Jose Ramon Batiller Scientific Computing

Boundary Value Problems and Numerical Continuation

Differential equations are usually solved by being given the initial conditions for x (and all its derivatives) and integrating forward numerically. Boundary value problems (BVP) are more difficult and handled differently. An example of a BVP is a chord, or a wire held stationary between two points. There are many forces acting on the wire, however, the end points of the wire will always be stationary. Another example, and the focus of the course, is the duffing equation with periodic boundary conditions. The duffing equation is a non-linear second order differential equation with damping and a driving force.

$$\ddot{x} + 2\xi\dot{x} + x + x^3 = \Gamma\sin(\omega t)$$
$$x(0) = x(T)$$
$$\dot{x}(0) = \dot{x}(T)$$

Two methods of solving were proposed, shooting and collocation. Shooting was the easier of the two, where an initial guess was made and allowed to integrate forward one period, T. If the start and end points didn't match, then initial guess was adjusted. The process was repeated until a guess with matching end points was found. The shooting code can be found in the file **Duffing.py.**

Collocation on the other hand was slightly more complex. It required the construction of a Chebyshev differentiation matrix and it compared an entire series of time points all at once. The collocation code is in **collocation.py.**

Aside from BVP, numerical continuation was also attempted. The behavior of x (or x_{max}) as a parameter, k, changed was examined. Natural parameter continuation was simply incrementing k by a small step size followed by finding x, usually through root finding. This method was severely limited especially when dealing with non-linear systems like the duffing equation since it cannot handle bifurcation points. A code that uses pseudo-arclength continuation was implemented to better handle the bifurcation points. Several versions of this code can be found in **continuation.py** and **lasttry.py**.

A. Duffing.py

Being given access to both the NumPy and SciPy libraries, arrays and odeint became integral tools to use. But to implement the shooting method, the duffing equation had to be rearranged into a form python could understand since odeint could only integrate first order differential equations.

$$y = \dot{x}$$
$$\dot{y} = \ddot{x} = \Gamma\sin(\omega t) - 2\xi y - x - x^3$$

These were initially written as two different badly named functions, dudt and d2udt2. Using arrays, both equations could be combined into one function. The **duffy** function accepts u0 which is

an array of x and the first derivative of x or [x,v]. The function will also return du, an array of the first and second derivatives of x or [v,a].

The duffy function was integrated using odeint by giving random initial conditions over a long time. The plot showed a 'chaotic' start, that eventually gave way to something periodic looking. An easier way of visualizing this was by plotting the x and v in phase space. The phase diagram did show that over time, the oscillator did settle into a definite orbit.

An attempt at isolating a period T was done by picking a sufficiently late time step (t ranged from 0 to 1000 over 1,000,000 time steps, a time step of 750,000 (t=750) was chosen) and then identifying the other time steps that had sufficiently close x and v values (np.sqrt(np.dot(A,A)) is below tolerance where A is the difference). This method sometimes picks up neighboring points because the time steps were too small, and the tolerance was too high. But it worked well enough to verify that the period did follow the equation $T = 2\pi/\omega$.

To find the periodic boundary conditions, the endpoints of the integral over one period had to be the same. The function **start_end_diff** accepted the initial conditions, bundled together as u, integrated forward one period and returned the difference of u and the last point in the integration. This was the function to be zeroed in the root finding algorithm. An important thing to note was that the time variable was initially written as [0, T]. This was later changed to [t, t+T] so that intervals like [-1, 1] could be used and the shooting algorithm results could be compared with the collocation results.

A few hours were wasted trying to make a Newton/Secant method from scratch before realizing that SciPy already had multiple root finders. Eventually **shooting** used fsolve, mostly because it outputted ier indicating convergence or failure. Fsolve needed all the parameters bundled into a list and passed as args=(pars,). The hanging comma was needed to make pars into a tuple. An alternative using lambda functions was equally clunky, so the hanging comma was left as is.

