Lab 1

Grids and Numerical Derivatives

Introduction to Python

In this course we will use Python to study numerical techniques for solving some partial differential equations that arise in Physics. Don't be scared of this new language. Most of the ideas, and some of the syntax, that you learned for Matlab will transfer directly to Python. We'll work through some brief tutorials about Python at the beginning of each lab, focusing on the particular ideas that you'll need to complete that lab. Pretty soon you will be Python wizards.

P1.1 Work through Chapter 1 of *Introduction to Python*. There you will learn the basics of how to write a Python program, how to declare and use entities called NumPy arrays, and also learn some basic plotting techniques.

With that Python knowledge under our belts, let's move on to begin our study of partial differential equations.

Spatial grids

When we solved ordinary differential equations in Physics 330 we were usually moving something forward in time, so you may have the impression that differential equations always "flow." This is not true. If we solve a spatial differential equation, like the one that gives the shape of a chain draped between two posts, the solution just sits in space; nothing flows. Instead, we choose a small spatial step size (think of each individual link in the chain) and seek to find the correct shape by somehow finding the height of the chain at each link.

In this course we will solve partial differential equations, which usually means that the desired solution is a function of both space x, which just sits, and time t, which flows. And when we solve problems like this we will be using *spatial grids*, to represent the x-part that doesn't flow. The NumPy arrays that you just learned about above are perfect for representing these kinds of spatial grids.

We'll encounter three basic types of spatial grids in this class. Figure 1.1 shows a graphical representation of these three types of spatial grids for the region $0 \le x \le L$. We divide this region into spatial *cells* (the spaces between vertical lines) and functions are evaluated at N discrete *grid points* (the dots). In a *cell-edge* grid, the grid points are located at the edge of the cell. In a *cell-center* grid, the points are located in the middle of the cell. Another useful grid is a cell-center grid with *ghost points*. The ghost points (unfilled dots) are extra grid points on either side of the interval of interest and are useful when we need to consider the derivatives at the edge of a grid.

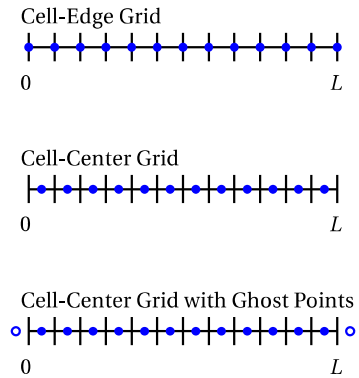

Figure 1.1 Three common spatial grids

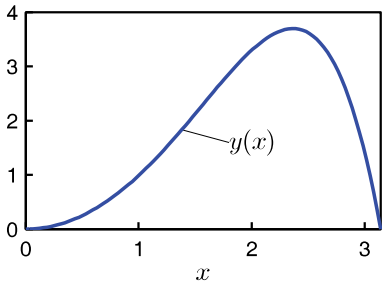

Figure 1.2 Plot from 1.2(a)

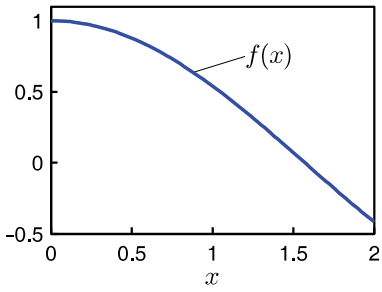

Figure 1.3 Plot from 1.2(b)

P1.2 (a) Write a Python program that creates a cell-edge spatial grid in the variable `x` as follows:

```
from numpy import linspace, pi

N=100   # the number of grid points
a=0
b=pi
x,h = linspace(a,b,N,retstep = True)
```

Plot the function $y(x) = \sin(x)\sinh(x)$ on this grid. Explain the relationship between the number of cells and the number of grid points in a cell-edge grid.

(b) Explain the relationship between the number of cells and the number of grid points in a cell-center grid. Then write some code using NumPy's `arange` function to create a cell-centered grid that has exactly 100 cells over the interval $0 \le x \le 2$.

