Lab 2

Differential Equations with Boundary Conditions

More Python

P2.1 Work through Chapter 2 of *Introduction to Python*.

Initial conditions vs. boundary conditions

In Physics 330, we studied the behavior of systems where the initial conditions were specified and we calculated how the system evolved forward in time (e.g. the flight of a baseball given its initial position and velocity). In these cases we were able to use Matlab's convenient built-in differential equation solvers (like `ode45`) to model the system. The situation becomes somewhat more complicated if instead of having initial conditions, a differential equation has boundary conditions specified at both ends of the interval (rather than just at the beginning). This seemingly simple change in the boundary conditions makes it hard to use canned differential equation solvers. Fortunately, there are better ways to solve these systems. In this section we develop a method for using a grid and the finite difference formulas we developed in Lab 1 to solve ordinary differential equations with linear algebra techniques.

Solving differential equations with linear algebra

Consider the differential equation

$$y''(x) + 9y(x) = \sin(x) \quad ; \quad y(0) = 0, \quad y(2) = 1 \tag{2.1}$$

Notice that this differential equation has boundary conditions at both ends of the interval instead of having initial conditions at $x = 0$. If we represent this equation on a grid, we can turn this differential equation into a set of algebraic equations that we can solve using linear algebra techniques. Before we see how this works, let's first specify the notation that we'll use. We assume that we have set up a cell-edge spatial grid with N grid points, and we refer to the x values at the grid points using the notation x_j, with $j = 1..N$. We represent the (as yet unknown) function values $y(x_j)$ on our grid using the notation $y_j = y(x_j)$.

Now we can write the differential equation in finite difference form as it would appear on the grid. The second derivative in Eq. (2.1) is rewritten using the

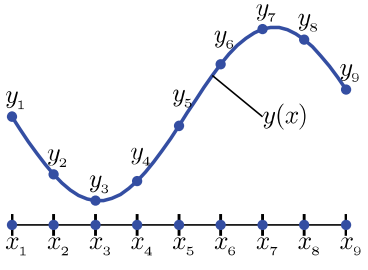

Figure 2.1 A function $y(x)$ repre-

centered difference formula (see Eq. (1.5)), so that the finite difference version of Eq. (2.1) becomes:

$$\frac{y_{j+1}-2y_j+y_{j-1}}{h^2}+9y_j=\sin(x_j) \tag{2.2}$$

Now let's think about Eq. (2.2) for a bit. First notice that it is not *an* equation, but a system of many equations. We have one of these equations at every grid point j, except at $j=1$ and at $j=N$ where this formula reaches beyond the ends of the grid and cannot, therefore, be used. Because this equation involves y_{j-1}, y_j, and y_{j+1} for the interior grid points $j=2\ldots N-1$, Eq. (2.2) is really a system of $N-2$ coupled equations in the N unknowns $y_1\ldots y_N$. If we had just two more equations we could find the y_j's by solving a linear system of equations. But we do have two more equations; they are the boundary conditions:

$$y_1=0 \quad ; \quad y_N=1 \tag{2.3}$$

which completes our system of N equations in N unknowns.

Before Python can solve this system we have to put it in a matrix equation of the form

$$\mathbf{Ay}=\mathbf{b}, \tag{2.4}$$

where **A** is a matrix of coefficients, **y** the column vector of unknown y-values, and **b** the column vector of known values on the right-hand side of Eq. (2.2). For the particular case of the system represented by Eqs. (2.2) and (2.3), the matrix equation is given by

$$\begin{bmatrix} 1 & 0 & 0 & 0 & \ldots & 0 & 0 & 0 \\ \frac{1}{h^2} & -\frac{2}{h^2}+9 & \frac{1}{h^2} & 0 & \ldots & 0 & 0 & 0 \\ 0 & \frac{1}{h^2} & -\frac{2}{h^2}+9 & \frac{1}{h^2} & \ldots & 0 & 0 & 0 \\ . & . & . & . & \ldots & . & . & . \\ . & . & . & . & \ldots & . & . & . \\ . & . & . & . & \ldots & . & . & . \\ 0 & 0 & 0 & 0 & \ldots & \frac{1}{h^2} & -\frac{2}{h^2}+9 & \frac{1}{h^2} \\ 0 & 0 & 0 & 0 & \ldots & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ . \\ . \\ . \\ y_{N-1} \\ y_N \end{bmatrix} = \begin{bmatrix} 0 \\ \sin(x_2) \\ \sin(x_3) \\ . \\ . \\ . \\ \sin(x_{N-1}) \\ 1 \end{bmatrix}. \tag{2.5}$$

Convince yourself that Eq. (2.5) is equivalent to Eqs. (2.2) and (2.3) by mentally doing each row of the matrix multiply by tipping one row of the matrix up on end, dotting it into the column of unknown y-values, and setting it equal to the corresponding element in the column vector on the right.