The shooting code could also accept any ode since the ode is one of the parameters passed to it. Both the mass spring ode and $\dot{x} = \sin\pi t - x$ were used successfully in the shooting method. Note that while fsolve prefers one dimensional arrays this should not be an issue for shooting. This was, however, a big issue in collocation.

The periodicity of the values returned by shooting were verified using **plot_stuff** a function that plots x and $\dot{x}$ vs t as well as a phase diagram. As the number of parameters being passed from function to function started increasing, it also became important to have a certain predetermined order in mind when making said functions. Some exceptions had to be made (start_end_diff needed x to be the first variable) but otherwise the order function, x, time, pars was used, even into the other codes.

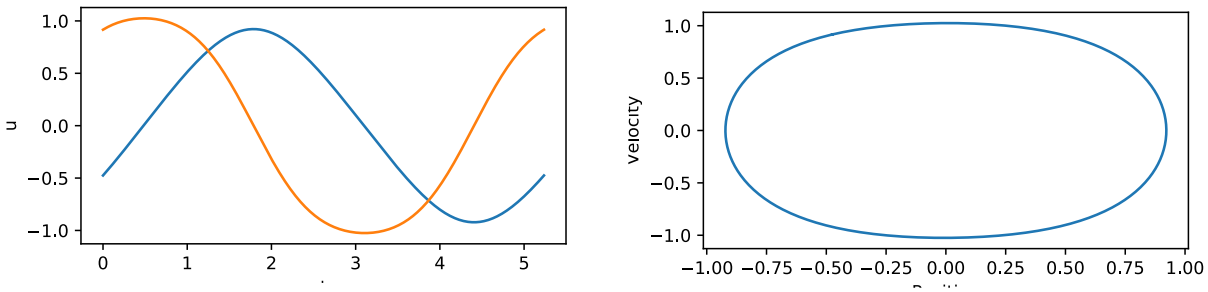

Figure 1 Sample output of plot_stuff function

Although this wasn't done, if the code were to be extended to accept odes with unknown periods then a separate period finding function would have to be made. This might be achieved by approximating the function as a Fourier series using numpy.fft.

B. Collocation.py

The second method for solving periodic boundary problems involved using the Chebyshev differentiation matrix. Given a differential equation, its solution can be approximated as a polynomial. It won't be accurate everywhere, but it can be made accurate over several time points given by

$$t = \cos\frac{\pi i}{n} \text{ for } i = 0,1,2,\ldots,n$$

as a polynomial, differentiation becomes a system of equations that can be solved using the Chebyshev differentiation matrix, D.

$$\dot{x} = f(x,t) \approx Dx$$

To enforce periodicity, the first element of x was set to equal to the last element of x. One thing to note was that the period of the function was strictly set to 2. This is fine since time can be scaled most of the time.

The differentiation matrix was generated using the **cheb** function that would accept any positive integer as input. It would then generate the appropriate matrix of the specified order. Matlab code was provided as a starting point, however, not being well versed in matlab, another source was used. A pdf of chapter 8 of the book "*Finite Difference and Spectral Methods for Ordinary and Partial Differential Equations*" by Lloyd N. Trefethen shows a way to generate the elements of the matrix, D_{ij}. Using the given formula, **cheb** used two nested for loops, to represent i and j and the elements of the matrix were filled up one by one (C++ being my first language). The file week5_chebtests.py contained some tests to compare the generated differential matrix with other matrixes of known value. Although **cheb** passed all the tests, it operated in a very "unpython" way.

The function **cheb2** was made to mimic the given matlab code. There are some differences with how python handles arrays vs matlab. One main difference is when you generate a 1-dimensional array in python composed of n elements, it is of shape (n,). It does not have a shape of (1,n) until specifically reshaped. That means when the array is transposed, it will not have shape of (n,1) but rather it will remain (n,). This can be useful sometimes since python automatically interprets the * operator between arrays as a row by row multiplication. When cheb2 was tested, it also passed all the given tests.

