```
import numpy as np
import matplotlib.pyplot as plt
```

Week 7 Lecture 1: Finite Differences

This week we will begin discussing the concept of numerical differentiation. That is, we will be trying to numerically approximate the derivative(s) of a given function. we will look at this concept from two slightly different angles. One possibility is that we are given some function $f(x)$ and a point x_0 and we want to compute the derivative of f at x_0. That is, we want to find $f'(x_0)$. If f is relatively simple, then we already know how to do this from calculus, so we will generally assume that f is either very complicated or that we do not have access to the formula. (For instance, we might be given a MATLAB function but not be able to read the code.) The other possibility is that we are given a set of data points $(x_0, y_0), (x_1, y_1), \ldots, (x_N, y_N)$ and we want to find out how fast the data points are changing at one of the points (x_k, y_k).

These two situations are not as different as they might sound, and it is easy to convert back and forth between them. If you are given a function $f(x)$, then you can simply plug in the numbers $x_0, x_1, \ldots, x_N$ to obtain all the necessary y-values and then forget about the function. Likewise, if you have a set of data, you can simply pretend that all of the y-values arose from some function $f(x)$, even if you don't know the formula.

Both approaches are useful, but for different reasons. From a theoretical point of view, it really only makes sense to talk about the derivative of a function. There is no good rigorous definition of a derivative on a discrete set of points. We will therefore always assume that we have a function $f(x)$ when working by hand. However, in real world applications we almost never have access to the true formula that produced our data. Instead, we have a set of data points and it is either expensive or impossible to collect any more. When writing code, we will therefore almost always assume that we have a fixed set of x and y values, even if we really do know the function involved.

Forward difference scheme

Remember from your calculus class that the definition of a derivative is

$$f'(x) = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}. \qquad (1)$$

This presents a problem for us because computing a limit requires infinitely many points. To find $f'(x)$ exactly, it is not enough to know $f(x + 0.1)$ or $f(x + 0.01)$ or $f(x + 0.001)$; we need to know y values infinitely close to $f(x)$. Of course, finding infinitely many points or points that are infinitely close together is not possible on a computer, so we will have to settle for some sort of approximation.

Equation (1) suggests a simple method of approximating the derivative. If $f'(x)$ is the limit of that difference quotient as Δx becomes arbitrarilly small, perhaps we can choose a small but fixed value for Δx and get a good approximation. That is, if we fix some small number Δx then we should have

$$f'(x) \approx \frac{f(x+\Delta x)-f(x)}{\Delta x}. \qquad (2)$$

It should be obvious from the definition of a limit that this is a valid approach. If we choose too large a Δx then we may get a bad approximation, but as we shrink Δx we will necessarily get closer and closer to the true value of $f'(x)$. Approximations of this form are called *difference schemes*, and this particular one is a *forward difference scheme*. The word "forward" refers to the fact that we are evaluating f at the point x that we care about and another point $x + \Delta x$ ahead of that point. If you were standing on the number line at x, then you would have to look forward to see $x + \Delta x$.

A very important question to ask about any difference scheme is: "How quickly do we converge to the true value $f'(x)$ as we shrink Δx?" That is, if we choose some small Δx then there will be some error between our approximation and the real $f'(x)$. If we choose a Δx that is ten times smaller, what will happen to our error? Will it also be ten times smaller? One hundred times smaller? Something else? We can analyze the error of all difference schemes in a systematic way using Taylor expansions. Remember that

$$f(x + \Delta x) = f(x) + \Delta x f'(x) + \frac{(\Delta x)^2}{2} f''(x) + \frac{(\Delta x)^3}{3!} f'''(x) + \ldots + \frac{(\Delta x)^n}{n!} f^{(n)}(x) + \ldots$$