Once we have the finite-difference approximation to the differential equation in this matrix form (**Ay** = **b**), a simple linear solve is all that is required to find the solution array y_j. NumPy can do this solve with this command:

```
numpy.linalg.solve(a, b)
```

P2.2 (a) Set up a cell-edge grid with $N=30$ grid points, like this:

```
from numpy import linspace

N=30   # the number of grid points
```

```
a=0
b=2
x,h = linspace(a,b,N,retstep = True)
```

Look over this code and make sure you understand what it does. You may be wondering about the command `x=x'`. This turns the row vector `x` into a column vector `x`. This is not strictly necessary, but it is convenient because the `y` vector that we will get when we solve will be a column vector and Python needs the two vectors to be the same dimensions for plotting.

(b) Solve Eq. (2.1) symbolically using Mathematica's `DSolve` command. Then type the solution formula into the Python script that defines the grid above and plot the exact solution as a blue curve on a cell-edge grid with N points.

(c) Now load the matrix in Eq. (2.5) and do the linear solve to obtain y_j and plot it on top of the exact solution with red dots (`'r.'`) to see how closely the two agree. Experiment with larger values of N and plot the difference between the exact and approximate solutions to see how the error changes with N. We think you'll be impressed at how well the numerical method works, if you use enough grid points.

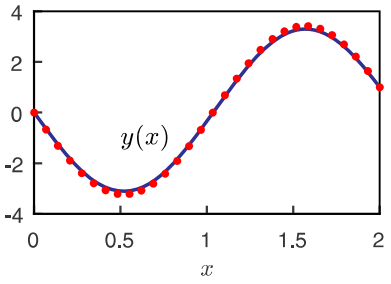

Figure 2.2 The solution to 2.2(c) with $N = 30$

Let's pause and take a minute to review how to apply the technique to solve a problem. First, write out the differential equation as a set of finite difference equations on a grid, similar to what we did in Eq. (2.2). Then translate this set of finite difference equations (plus the boundary conditions) into a matrix form analogous to Eq. (2.5). Finally, build the matrix **A** and the column vector **y** in Python and solve for the vector **y**. Our example, Eq. (2.1), had only a second derivative, but first derivatives can be handled using the centered first derivative approximation, Eq. (1.5).

Now let's practice this procedure for a couple more differential equations:

P2.3 (a) Write out the finite difference equations on paper for the differential equation

$$y'' + \frac{1}{x}y' + (1 - \frac{1}{x^2})y = x \quad ; \quad y(0) = 0, \quad y(5) = 1 \tag{2.6}$$

Then write down the matrix **A** and the vector **b** for this equation. Finally, build these matrices in a Python script and solve the equation using the matrix method. Compare the solution found using the matrix method with the exact solution

$$y(x) = \frac{-4}{J_1(5)} J_1(x) + x$$

($J_1(x)$ is the first order Bessel function.)

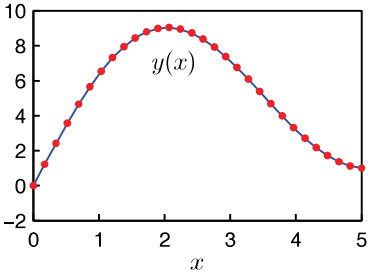

Figure 2.3 Solution to 2.3(a) with $N = 30$ (dots) compared to the exact solution (line)

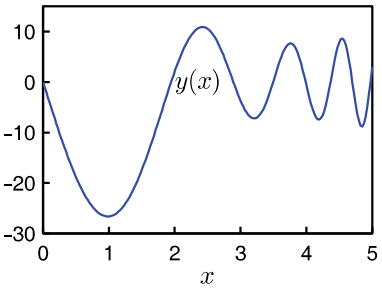

Figure 2.4 Solution to 2.3(b) with $N = 200$

(b) Solve the differential equation

$$y'' + \sin(x)y' + e^x y = x^2 \quad ; \quad y(0) = 0, \quad y(5) = 3 \tag{2.7}$$

in Python using the matrix method. Also solve this equation numerically using Mathematica's `NDSolve` command and plot the numeric solution. Compare the Mathematica plot with the Python plot. Do they agree? Check both solutions at $x = 4.5$; is the agreement reasonable? How many points do you have to use in your numerical method to get agreement with Mathematica to 3 decimal places?

Derivative boundary conditions

Now let's see how to modify the linear algebra approach to differential equations so that we can handle boundary conditions where derivatives are specified instead of values. Consider the differential equation

$$y''(x) + 9y(x) = x \quad ; \quad y(0) = 0 \quad ; \quad y'(2) = 0 \tag{2.8}$$

We can satisfy the boundary condition $y(0) = 0$ as before (just use $y_1 = 0$), but what do we do with the derivative condition at the other boundary?

