Chapter 8.7 - Implementing Python for Boundary Value Problems

11 MIN · YOUTUBE · HTTPS://WWW.YOUTUBE.COM/WATCH?V=9AELCMW12FC
https://www.youtube.com/watch?v=9AelCMw12fc

SUMMARY

The discussion focuses on using Python's built-in tools to solve boundary value problems (BVPs) through the `scipy.integrate.solve_bvp` function. The speaker emphasizes the importance of transforming differential equations into a system of first-order equations and the necessity of good initial guesses for effective solutions, especially in nonlinear scenarios.

- *Boundary value problems require expressing differential equations as first-order systems.*
- *Python's `solve_bvp` function simplifies solving BVPs with built-in algorithms.*
- *Boundary conditions must be expressed as functions equal to zero for compatibility with Python's architecture.*
- *Initial guesses significantly impact the convergence of solutions, particularly for nonlinear problems.*
- *The example includes both linear and nonlinear BVPs, illustrating the process of defining equations and conditions.*
- *Nonlinear problems can exhibit complex behavior, necessitating careful selection of initial guesses to achieve accurate solutions.*
- *The speaker demonstrates how varying parameters, such as beta, affects the solutions of nonlinear eigenvalue problems.*
- *Overall, Python's tools are powerful for solving sophisticated boundary value problems when used correctly.*

We're now going to move on to essentially using some of the Python built-in tools for solving boundary value problems. I'm going to walk you through what is built into Python. some of at least some of the code that's built into Python and we're going to execute some boundary value problems solvers. And the nice thing is much of it's packaged up very nicely for you. and that you can really easily use this even for fairly sophisticated problems.

One of the big questions here is how to use good guesses though for your solutions. It's going to be one of the major themes we use towards the end of this one of the last examples here. And so if you don't have good guesses, these things can be problematic. But if you get good guesses, you can actually do quite a good job with using the built-ins out of Python. So let's start off. We're going to solve a boundary value problem, right? So we're going to have some differential equation with boundary conditions. And so let's start off with something simple linear second order differential equation with the following boundary conditions. Our first step always when we set it up for ourselves typically is to write this as a system of first order equations. So it's going to be sort of your standard step whenever you start off these problems. That's probably what you have to do. Certainly when we did shooting you had to do that. and so we want to do the same thing here which is how do you set it up as a first order set of equations. So again you define y1 is the solution, y2 is the derivative of the solution and here is what you end up getting as a system of first order equations and then you write your

boundary condition in terms of the these new variables y1 and y2. So those there you are. So that's the transformation you have to do right out of the gate. If you get this wrong, you're never going to get this thing to work. This is a fairly simple thing to do and it's you always have to do it. So just whatever your order your system always turn into system of first order equations. That's going to be the first step you're always going have to do and that's up to you.

So the code isn't going to do that. You're going to do that is write that down. Now once we have that, let's talk about setting up for Python. The way Python wants to work at the left and the right boundary conditions, they want the boundaries expressed as some function of this variables. Let's say one one two equal to zero on the right and on the left. So that's the way the code is going to be set up. That's their architecture as the default. This is what we expect. You express your boundary conditions as something equal to zero. So in what we just had here, here's the boundary conditions, right?

Right now it's written as y1 is equal to 3. To write that the way that Python wants it, we'd have to write y1 - 3 is equal to 0. Same thing with the second equation. And so that's what we've done here. So the first boundary condition on the left is y1 - 3 is 0. And the second boundary condition is y1 + 2 y2 - 5 is equal to zero. So this is just the default way you would have to write out the boundary conditions to use the python packages.

So first of all let's go ahead and then start setting up to use from scypi integrate use solve bvp. So solve boundary value problem. So it's a it's a it's an algorithm. It's already built for you to use. So once you have your differential equation written in first order form with your boundary conditions set up as the left one equal to zero, the right one equal to zero, whatever those functions are, you are now ready to use this piece of code. So let's go ahead and walk through it and the different components. The first thing you do is you define the the the boundary the differential equation. In other words, the right hand side of the boundary value problem, which is this first line of code is just your first order system of differential equations.

Once you have that, that sets up the differential equation. Then you have a second definition here, which is the boundary conditions. BV BVP_BC, which is the boundary conditions. And you write the left and the right boundary conditions. Again, both of these are equal to zero. So the first thing is equal to zero. The second function is equal to zero. So we just wrote those out and that's way we impose them here. So you have the boundary, you have the differential equation, you have its boundary conditions. Now what comes next is defining where you're solving it. I'm f solving on the range 1 to three. Okay? And I'm going to bring out 10 points for instance. I could do more if I wanted, but that's simple enough for this for now. And this other piece of code here, which is an initial guess.