For this equation to be an equality, we need infinitely many terms. Of course, the whole point of using a difference scheme was so that we could avoid doing something infinitely many times. Fortunately, when Δx is small, $(\Delta x)^2$ or $(\Delta x)^3$ or $(\Delta x)^n$ will be *much* smaller. This means that, as long as Δx is small enough, we can safely ignore the terms with higher powers because they are very close to zero. We will use the notation $\mathcal{O}(\Delta x^n)$ to indicate which terms we are ignoring. We have already encountered this notation before, when counting the number of instructions required to solve a linear system. The idea here is the same: $\mathcal{O}(\Delta x^n)$ represents a polynomial in Δx where the largest term has a power of n. Notice that in the section on linear systems, the largest term was the one with the largest exponent because we were dealing with a variable N that was very large. In this case, the largest term is the one with

the smallest exponent because Δx is very small. This means that $\mathcal{O}(\Delta x^n)$ represents a polynomial where the smallest power of Δx is n. We can therefore rewrite our Taylor expansion as

$$f(x+\Delta x) = f(x) + \Delta x f'(x) + \frac{(\Delta x)^2}{2} f''(x) + \frac{(\Delta x)^3}{6} f'''(x) + \mathcal{O}(\Delta x^4).$$

If we substitute this into the formula from (2), we get

$$f'(x) \approx \frac{1}{\Delta x}[f(x+\Delta x) - f(x)]$$

$$\approx \frac{1}{\Delta x}\left[f(x) + \Delta x f'(x) + \frac{(\Delta x)^2}{2} f''(x) + \frac{(\Delta x)^3}{6} f'''(x) + \mathcal{O}(\Delta x^4) - f(x)\right]$$

$$= \frac{1}{\Delta x}\left[\Delta x f'(x) + \frac{(\Delta x)^2}{2} f''(x) + \frac{(\Delta x)^3}{6} f'''(x) + \mathcal{O}(\Delta x^4)\right]$$

$$= f'(x) + \frac{\Delta x}{2} f''(x) + \frac{(\Delta x)^2}{6} f'''(x) + \mathcal{O}(\Delta x^3).$$

Notice that when Δx goes all the way to zero, everything but the first term goes away and we are just left with $f'(x)$. This is what we were trying to approximate in the first place, which is a good sign. We say that the difference scheme is *consistent*, because in the limit as Δx approaches zero it gives the correct value.

Also note that when Δx is very small, raising it to a power makes it even smaller. In particular, Δx^2 and Δx^3 are much, much closer to zero than Δx. This means that once Δx is sufficiently small, we don't really need to worry about any of the higher powers of Δx. We can therefore write (for sufficiently small Δx):

$$\frac{f(x+\Delta x) - f(x)}{\Delta x} \approx f'(x) + \frac{\Delta x}{2} f''(x).$$

We call the second term in this equation the *leading error term*. It tells us approximately how far off our approximation is from the true value of $f'(x)$. Of course, we don't actually know $f''(x)$, so we don't know exactly what the error is. However, we do know that the error is proportional to Δx raised to the first power. This means that if we shrink Δx by a factor of ten, then the error will also shrink by a factor of ten. Likewise, if we shrink Δx by a factor of one hundred, then the error will also shrink by a factor of 100. Because of this, we say that our difference scheme is first order. We call equation (2) a *first order forward difference scheme* for $f'(x)$. It is important to remember that the phrase "first order" in this name refers to the power of Δx in the error, not to the fact that we are

Backward difference scheme

In general, we like to keep Δx positive in these equations, but the definition of a limit makes no such assumptions. It therefore makes sense to consider negative Δx's as well. Instead of allowing Δx to change sign, we will make a small modification to our difference scheme and just replace every Δx in equation (2) with $-\Delta x$. We obtain

$$f'(x) \approx \frac{f(x)-f(x-\Delta x)}{\Delta x}. \qquad (3)$$

This is called a *backward difference scheme* because it involves the point x and a point behind x. (Again, if you imagine standing on the number line at x then you would have to look backwards to see the point $x - \Delta x$.) We can analyze the error properties of this scheme just like we did with the forward difference scheme. To do so, we need to adjust our Taylor expansion by replacing every Δx with a $-\Delta x$. We get

$$f(x - \Delta x) = f(x) - \Delta x f'(x) + \frac{(\Delta x)^2}{2} f''(x) - \frac{(\Delta x)^3}{6} f'''(x) + \mathcal{O}(\Delta x^4).$$

(Notice that the notation $\mathcal{O}(\Delta x^4)$ is already ignoring any constants in front of the Δx^4 term, so it does not matter if we say $+\mathcal{O}(\Delta x^4)$ or $-\mathcal{O}(\Delta x^4)$. By convention, we always use a + sign.)