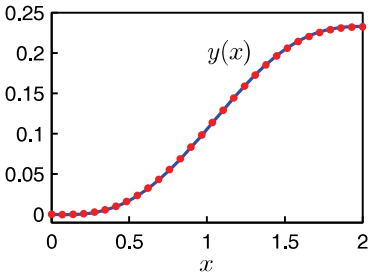

Figure 2.5 The solution to 2.4(a) with $N = 30$. The RMS difference from the exact solution is 8.8×10^{-4}

P2.4 (a) A crude way to implement the derivative boundary condition is to use a forward difference formula

$$\frac{y_N - y_{N-1}}{h} = y'\Big|_{x=2} \; . \tag{2.9}$$

In the present case, where $y'(2) = 0$, this simply means that we set $y_N = y_{N-1}$. Solve Eq. (2.8) in Python using the matrix method with this boundary condition. (Think about what the new boundary conditions will do to the final row of matrix **A** and the final element of vector **b**). Compare the resulting numerical solution to the exact solution obtained from Mathematica:

$$y(x) = \frac{x}{9} - \frac{\sin(3x)}{27\cos(6)} \tag{2.10}$$

(b) Let's improve the boundary condition formula using quadratic extrapolation. Use Mathematica to fit a parabola of the form

$$y(x) = a + bx + cx^2 \tag{2.11}$$

to the last three points on your grid. To do this, use (2.11) to write down three equations for the last three points on your grid and then solve these three equations for a, b, and c. Write the x-values in terms of the

last grid point and the grid spacing ($x_{N-2} = x_N - 2h$ and $x_{N-1} = x_N - h$) but keep separate variables for y_{N-2}, y_{N-1}, and y_N. (You can probably use your code from Problem **??** with a little modification.)

Now take the derivative of Eq. (2.11), evaluate it at $x = x_N$, and plug in your expressions for b and c. This gives you an approximation for the $y'(x)$ at the end of the grid. You should find that the new condition is

$$\frac{1}{2h}y_{N-2} - \frac{2}{h}y_{N-1} + \frac{3}{2h}y_N = y'(x_N) \tag{2.12}$$

Modify your script from part (a) to include this new condition and show that it gives a more accurate solution than the the crude technique of part (a). When you check the accuracy, don't just look at the end of the interval. All of the points are coupled by the matrix **A**, so you should use a full-interval accuracy check like the RMS (root-mean-square) error:

```
sqrt(mean((y-yexact).^2))
```

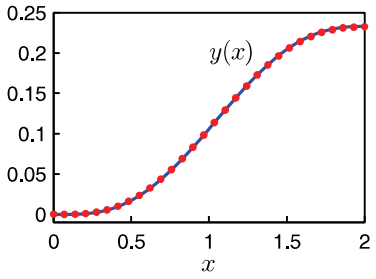

Figure 2.6 The solution to 2.4(b) with $N = 30$. The RMS difference from the exact solution is 5.4×10^{-4}

Nonlinear differential equations

Finally, we must confess that we have been giving you easy problems to solve, which probably leaves the impression that you can use this linear algebra trick to solve all second-order differential equations with boundary conditions at the ends. The problems we have given you so far are easy because they are *linear* differential equations, so they can be translated into *linear* algebra problems. Linear problems are not the whole story in physics, of course, but most of the problems we will do in this course are linear, so these finite-difference and matrix methods will serve us well in the labs to come.

P2.5 (a) Here is a simple example of a differential equation that isn't linear:

$$y''(x) + \sin\big[y(x)\big] = 1 \quad ; \quad y(0) = 0, \;\; y(3) = 0 \tag{2.13}$$

Work at turning this problem into a linear algebra problem to see why it can't be done, and explain the reasons to the TA.

(b) Find a way to use a combination of linear algebra and iteration (initial guess, refinement, etc.) to solve Eq. (2.13) in Python on a grid. Check your answer by using Mathematica's built-in solver to plot the solution.

HINT: Write the equation as

$$y''(x) = 1 - \sin\big[y(x)\big] \tag{2.14}$$

Make a guess for $y(x)$. (It doesn't have to be a very good guess. In this case, the guess $y(x) = 0$ works just fine.) Then treat the whole right

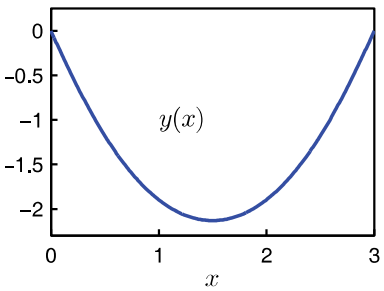

Figure 2.7 The solution to 2.5(b).

side of Eq. (2.14) as known so it goes in the **b** vector. Then you can solve the equation to find an improved guess for $y(x)$. Use this better guess to rebuild **b** (again treating the right side of Eq. (2.14) as known), and then re-solve to get and even better guess. Keep iterating until your $y(x)$ converges to the desired level of accuracy. This happens when your $y(x)$ satisfies (2.13) to a specified criterion, *not* when the change in $y(x)$ from one iteration to the next falls below a certain level. An appropriate error vector would be `Ay-(1-sin(y)` evaluated at interior points only, `n=2:N-1`. This is necessary because the end points don't satisfy the differential equation.