Evaluate the function $f(x) = \cos x$ on this grid and plot this function. Then estimate the area under the curve by summing the products of the centered function values f_j with the widths of the cells h like this (midpoint integration rule):

```
sum(f)*h
```

Compare this result to the exact answer obtained by integration:

$$A = \int_0^2 \cos x \, dx = \sin(x)\Big|_0^2 = \sin(2)$$

(c) Build a cell-center grid with ghost points over the interval $0 \le x \le \pi/2$ with 500 cells (502 grid points), and evaluate the function $f(x) = \sin x$ on this grid. Now look carefully at the function values at the first two grid points and at the last two grid points. The function $\sin x$ has the property that $f(0) = 0$ and $f'(\pi/2) = 0$. The cell-center grid doesn't have points at the ends of the interval, so these boundary conditions on the function need to be enforced using more than one point. Explain how the ghost points can be used in connection with interior points to specify both function-value boundary conditions and derivative-value boundary conditions.

Interpolation and extrapolation

Grids only represent functions at discrete points, and there will be times when we want to find good values of a function *between* grid points (interpolation) or *beyond* the last grid point (extrapolation). We will use interpolation and extrapolation techniques fairly often during this course, so let's review these ideas.

The simplest way to estimate function values is to use the fact that two points define a straight line. For example, suppose that we have function values (x_1, y_1)

and (x_2, y_2). The formula for a straight line that passes through these two points is

$$y - y_1 = \frac{(y_2 - y_1)}{(x_2 - x_1)}(x - x_1) \tag{1.1}$$

Once this line has been established it provides an approximation to the true function $y(x)$ that is pretty good in the neighborhood of the two data points. To linearly interpolate or extrapolate we simply evaluate Eq. (1.1) at x values between or beyond x_1 and x_2.

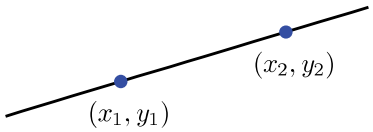

Figure 1.4 The line defined by two points can be used to interpolate between the points and extrapolate beyond the points.

P1.3 Use Eq. (1.1) to do the following special cases:

(a) Find an approximate value for $y(x)$ halfway between the two points x_1 and x_2. Does your answer make sense?

(b) Find an approximate value for $y(x)$ 3/4 of the way from x_1 to x_2. Do you see a pattern?

(c) If the spacing between grid points is h (i.e. $x_2 - x_1 = h$), show that the linear extrapolation formula for $y(x_2 + h)$ is

$$y(x_2 + h) = 2y_2 - y_1 \tag{1.2}$$

This provides a convenient way to estimate the function value one grid step beyond the last grid point. Also show that

$$y(x_2 + h/2) = 3y_2/2 - y_1/2. \tag{1.3}$$

We will use both of these formulas during the course.

Derivatives on grids

When solving partial differential equations, we will frequently need to calculate derivatives on our grids. In your introductory calculus book, the derivative was probably introduced using the *forward difference* formula

$$f'(x) \approx \frac{f(x + h) - f(x)}{h}. \tag{1.4}$$

The word "forward" refers to the way this formula reaches forward from x to $x + h$ to calculate the slope. The exact derivative represented by Eq. (1.4) in the limit that h approaches zero. However, we can't make h arbitrarily small when we represent a function on a grid because (i) the number of cells needed to represent a region of space becomes infinite as h goes to zero; and (ii) computers represent numbers with a finite number of significant digits so the subtraction in the numerator of Eq. (1.4) loses accuracy when the two function values are very close. But given these limitation we want to be as accurate as possible, so we want to use the best derivative formulas available. The forward difference formula isn't one of them.

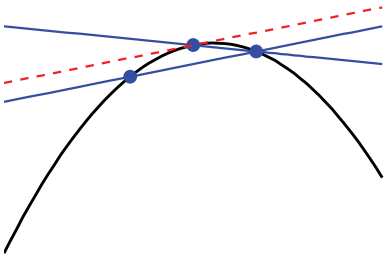

Figure 1.5 The forward and centered difference formulas both approximate the derivative as the slope of a line connecting two points. The centered difference formula gives a more accurate approximation because it uses points before and after the point where the derivative is being estimated. (The true derivative is the slope of the dotted tangent line).