If we substitute this into formula (3), we obtain

$$f'(x) \approx \frac{1}{\Delta x}\left[f(x) - \left(f(x) - \Delta x f'(x) + \frac{(\Delta x)^2}{2} f''(x) - \frac{(\Delta x)^3}{6} f'''(x) + \mathcal{O}(\Delta x^4)\right)\right]$$

$$= \frac{1}{\Delta x}\left[\Delta x f'(x) - \frac{(\Delta x)^2}{2} f''(x) + \frac{(\Delta x)^3}{6} f'''(x) + \mathcal{O}(\Delta x^4)\right]$$

$$= f'(x) - \frac{\Delta x}{2} f''(x) + \frac{(\Delta x)^2}{6} + \mathcal{O}(\Delta x^4).$$

Once again, if Δx goes all the way to zero then every term disappears except for the $f'(x)$, so this difference scheme is also consistent. If Δx is very small, but not actually zero, then all the terms after the first two are so small that they don't really matter, so we can say that

$$\frac{f(x)-f(x-\Delta x)}{\Delta x} \approx f'(x) - \frac{\Delta x}{2} f''(x).$$

The leading error term is therefore $-\frac{\Delta x}{2} f''(x)$, which has a Δx^1. We therefore know that our difference scheme is first order, and so we call it a *first order backward difference scheme*.

Central difference scheme

The leading error terms for these forward and backward difference schemes have an interesting relationship. They are the same except for their sign. This means that if $f''(x)$ is positive then the forward difference scheme will overestimate $f'(x)$ by some amount and the backward difference scheme will underestimate by almost exactly the same amount. (The "almost" is important - we threw away higher order terms in those approximations because they were much smaller than the leading term, but those terms aren't exactly zero and they probably don't cancel.) This suggests a better difference scheme: What if we calculate the forward and backward difference approximations and then average them together? This gives us the following scheme:

$$f'(x) \approx \frac{1}{2}\left(\frac{f(x+\Delta x)-f(x)}{\Delta x} + \frac{f(x)-f(x-\Delta x)}{\Delta x}\right) = \frac{f(x+\Delta x)-f(x-\Delta x)}{2\Delta x}. \qquad (4)$$

This is called a *central difference scheme* because x is in between the other points that we use. From the above argument, we have reason to hope that this scheme has better error properties than either of the original schemes alone. We can check this using Taylor expansions, just as we did before. We get

$$f'(x) \approx \frac{f(x+\Delta x)-f(x-\Delta x)}{2\Delta x}$$

$$= \frac{1}{2\Delta x}\left[f(x) + \Delta x f'(x) + \frac{(\Delta x)^2}{2}f''(x) + \frac{(\Delta x)^3}{6}f'''(x) + \mathcal{O}(\Delta x^4) - \left(f(x) - \Delta x f'(x) + \frac{(\Delta x)^2}{2}f''(x) - \frac{(\Delta x)^3}{6}f'''(x) + \mathcal{O}(\Delta x^4)\right.\right.$$

$$= \frac{1}{2\Delta x}\left[2\Delta x f'(x) + \frac{(\Delta x)^3}{3}f'''(x) + \mathcal{O}(\Delta x^4)\right]$$

$$= f'(x) + \frac{(\Delta x)^2}{6}f'''(x) + \mathcal{O}(\Delta x^3).$$

Notice that when Δx goes all the way to zero, we are just left with $f'(x)$. This means that our new scheme is still consistent. The leading error term, however, is quite different in this scheme. In particular, it has a Δx^2 in it, so the error is second order or $\mathcal{O}(\Delta x^2)$. This means that if we shrink Δx by a factor of 10, the error of the forward and backward difference schemes from above only shrinks by a factor of 10, but the error for the central difference scheme shrinks by a factor of $10^2 = 100$. We call this scheme a *second order central difference scheme*.

Why use higher order schemes?