Week5_odetests.py initially contained one working ode test and a 2nd test that contained a 2nd order differential equation. That test was edited to include 2 more simple ode equations just to prove that the collocation function worked. The tests called a collocation function and supplied it with the ode, the order of the matrix, the time intervals, and the parameters of the equation.

The **collocation** function simply got the necessary parameters and shoved them into fsolve, where the function fsolve was zeroing was called **diff.** Diff took the guesses, ode, time and pars and not the order n. The order n could be found by getting the length of the time array and subtracting 1.

The guesses would be multiplied to the differential matrix D and this Dx should have the same values as f(x,t) so diff would return the difference of the two.

It is interesting to note how **diff** enforces the periodic boundary condition. Unlike shooting, which gets the difference of the first and last point, diff would simply set the first value and last value of x to be equal before multiplying with D. This avoids needing to add another equation into fsolve.

The collocation code passed ode test 1 and the other two made up tests 3 and 4 easily. But ode test 2, the second order one, proved to be more difficult. Initially fsolve kept complaining about a shape mismatch between the inputs and the output. This was because fsolve flattens the incoming arrays making the input (21,2) array into (42,0). This was rectified by making a **deflatten** function.

Deflatten compares the length of the supplied array with the length of t. If they were equal, it would do nothing to it. This avoids reshaping it leaving it as a (n,) array. When the lengths were not equal, and assuming an array of appropriate size was used as input, deflatten would reshape the array back into its original shape, using the length of t as reference. In hindsight, just supplying it with the order n could be better, but Diff, the function deflatten is nested in, does not accept n as a parameter. Dealing with a two dimensional matrix, several operations in **diff** needed to have transpose() added. This was where transpose having no effect on (n,) arrays was very beneficial and finally resulted in collocation passing all 4 ode tests.

C. continuation.py and lasttry,py

As the name suggests, several different trials have been made to make a working pseudo arc length continuation function that is sufficiently abstracted to work with different types of equations, discretization's and solvers. This has been especially frustrating since shooting and collocation need different inputs, and have different outputs. For second order differential equations, shooting needs an initial guess, u, of shape (2,) and it will return the values of x and $\dot{x}$ bundled together at some initial time. Collocation, on the other hand, needs an initial guess that has a shape of (n+1,2) where n is the desired order. It will then return the entire function of x and $\dot{x}$ at the necessary time intervals. This mismatch of inputs and outputs made it difficult to code something that can work with both. Not to mention that there was a third discretization method lambda x:x. Instead, the code **continuation.py** works for lambda x: x while **lasttry.py** has a code that works for shooting. An attempt was made to make a code that works for both shooting and collocation by truncating the output of collocation to only return the first element. Its remaining output could be generated by calling odeint at the different time intervals.

The code **continuation.py** was written to handle functions of x and uses lambda x:x as discretization. Using pseudo-arclength continuation it would return the root of x for different values of k. This was done by creating an array z that housed both k and x. The **continuation** function would **initialize** the first two points of array z. This was done using natural parameter continuation. From the third point onward, a **predictor** function would predict where the next point would be given the previous two points (packaged together as prevz which takes the last two rows of z). and then a **corrector** function would use the predicted value as its initial guess. The corrector basically throws

f(x) and **is_tan,** a function that computes for the dot product of (z-predz) and dprevz, into a root solver and the returned answers are placed at the bottom of the growing array of z.

