and the Hopf modified in Figure 15 to properly identify root bifurcations. Figure 15 appears to show a flat second variable, but closer inspection shown in Figures 16 and 17 show how the bifurcations are also impacting this variable.

5 Partial Differential Equation Systems

Run 'PDEs_demo.py' to replicate results.

Function 'tridiagonal' produces tridiagonal matrices based on a given size, a mesh Fourier number, λ, and a scheme given by the variable 'direction', which can be set to 'fe' for Forward Euler, 'be' for Backward Euler or 'cn' for Crank Nicholson, which is its default. For Euler methods, the return is a single matrix A. For Crank Nicholson, returning two matrices A and B.

Function 'var_tridiagonal' takes in a size n, a diffusion function κ, a vector of space mesh points x, and grid space values Δx and Δt. From these, it can construct a tridiagonal matrix in the Forward Euler or Backward Euler schemes but with variation with respect to the varying kappa.

Function 'solve_diffusion_pde' works as the equivalent of 'solve_to' for ODE systems. It takes an initial state vector 'u_j', space and time domains 'x' and 't', 'mx', the number of points in the space domain, 'kappa', the diffusion coefficient, and 'L' and 'T', the total sizes of the space and time domains.

Optionally, the user can set an extension 'ext' to use Non-homogeneous Dirichlet, Neuman, or periodic boundary conditions. They can then enter these conditions by setting 'bounds' or 'neu_bounds' for the method they wish to use. By default, the scheme used 'direction', is Crank Nicholson. Finally, they may also input a Right-Hand-Side function 'rhs_func' in order to follow that scheme.

The return is 'u_jp1', the next state vector.

Function 'steady_state' finds the the time taken for a system to reach a steady state across a variety of values for diffusion coefficients. It requires the initial state vector 'u_j', the number of points in space and time 'mx' and 'nt', a range of diffusion coefficients 'kappa_range', and the total sizes of the space/time domains 'L' and 'T'.

Optional arguments include ‘tol’, the tolerance between current and previous values from which to determine if a steady state is reached.

Also, the user can activate ‘show_progress’ to see a full breakdown of processes, and activate ‘plot’ to view a graph of the continuation after solving.

Keyword arguments can also be entered to change conditions used in a number of internal processes such as boundary conditions and schemes.

It returns a Numpy array of times taken to reach the steady state.

Figure 18 shows a result from a ‘steady_state’ call on a system with non-homogeneous Dirichlet boundary conditions.

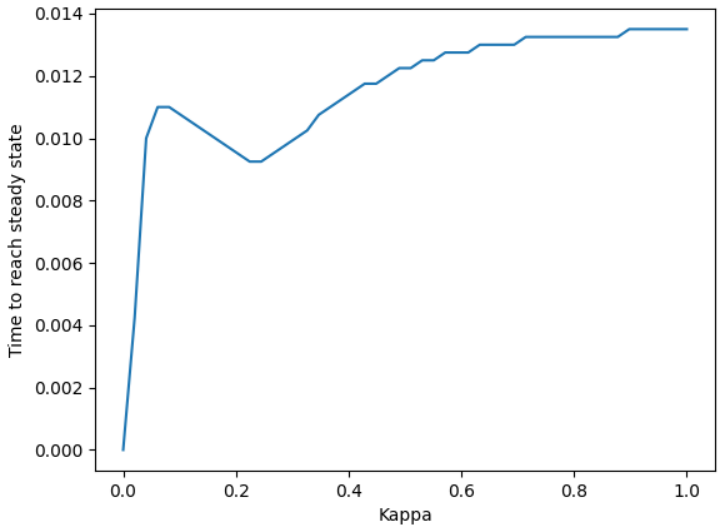

Figure 18: *Steady-state analysis of a diffusion PDE for different diffusion constant values with a non-homogeneous boundary condition.*

6 Reflective Learning

In this project, I have mainly progressed my software engineering abilities. In particular, I have improved in the use of manipulation of optional arguments and code modulation. Also, I now have a much greater understanding of the processes of function discretisation for root finding and vectorising processes to increase speed instead of iterating. I have also learnt about the importance of code testing and construction of testing software.

This experience will be useful in every program I write from now on, as I can have new tools to make my code more intuitive to use, and much shorter. In the long term, I will work to make my code more informative to the user, providing more errors and warning messages wherever they may occur. As well, it is useful to have experience in writing informative docstrings.

If I were to repeat the project over again, I would aim to generalise code even further. For example, my natural parameter continuation and pseudo-arclength continuation are different functions, but they could be put together given they take similar arguments.

It is also worth mentioning that there are three 'ODE_utils' files. Starting over and over again was not helpful, so in future I will work to make my code more robust from the earlier stages upward. 'ODE_utils3' was created.

References

[1] Charles R. Harris, K. Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585:357–362, 2020.

[2] Nathan Browne. Nathan browne scientific computing. `https://github.com/NateBrowne/Sci-Comp`, 2021.

[3] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.