Scientific Computing - Boundary Value Problems

Jack Healey

November 14, 2019

University of Bristol

1 Summary of Software

1.1 Format

The code is divided into many functions, which gives the user the ability to easily debug an error should one arise. Each function has its own task, so the process of solving a particular ODE is easy to follow. The outcome is a function named *display*(), that ties the methods together, where the functions called in *display*() are called in the same order they are listed in the code, chronologically. It is good practise in coding to avoid using global variables, as for larger pieces of code, one's work is harder to read and maintain. For this reason, the file num_shoot.py contains no global variables. This also makes the code adaptable if the user wished to make minor alterations, as global variables restrict the code to the exact use in which the author originally intended it.

1.2 Input

The num_shoot.py script assumes the user has a certain level of prior knowledge about the ODE they wish to solve. To produce an accurate result, the *display*() function must be fed:

- A periodic ODE - an acceptable example of this is the Hopf bifurcation normal form ($\sigma = -1$):

$$\frac{du_1}{dt} = \beta u_1 - u_2 + \sigma u_2(u_1^2 + u_2^2) \tag{1}$$

$$\frac{du_2}{dt} = u_1 + \beta u_2 + \sigma u_1(u_1^2 + u_2^2) \tag{2}$$

- A reasonable guess of the initial conditions
- A parameter value that provides a stable solution

The phase condition is set in the function $func()$, and is set initially as $\frac{du_2}{dt}(0) = 0$. As the phase condition is dependant on the ODE, the user should edit this in order to adapt it to their specific problem (see USER INPUT: in $func()$).

1.3 Functionality

With the necessary inputs, calling the function *display*(X0, ODE_func, par) initially solves the equation using the initial conditions X0, parameter value (par), and the in-built python integrator 'odeint', over a given time span that can be manipulated in the function *tpoints*(). This given solution is not an adequate result as of yet, but gives us an accurate result for the period of the function. This period is calculated by using the function *period_calc*(), which highlights the minima of an oscillating function, and uses the time-points of the minima to produce the period (T).

The next step is to satisfy the periodic boundary condition, $u(0) = u(T)$. The python root-finder $fsolve()$ was used in order to meet this condition. This function was used to produce a new set of initial conditions that met not only the periodic boundary condition, but also provided a phase condition of $\frac{du_2}{dt}(0) = 0$. These conditions are defined in the function $fun()$, where the sole use of this function is to be inputted into $fsolve()$.

Having achieved these objectives, the function *display*() inputs the new initial conditions into the python 'odeint' solver, providing an accurate solution of the ODE, starting at an appropriate phase condition. This can then be visualised in a plot, as seen below.

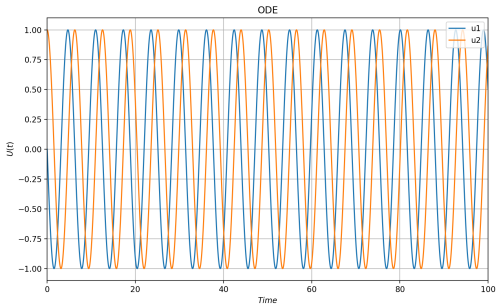
As the parameter value β is varied in the Hopf bifurcation normal form, the behaviour of the solution changes. Natural parameter continuation is performed in the function *param_cont*(), where the limit cycle is analysed for a range of paramater values.

1.4 Errors

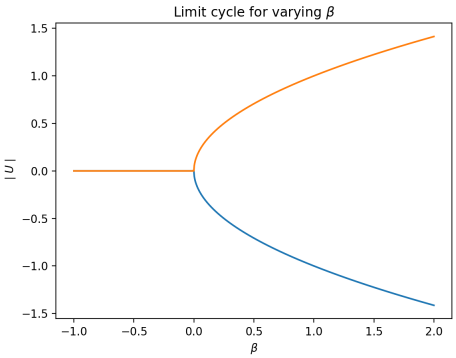
It is known that certain functions will cause errors when inputted into an oscillatory ODE solver, and inputs that do not have a solution should be handled gracefully. If the user inputs an ODE that is not oscillatory, the user is told 'Attempted ODE is non-periodic' in an error message. Without this message it might be unclear to the user the exact nature of their issue. If the user inputs initial conditions that do not match the dimensions of their input ODE, they are met with the error message 'Dimensions of initial values don't match those of the ODE', informing them that there has been an input error in setting up their solution. The test script is built for more explicit errors. For the same set of equations as above, an exact solution is known. These known solutions can then be tested against, and if an error shows here, it is likely the solution has been tackled incorrectly, despite correct inputs.

1.5 Plots

The outputs of the *display*() and *param_cont*() functions give the user the ability to visualise the results. Should the user wish, two plots can be produced to show the relationship between u_1 and u_2 in this case, and also the relationship between the solution and parameter β.