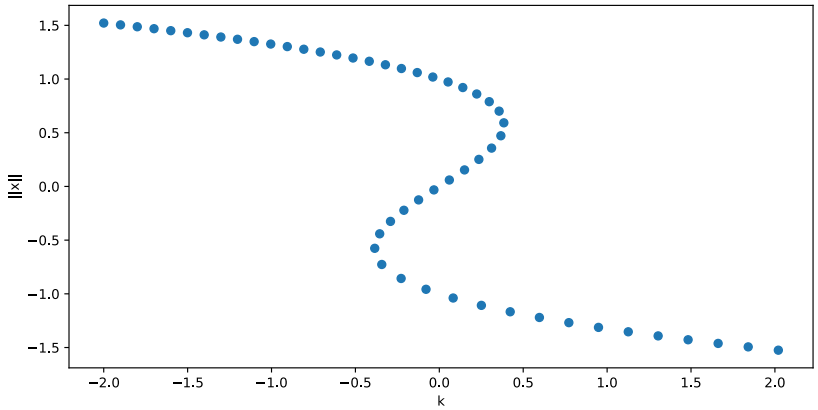

Figure 2 Plot of x vs k of the simple cubic function x^3-x + k

Things became more complicated when ODEs were involved. Both shooting and collocation methods returned values of x and $\dot{x}$ so the code had to be adjusted for arrays. Several badly named trial codes were made namely retry.py, rerety.py, and rereretry.py. All of them functioned similarly. They would bundle k and x into some array z. The use of pythons packing and unpacking functionality (ex. using k, *x = z) ensured that it could handle inputs of differing sizes. Initializing and predictor worked the same as before. **Corrector** this time would call on the chosen discretization function (in this case, shooting). Shooting would call on **start_end_diff** and this is where it would apply the periodic boundary conditions and **is_tan** would be shoved into fsolve.

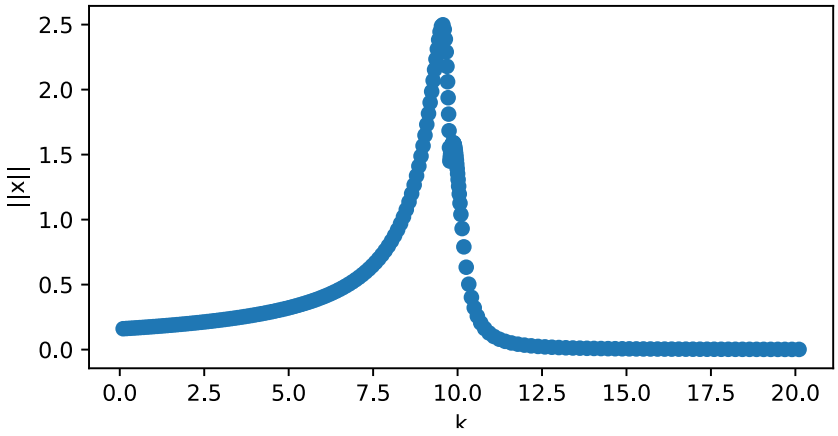

Figure 3 Plot of x_{max} *vs k*

The array z now contains the parameter k and the values of x and $\dot{x}$. A simple for loop would then compute for the maximum value of x for each parameter k over the period. The resulting plots

were always the same and would always fail at the peak around k=9.5. Even making the initial step size small couldn't get around the peak properly.

The last code, aptly named **lasttry.py**, had a different way of thinking and dealing with the problem. The previous trials aimed to get the starting points of a period in the time interval [-1, 1] and then it would find x_{max} later on. Lasttry.pt would instead find x_{max} right away. The plots x_{max} vs k basically shows how the amplitude of a periodic function changes as the parameter k changes. The time period [-1, 1] wasn't important, any period would do.

Lasttry.py aimed to solve $F(k, t, x) = 0$ where F is some function of k , t and x (x had to be last so that packing *x would work for any size of x). Having three unknowns the solver had to be fed three equations to zero namely

$$\text{is\_tan}(\text{k}, \text{x}) = 0$$
$$x(\text{t}) = x(\text{t} + \text{T})$$
$$\dot{\text{x}} = 0$$

The first equation is the same as before and ensures that the new point is along the line tangent to the predicted point and the old points. The second equation is the periodic boundary condition, but it is not strictly bound to [-1, 1]. The solver can freely find any period of the function. The last equation $\dot{\text{x}} = 0$, ensures that $abs(x)$ is the maximum value.

The code still had trouble getting around sharp curves so adaptive step size was implemented. A multiplier, ds=1.2, was introduced to the predictor step. This multiplier would then be halved whenever corrector returns a non-finite value and returned to 1.2 once the corrector finds a converging solution. Once adaptive step size was implemented, and the proper parameters set, miraculous plots could be made.