We are usually not just interested in finding $f'(x)$ at one point. Instead, we have a collection of many data points and we want to find f' at all of them so that we will have a good approximation to the function $f'(x)$. In order to get a good approximation to f', we need

the error at each of these points to be small, which means we need to know y values at points very close to all of our original data points. In practice, this means that if you want to make Δx smaller, then you have to collect data at many more points. In particular, if you want to make Δx ten times smaller, then you will typically need to collect ten times as much data. Data collection is often the most expensive part of an experiment, so it is important to use methods that don't require any more data than necessary. This is why we place such an emphasis on the order of the error term. It is not uncommon to encounter a situation like this; You plan to do an experiment that will require (at some point in the analysis) the approximation of a derivative. You collect some data for the pilot study and find that the error in this approximation is so large that you can't use the results. For the full study to be useful, you will need 100 times less error in your approximation. If you use a first order difference scheme, then you will need 100 times more data for the full study, which probably means that the full study will be around 100 times more expensive than the pilot. If you instead used a second order difference scheme, you would only need 10 times more data, which would bring down the cost by a factor of 10.

It is entirely possible to craft higher order difference schemes as well. For some applications this is entirely justified, but in practice it often turns out that second order schemes have a good balance between accuracy and efficiency. Each of our difference schemes in this lecture have required evaluating f twice for each derivative. It turns out that higher order schemes require evaluating f at more points. If you are using a set of pre-computed data points, then this is not really a problem, but if f is some more complicated and slow function, then it is a good idea to avoid evaluating it any more than necessary. This means that more accurate schemes can often become substantially slower. For this reason, it is common to only use second order accurate schemes unless you have a compelling reason not to.

Confirming Order

In the last lecture, we discussed several difference schemes for approximating the derivative $f'(x)$. The forward and backward difference schemes that we discussed were first order accurate (i.e., the error was $\mathcal{O}(\Delta x)$), while the central difference scheme was second order accurate (i.e., the error was $\mathcal{O}(\Delta x^2)$).

Let's try to confirm this analysis with an example. In particular, let's use $f(x) = \sin(x)$ and try to calculate $f'(x_0)$ at $x_0 = 1$. This is a simple enough problem that we know how to do it by hand. We have $f'(x) = \cos(x)$, and so the true value of $f'(x_0)$ is $\cos(1)$.

```
true_solution = np.cos(1)
print(true_solution)
```

```
0.5403023058681398
```

Let's try to approximate this derivative with several different schemes. First, let's use a forward scheme.

Forward Scheme

We are trying to approximate the derivative of $f(x) = \sin(x)$ at $x_0 = 1$. We know that the actual derivative is $f'(x) = \cos(x)$, so we can determine exactly how far off our approximations are.

We want to use the forward difference scheme

$f'(x_0) \approx \frac{f(x_0+\Delta x)-f(x_0)}{\Delta x}$

for various choices of Δx. For example,

```
x0 = 1
dx = 1
forward_approx = (np.sin(x0 + dx) - np.sin(x0)) / dx
print(forward_approx)
```

```
0.0678264420177852
```

The true solution is roughly 0.5403, so this is not a particularly good approximation. That is to be expected. Remember that all of our analysis in the last class relied on the idea that Δx^n got very close to zero as the power of n got larger. This is valid when Δx is small, but not true at all when Δx is large. This means that we have no reason to expect our approximation is any good when Δx is large.

We could try the same thing with a smaller Δx.

```
dx = 0.1
forward_approx = (np.sin(x0 + dx) - np.sin(x0)) / dx
print(forward_approx)
```

```
0.4973637525353891
```

This is certainly a better approximation than before, but it is still not particularly close. Let's continue this process with smaller and smaller Δx's. To make our analysis more convenient, we will collect all of our approximations in an array named `forward_approx`.

```
forward_approx = np.zeros(5)
for k in range(5):
    dx = (1e-1)**k
    forward_approx[k] = (np.sin(x0 + dx) - np.sin(x0)) / dx
print(forward_approx.reshape(-1,1))
```

```
[[0.06782644]
 [0.49736375]
 [0.53608598]
 [0.53988148]
 [0.54026023]]
```

At each step we reduced Δx by a factor of 10. As you can see, the approximations get closer and closer to the correct answer. By the time we get to $\Delta x = 0.0001$, we have the correct answer to at least four decimal places.