Right now, we're just going to get zeros. But this is actually probably one of the most important pieces of the code to actually make it work in practice. For this simple problem, it almost doesn't matter what I guess. I could just guess zeros. And in fact, if you don't put anything in, it will start with zeros. But often times, that's going to be insufficient to get you good solutions. So once you've defined all these things, the differential equation, the boundary conditions, the range I'm solving it on, and the initial guess, you just call solve BVP.

And you tell it, here's my differential equation, here are the boundary conditions, here is the range I'm solving on, and here's my guess. Go. And there's your solution. So out pops the answer. So it will go through and actually what it typically does is sets up a relaxation scheme an iteration scheme to go solve for this which we talked about in the last chapter or the last section of this set of lectures. And so here is the solution to this. Once you get that out it's converged to the solution in fact which is the answer to this thing. So great we have a solution and of course this is a linear problem so it's pretty easy to do and I don't even have to worry about a good guess. It just solves this very easily. But of course, most problems aren't that easy because I can actually write down the solution analytically for that. But here we want to go to a little bit harder problem. So let's go to something that's nonlinear and that actually is a kind of and it's about it's actually an igen value problem. So it's a nonlinear value problem I'm going to solve here which is here. It is this is fairly close to what we did in when we were looking at the wavegu problems which is second derivative 100 minus beta. Beta is our igen value y plus now but here's the complication no longer linear I added gamma y cubed we can always solve this with shooting but this is going to do a relaxation technique for this here the boundary conditions zero on the left at minus one zero on the right at one and so we want to set this problem up and see how would we solve it with this solve BVP routine in python so step one write this equation as a set of first order odes and then express these in those two variables the boundary conditions. So that's what we're doing here. We define again y1 as the solution, y2 as the derivative of the solution, write it in terms of new variables y1 and i2. So here's the system of first order equations. It's nonlinear. Look, there's a y1 cubed there. And here are the boundary conditions expressed there. So I have the boundary conditions. I have the differential equation. So let's go after it. So going back to that code base, first thing I do is define the differential equations. Okay, well I just did that. That's this here is now written here. Second, define the boundary conditions. Well, the boundary conditions are really simple, which is things are are actually going to zero at the left and the right. And so that's that's how we're implementing it here directly. And actually it's interesting what I'm going to do is when I define these boundary conditions I'm actually going to define the solutions are zero but I can have anything for the derivative. So I'm going to just impose that the derivative is not zero. This is important actually. So when I do this often times because one of the solutions to this problem is that this is the zero solution. So that's the thing it usually would find unless I impose here that here this is the y1 is the derivative. I make it so the derivative is not zero. So there's no if if the derivative is zero, the solution is going to be zero across there. So if I'm by making it enforcing that it's not zero, it's going to try to find something that's a non-trivial solution.

So I have to enforce that otherwise it has a little bit of problem. And again, so I'm imposing these two boundary conditions on the left, this one on the right. Okay. And then finally, I just go ahead and what I'm going to do here is I'm going to actually guess the solution form. Now, for this, we already did this problem where we knew what the solution sort of looked like in this wave guide. So, so I'm going to just guess something like a cosine across there initially and then you can even look for the higher order modes if you want. And by doing this, then we can just simply say, okay, go for it. Solve on the domain negative one to one. That's this thing here. So give me your differential equation, the boundary conditions, the domain you're solving on which one with 50, and your guess. And notice what I'm doing here. Beta is a parameter that I let it actually play with. So it's going to have to adjust beta. And so I'm explicitly telling that you can play around with beta. So we're finding this values to this problem. And when you do that, here's the solution. So it adjusts this thing. it finds the value of beta that will allow it to solve the left boundary conditions and the round boundary conditions and converge. So this is a very nice technique. So again this solve BVP is pretty powerful. It can even let you solve this when you

get pretty strong nonlinearities. When I saw that solution that the nonlinearity the y1 cubed in there was actually quite small but that has a huge impact on the solution. So as you make it the y1 cube bigger in other words you make it not weakly nonlinear but strongly nonlinear the solutions change significantly. It puts a lot of pressure on that code to actually find the solutions and this is where the guess really plays a huge role and when you do that you can find solutions like this.

So when you start this is the solution I showed you. But when you make the beta value very big bigger in terms of both being positive and negative these are what the solutions warp into. So these are quite different shapes than this here right so these are actually very difficult to get in practice. These are and but this thing actually pulls it out for you. If you give it a good guess it will actually find these nice solutions for you. These are strongly nonlinear igen functions. And so what we've done here is we've solved a nonlinear igen value problem that's fairly sophisticated solutions using the solve BBP structure. But again, I want you to recognize that if you're going to use this, having good initial guesses is going to be really critical to making that piece of code work well for you. So you have to kind of know a little bit about the solution base. So once you have that, you can use these tools that are already built into Python and are actually really quite nice tools in general for solving value problems.