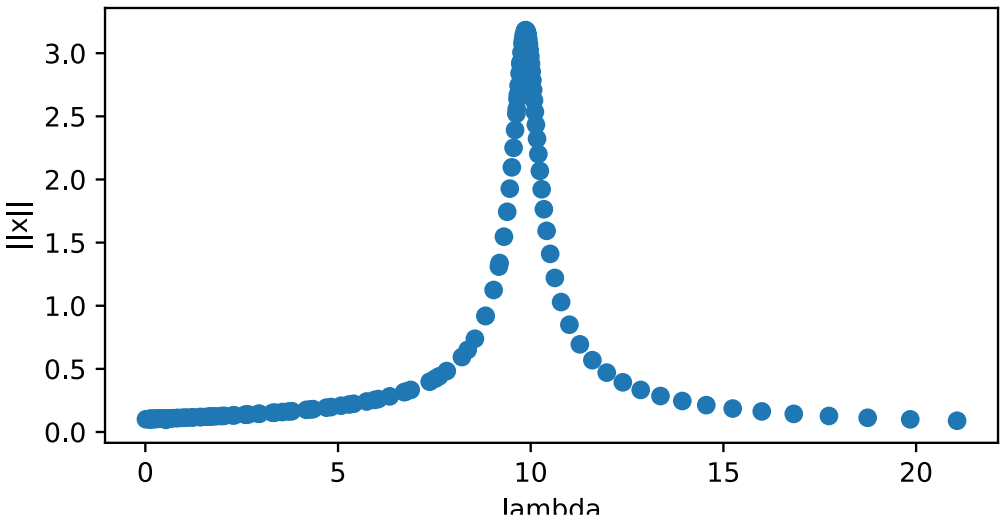

Figure 4 Mass Spring Damper

$\ddot{\text{x}} + 2\xi\dot{\text{x}} + \text{kx} = \Gamma \sin \pi\text{t}$ *where Γ=1, and ξ=0.05*

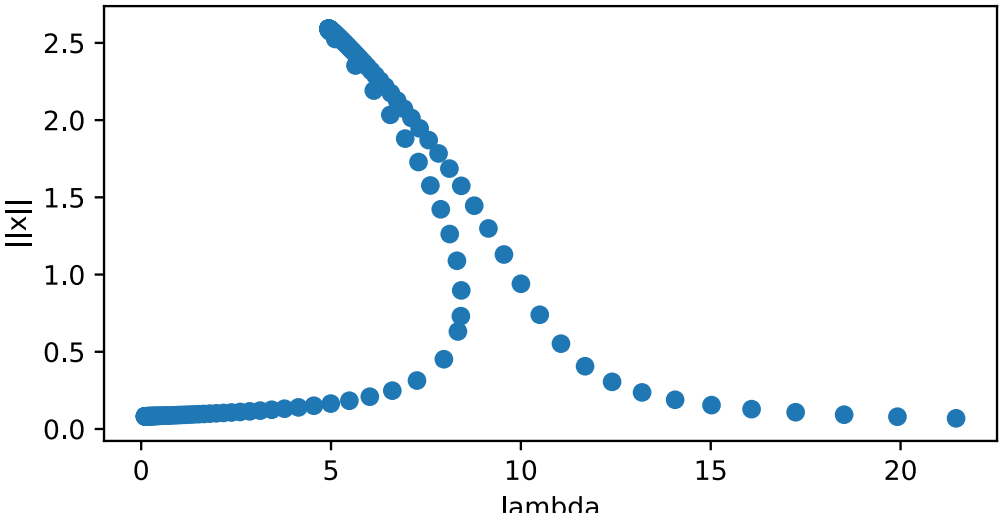

Figure 5 Duffing Equation

$\ddot{x} + 2\xi\dot{x} + kx + \beta x^3 = \Gamma \sin \pi t$ *where Γ=0.8, ξ=0.05 and β=1*