Plot of the Solved ODE, $\beta = 1$



Bifurcation plot

2 Key Software Design Decisions

2.1 ODE integrator

The first major design choice occurred when deciding which integrator to use. Three options were considered initially, the Euler method, the Runge-Kutta 4th order method, and the in-built python integrator 'odeint'. These options were analysed as they have appeared most frequently in past tasks of a similar nature. Through this past experience, it was known that the Runge-Kutta method was preferable to the Euler method due to its greater accuracy and efficiency. The point at which this method is most efficient is when the step-size is 1e-3. Therefore, in making this particular design decision, the Runge-Kutta 4th order method (h = 1e-3) was compared to the python 'odeint' solver. In terms of accuracy, the Runge-Kutta method performs better, however the 'odeint' method is sufficiently faster, by a factor of more than 10. Despite the 'odeint' solver being less accurate, the accuracy of 1e-8 is more than sufficient for the problems set out in this coursework. The gains in efficiency here out-way the losses in accuracy, therefore, it is beneficial to use the 'odeint' solver.

2.2 Phase condition

The phase condition is used to initiate the solution. For a known solution, a phase condition in the form of

$$u_2(0) = 0.4 \tag{3}$$

could be adequate. However, when solving a set of ODEs over varying parameter values, the limit cycles may never pass through the set initial point, here 0.4. As the functions we are working with are oscillatory, a phase condition in the form of

$$\frac{du_2}{dt}(0) = 0 \tag{4}$$

would be more appropriate. All oscillatory solutions have minima/maxima, therefore a solution should always be found even if the parameter values were changed.

2.3 Tests - Test Script

The test script's use is to test the code against known solutions. Therefore, it is key to define what conditions must be passed in order to produce accurate results. The first condition in the test script is that the solution should fit a periodic oscillation, where we know this condition must be true for a correct solution as it is a preassigned necessity through prior knowledge. Therefore, the first test the solution must pass is:

$$\underline{U}(0) = \underline{U}(T). \tag{5}$$

We must also check that the results produced match the explicit solution. In this case, the ODEs that have been inputted into the solver have exact solutions of

$$u_1(t) = \sqrt{\beta}cos(t + \theta) \tag{6}$$

and,

$$u_2(t) = \sqrt{\beta}sin(t + \theta). \tag{7}$$

As the phase parameter is set as $\frac{du_2}{dt}(0) = 0$, θ can be set to zero, where the test script analyses the magnitude of the result of the coded solution against the magnitude of the known solution at some point t, where these should be equal. Therefore, a value of t can be inputted into the equations above, and using the tspan input into the solver, the difference between the results produced and the explicit solution will be given, providing the second and third tests.

2.4 Errors

When working through the solutions for the ODEs, certain errors appeared, meaning it would be valuable if the code handled such errors gracefully, and provided some information as to why these errors occurred.

The first case in which this was dealt with was where an ODE was inputted into the solver that was not periodic. This was discovered as the period (T) could not be calculated due to the absence of minima. This is an important error to highlight when attempting to solve oscillatory ODEs. When this arises, the error statement 'Attempted ODE is non-periodic' appears, informing the user that there was an error in calculating the period of the ODE, due to the fact the ODE is non-oscillatory. The use of the TypeError here also allows to solve the bifurcation of certain solutions after they have broken down and are no longer oscillatory. This can be seen in the bifurcation diagram above for $\beta < 0$.

The next error checks for when the provided initial values do not match those of the ODE dimensions. When the initial values provided are of length 3 for an ODE with 2 dimensions for example, the error message 'Dimensions of initial values don't match those of the ODE' is provided, giving the user a stronger idea on how to fix their error.

3 Reflective Learning Log

What did I learn about the mathematical algorithms?	I am now able to greater appreciate the importance of the initial conditions in approaching a solution, where a phase condition based on the derivative of a function is a more successful, general initial condition. In addition, when solving BVPs with code, accuracy is not the only considerable factor, as the 'odeint' function is beneficial due to its much greater efficiency than the Runge-Kutta 4th order method.
What did I learn about software engineering?	In coding projects prior to this, the code and results have taken precedence over user interface. That has changed after this unit, as it is vital the user is able to understand how to use the provided code when adapting your generalised code to solve their own ODE. Similar to that thought, error messages to help the user debug their errors are of use. General errors can be difficult to decipher and these messages can help one understand what might be the reason for a certain fault. I also learnt the benefits of testing framework to maintain performance throughout the process of editing code, as well as using version control software such as Git to track my progress and resort back to previous commits if necessary. Furthermore, I have improved the functionality of my code, in terms of using no global variables and the modularisation of my code.
What are the implications of what I've learnt?	In the short term, this project has given me improved coding abilities, with a wider focus on usability and functionality, along with accuracy of results. These skills are easily transferable to other modules such as Mathematical and Data Modelling, where BVPs are also addressed. In the long run, these skills improve my employment prospects as I am now able to handle more complex coding tasks with a more professional structure in terms of providing a client interface.
What would I have done differently if I started the unit over again/in the future?	Although a strong relationship with Git was maintained throughout, the commits could be more detailed. A third party user may not understand the syntax in some of the commits, as they relate to specific updates made to the code. To prevent confusion in this case, a commit could explain generally what has been altered, and then give more specific details on changes to certain functions. When adding a new function, a docstring should be produced straight away to detail its functionality. It is easy to get carried away with coding and lose the user interface, so in the future it would be beneficial to maintain this balance.