The best first derivative formula that uses only two function values is usually the *centered difference* formula:

$$f'(x) \approx \frac{f(x+h)-f(x-h)}{2h}. \tag{1.5}$$

It is called "centered" because the point x at which we want the slope is centered between the places where the function is evaluated. Take a minute to study Fig. 1.5 to understand visually why the centered difference formula is so much better than the forward difference formula. The corresponding centered second derivative formula is

$$f''(x) \approx \frac{f(x+h)-2f(x)+f(x-h)}{h^2} \tag{1.6}$$

We will derive both of these formulas later, but for now we just want you to understand how to use them.

The colon operator provides a compact way to evaluate Eqs. (1.5) and (1.6) on a grid. If the function we want to take the derivative of is stored in an array `f`, we can calculate the centered first derivative like this (remember that Python array indexes are zero-based):

```
fp = numpy.zeros_like(f)
fp[1:N-2]=(f[2:N-1]-f[0:N-3])/(2*h)
```

and the centered second derivative at each interior grid point like this:

```
fpp = numpy.zeros_like(f)
fpp[1:N-2]=(f[2:N-1]-2*f[1:N-2]+f[0:N-3])/h^2
```

The variable `h` is the spacing between grid points and `N` is the number of grid points. Study this code until you are convinced that it represents Eqs. (1.5) and (1.6) correctly.

The derivative at the first and last points on the grid can't be calculated using Eqs. (1.5) and (1.6) since there are not grid points on both sides of the endpoints. About the best we can do is to extrapolate the interior values of the two derivatives to the end points. If we use linear extrapolation then we just need two nearby points, and the formulas for the derivatives at the end points are found using Eq. (1.2):

```
fp[0]=2*fp[1]-fp[2]
fp[N-1]=2*fp[N-2]-fp[N-3]
fpp[0]=2*fpp[1]-fpp[2]
fpp[N-1]=2*fpp[N-2]-fpp[N-3]
```

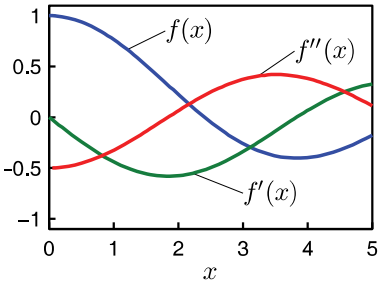

Figure 1.6 Plots from 1.4

P1.4 Create a cell-edge grid with $N = 20$ on the interval $0 \le x \le 5$. Load $f(x)$ with the Bessel function $J_0(x)$ and numerically differentiate it to obtain $f'(x)$ and $f''(x)$. Then make overlaid plots of the numerical derivatives with the exact derivatives:

$$f'(x) = -J_1(x)$$

$$f''(x) = \frac{1}{2}\left(-J_0(x) + J_2(x)\right)$$

Errors in the approximate derivative formulas

We'll conclude this lab with a look at where the approximate derivative formulas come from and at the types of the errors that pop up when using them. The starting point is Taylor's expansion of the function f a small distance h away from the point x

$$f(x+h) = f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + \cdots + f^{(n)}(x)\frac{h^n}{n!} + \cdots \tag{1.7}$$

Let's use this series to understand the forward difference approximation to $f'(x)$. If we apply the Taylor expansion to the $f(x+h)$ term in Eq. (1.4), we get

$$\frac{f(x+h)-f(x)}{h} = \frac{\left[f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + \cdots\right] - f(x)}{h} \tag{1.8}$$

The higher order terms in the expansion (represented by the dots) are smaller than the f'' term because they are all multiplied by higher powers of h (which we assume to be small). If we neglect these higher order terms, we can solve Eq. (1.8) for the exact derivative $f'(x)$ to find

$$f'(x) \approx \frac{f(x+h)-f(x)}{h} - \frac{h}{2}f''(x) \tag{1.9}$$

From Eq. (1.9) we see that the forward difference does indeed give the first derivative back, but it carries an error term which is proportional to h. But if h is small enough then the contribution from the term containing $f''(x)$ will be too small to matter and we will have a good approximation to $f'(x)$.