We are interested in how the error changes, so it will be more useful to look at the error between our approximations and the true solutions.

```
print(abs((forward_approx - true_solution).reshape(-1,1)))
```

```
[[4.72475864e-01]
 [4.29385533e-02]
 [4.21632486e-03]
 [4.20825508e-04]
 [4.20744495e-05]]
```

Notice the pattern of the errors. At each step, the error gets divided by 10. We also know that we divided Δx by 10 at each step, so this really is a first order method.

Backward Scheme

Now we will do the same thing with the backward scheme

$f'(x_0) \approx \frac{f(x_0)-f(x_0-\Delta x)}{\Delta x}$.

The code looks almost exactly the same as before.

```
backward_approx = np.zeros(5)
for k in range(5):
    dx = (1e-1)**k
    backward_approx[k] = (np.sin(x0) - np.sin(x0 - dx)) / dx
print(backward_approx.reshape(-1,1))
```

```
[[0.84147098]
 [0.58144075]
 [0.54450062]
 [0.54072295]
 [0.54034438]]
```

Again, these appear to be converging to the true solution (≈ 0.5403) as Δx gets smaller. We can also look at the error like before:

```
print((backward_approx - true_solution).reshape(-1,1))
```

```
[[3.01168679e-01]
 [4.11384459e-02]
 [4.19831487e-03]
 [4.20645407e-04]
 [4.20726487e-05]]
```

Once again, it looks like the error goes down by a factor of 10 at each step. (Actually, that does not appear to be true at the first step, where we reduced Δx from 1 to 0.1. Remember, though, that our results are only really supposed to be valid for sufficiently small Δx.) Once Δx drops below about 0.1, it looks like every time we reduce Δx by a factor of ten, the error also goes down by a factor of ten. This means that the backward difference scheme is also first order.

Central Scheme

Now we will do the same thing with the central scheme

$f'(x) \approx \frac{f(x_0+\Delta x)-f(x_0-\Delta x)}{2\Delta x}$.

Once again, the code looks essentially the same

```
central_approx = np.zeros(5)
for k in range(5):
    dx = (1e-1)**k
    central_approx[k] = (np.sin(x0 + dx) - np.sin(x0 - dx)) / (2 * dx)
print(central_approx.reshape(-1,1))
```

```
[[0.45464871]
 [0.53940225]
 [0.5402933 ]
 [0.54030222]
 [0.5403023 ]]
```

As with the other two schemes, it seems clear that these are approaching the true solution 0.5403, but we need to look at the error values to see how fast we are approaching the correct answer.

```
print((central_approx - true_solution).reshape(-1,1))
```

```
[[-8.56535925e-02]
 [-9.00053698e-04]
 [-9.00499341e-06]
 [-9.00504503e-08]
 [-9.00429620e-10]]
```

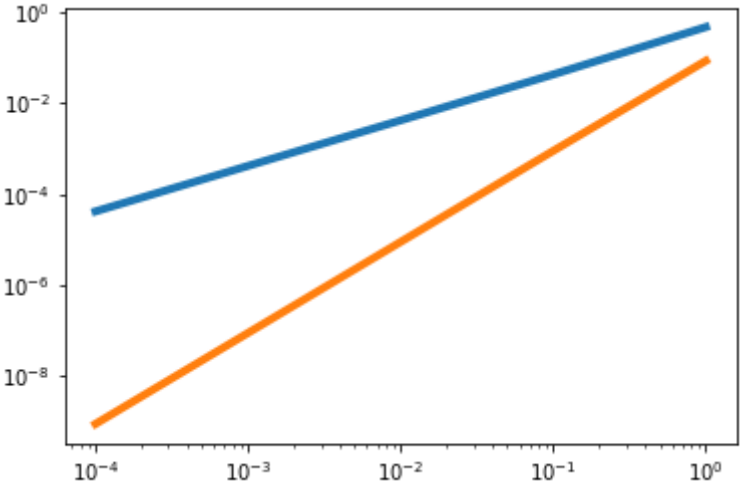
This time the error values look substantially different. In particular, they are decreasing by a factor of 100 every time we reduce Δx by a factor of 10. This means that the central difference scheme really is second order.