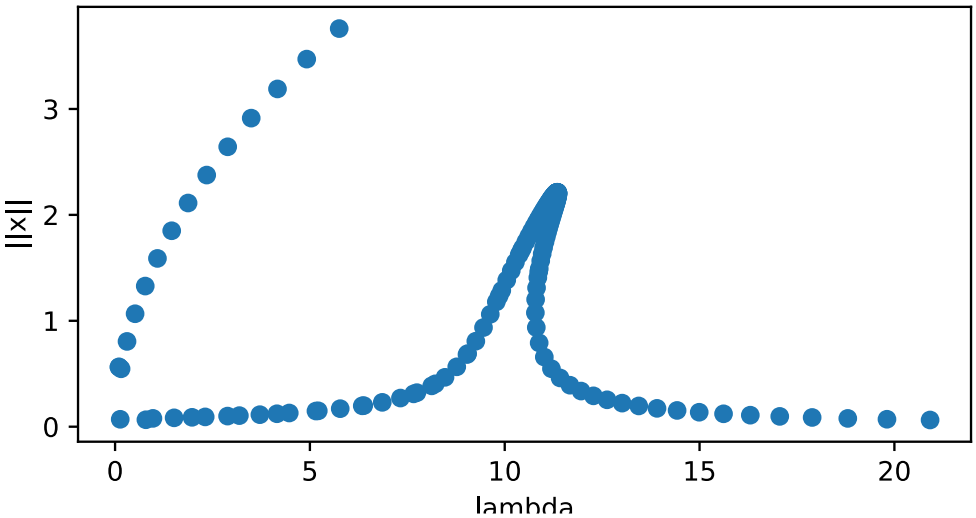

Figure 6 Duffing Equation

$\ddot{x} + 2\xi\dot{x} + kx + \beta x^3 = \Gamma \sin \pi t$ *where Γ=0.&, ξ=0.05 and β=-0.4*

D. Conclusion

The entire continuation code was an exercise in understanding and using arrays. Having never coded using Matlab and my previous languages being C++ and java, I would always resort to using nested for loops. Taking advantage of the properties of arrays make the resulting code cleaner and more elegant.

The numpy and scipy libraries had many different tools and methods that could be used. Numpy had an element for matrixes, but I found that arrays were easier to use in general. One benefit of matrixes was that the * operator would function like matrix multiplication but np.dot(array1,array2) could also achieve the same effect. Scipy had different root finders like newton or root, and more reading into the documentation would be needed to cover the benefits and uses of each.

The main issue was the proper abstraction of the code. I felt that the initial shooting and collocation codes were abstracted enough to deal with ODEs of any shape. Both codes could handle first order and second order ODEs. I haven't tested them with 3rd order ODEs, but they should work.

Putting them together, however, into one giant program proved to be more difficult than anticipated. The difference in the required inputs and the outputs of the functions was hard to reconcile. If I had more time I would try to resolve this problem by making the function outputs more similar. Storing only the first row of the collocation output, as well as the required order, should be enough to remake the entire array.

I was completely at a loss with how to handle shooting vs lamda x:x. I think I am misunderstanding how the solver works on some level, but I feel the two functions are too different. In fact, I even created an **islambda** function to specifically detect when the discretization passed is a lambda function. However, this required the use of an if statement and I felt that this would go against the spirit of properly abstracting the code.

As the continuation code grew bigger, the number of parameters being passed between functions also grew. It was important to have a specific order of parameters in mind when writing the code so as not to get confused. Giving parameters initial values, and the use of *args or **kwargs, would also help make things less cluttered.

Lastly, abstracting the solver was also an issue. The other root finders would sometimes return a finite value even when the solution failed to converge. This was a problem and making a check function that verifies that the function properly zeroed tended to cause infinite loops (tolerance too low/high). This lead to an over reliance on fsolve, full_output=1. A lot more reading on the python documentation available on line is necessary.