For the centered difference formula, we use Taylor expansions for both $f(x+h)$ and $f(x-h)$ in Eq. (1.5) to write

$$\begin{aligned}\frac{f(x+h)-f(x-h)}{2h} &= \frac{\left[f(x) + f'(x)h + f''(x)\frac{h^2}{2} + f'''(x)\frac{h^3}{6} + \cdots\right]}{2h} \\ &\quad - \frac{\left[f(x) - f'(x)h + f''(x)\frac{h^2}{2} - f'''(x)\frac{h^3}{6} + \cdots\right]}{2h}\end{aligned} \tag{1.10}$$

If we again neglect the higher-order terms, we can solve Eq. (1.10) for the exact derivative $f'(x)$. This time, we find that the f'' terms exactly cancel to give

$$f'(x) \approx \frac{f(x+h)-f(x-h)}{2h} - \frac{h^2}{6}f'''(x) \tag{1.11}$$

Notice that for the centered formula the error term is much smaller, only of order h^2. So if we decrease h in both the forward and centered difference formulas by a factor of 10, the forward difference error will decrease by a factor of 10, but the centered difference error will decrease by a factor of 100. This is the reason we try to use centered formulas whenever possible in this course.

P1.5 Write a Python program to compute the forward and centered difference formulas for the first derivative of the function $f(x) = e^x$ at $x = 0$ with $h = 0.1,\ 0.01,\ 0.001$. Also calculate the centered second derivative formula for these values of h. Verify that the error estimates in Eqs. (1.9) and (1.11) agree with the numerical testing.

Note that at $x = 0$ the exact values of both f' and f'' are equal to $e^0 = 1$, so just subtract 1 from your numerical result to find the error.

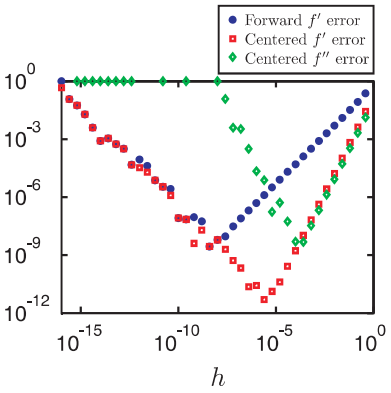

Figure 1.7 Error in the forward and centered difference approximations to the first derivative and the centered difference formula for the second derivative as a function of h. The function is e^x and the approximations are evaluated for $x = 0$.

In problem 1.5, you should have found that $h = 0.001$ in the centered-difference formula gives a better approximation than $h = 0.01$. These errors are due to the finite grid spacing h, which might entice you to try to keep making h smaller and smaller to achieve any accuracy you want. This doesn't work. Figure 1.7 shows a plot of the error you calculated in problem 1.5 as h continues to decrease (note the log scales). For the larger values of h, the errors track well with the predictions made by the Taylor's series analysis. However, when h becomes too small, the error starts to increase. Finally (at about $h = 10^{-16}$, and sooner for the second derivative) the finite difference formulas have no accuracy at all—the error is the same order as the derivative.

The reason for this behavior is that numbers in computers are represented with a finite number of significant digits. Most computational languages (including Python) use double precision variables, which have 15-digit accuracy. This is normally plenty of precision, but look what happens in a subtraction problem where the two numbers are nearly the same:

$$\begin{array}{rr} & 7.38905699669556 \\ - & 7.38905699191745 \\ \hline & 0.00000000477811 \end{array} \tag{1.12}$$

Notice that our nice 15-digit accuracy has disappeared, leaving behind only 6 significant figures. This problem occurs in calculations with real numbers on all digital computers, and is called *roundoff*. You can see this effect by experimenting with the Python console:

```
h=1e-17
g=1+h
print(g-1)
```

for different values of h and noting that you don't always get h back. Also notice in Fig. 1.7 that this problem is worse for the second derivative formula than it is for the first derivative formula. The lesson here is that it is impossible to achieve arbitrarily high accuracy by using arbitrarily tiny values of h. In a problem with a size of about L it doesn't do any good to use values of h any smaller than about $0.0001L$.