How good is our approximation?

Lets compare the forward and central difference schemes

```
dx = (1e-1)**np.array([0, 1, 2, 3, 4])
plt.loglog(dx, abs(forward_approx - true_solution), dx, abs(central_approx - true_solution), linewidth = 4)
```

```
[<matplotlib.lines.Line2D at 0x7f911d69f2d0>,
 <matplotlib.lines.Line2D at 0x7f911dd55c90>]
```




```
Slope1 = (np.log10(abs(forward_approx[-1] - true_solution))-np.log10(abs(forward_approx[0] - true_solution)))/(np.
Slope2 = (np.log10(abs(central_approx[-1] - true_solution))-np.log10(abs(central_approx[0] - true_solution)))/(np.
print('\n Forward Slope = ', Slope1, '\n Central Slope = ', Slope2)
```

```
 Forward Slope =  1.0125902962860651
 Central Slope =  1.9945739524957418
```

Notice that the second slope is twice that of the first. Since we are using log-log scale, the accuracy of central approximation will be the square of the first. To convince you lets do the following:

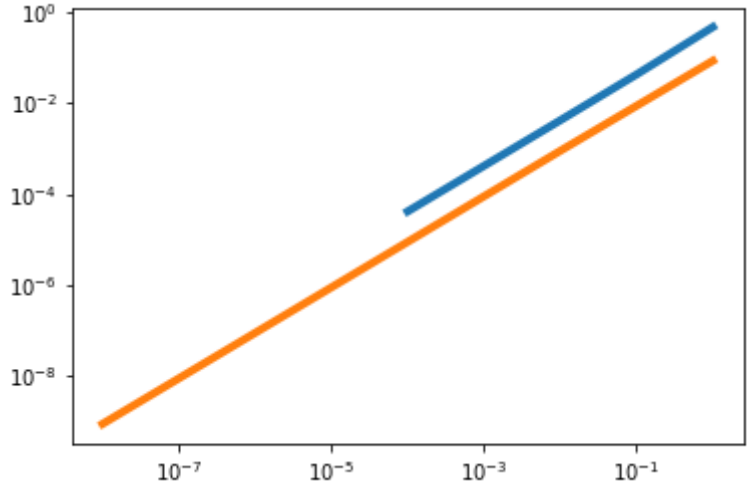
```
print('\n Central Slope with dx^2 = ', (np.log10(abs(central_approx[-1] - true_solution))-np.log10(abs(central_app
```

```
Central Slope with dx^2 =  0.9999697746748636
```

Viola! After we square the timestep we see that this is the same value as forward differences. Actually this tells us for this particular case central differences is even more accurate than we theorized. We can also see this visually:

```
plt.loglog(dx, abs(forward_approx - true_solution), dx**2, abs(central_approx - true_solution), linewidth = 4)
```

```
[<matplotlib.lines.Line2D at 0x7f911d483790>,
 <matplotlib.lines.Line2D at 0x7f911d384ed0>]
```



Approximating over entire intervals

Now let's consider a very common situation: We have a list of x values $x_0 < x_1 < \cdots < x_N$. We want to approximate $f'(x_k)$ at *every* point in this list. For example, suppose $f(x) = \sin(x)$ as before and we have the x values 0, 0.1, 0.2, ..., 5.9, 6.

```
x = np.arange(0, 6.1, 0.1)
n = x.size
```

We will collect all of these approximations in a vector named `deriv`, so that the first entry of `deriv` corresponds to the derivative at

```
deriv = np.zeros(n)
```

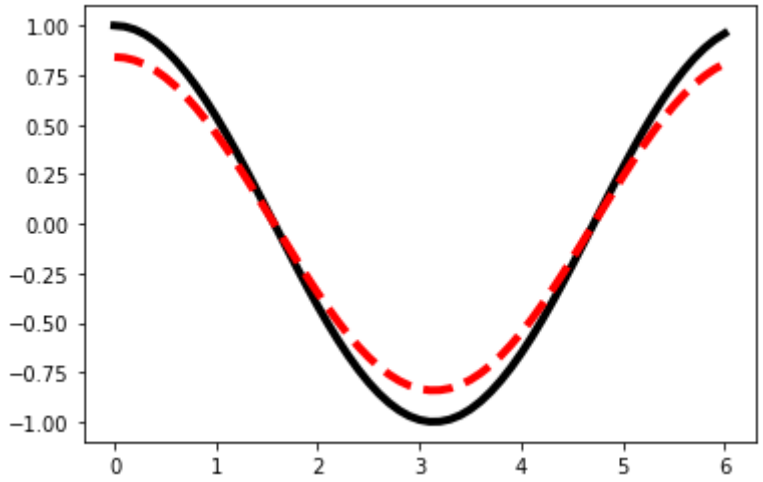
We can use any of our difference schemes for this problem. For no particular reason, let's use the central difference scheme for each point. We have to choose some value for Δx as well, so let's start with $\Delta x = 1$.

```
dx = 1
for k in range(n):
    deriv[k] = (np.sin(x[k] + dx) - np.sin(x[k] - dx)) / (2 * dx)
```

We can see how good our approximation is in a couple of ways. To start, we could plot our approximation alongside the true solution $f'(x) = \cos(x)$.

```
plt.plot(x, np.cos(x), 'k', x, deriv, 'r--', linewidth = 4)
```

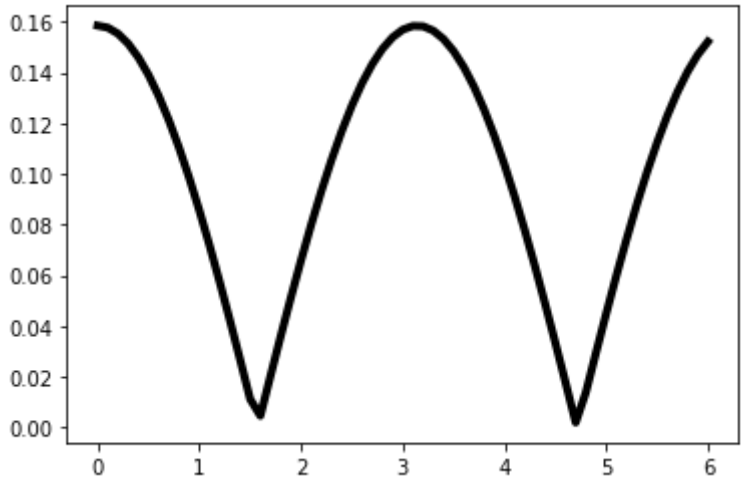
```
[<matplotlib.lines.Line2D at 0x7f911d2f8dd0>,
 <matplotlib.lines.Line2D at 0x7f911d28c8d0>]
```



or we can plot the error (i.e., the difference between our approximation and the true solution).

```
err = abs(deriv - np.cos(x))
plt.plot(x, err, 'k', linewidth = 4)
```

```
[<matplotlib.lines.Line2D at 0x7f911d23cf90>]
```



Notice the scale on the y axis here. Our approximation is off by roughly ± 0.2. If we decrease Δx, we can make this more accurate. For example,

```
dx = 0.01
deriv = np.zeros(n)
for k in range(n):
    deriv[k] = (np.sin(x[k] + dx) - np.sin(x[k] - dx)) / (2 * dx)
err = abs(deriv - np.cos(x))
plt.plot(x, err, 'k', linewidth = 4)
```

```
[<matplotlib.lines.Line2D at 0x7f911d1b75d0>]
```

Again, note the scale on the y axis. We reduced Δx by a factor of 100 and the error fell by a factor of $100^2 = 10,000$. This once again confirms that the central scheme is second order.

Older notes:

Approximating Derivatives From Data

To see why we might care about higher order schemes, let's change our scenario slightly. Instead of knowing $f(x)$ and a list of x values, let's instead suppose that we only have a list of points: $(x_0, y_0), (x_1, y_1), \ldots, (x_N, y_N)$. We assume that these y values come from some function $f(x)$, so that $f(x_k) = y_k$, but we don't know the formula for this function. We will further assume that the x values are evenly spaced and we will call the distance between two subsequent x values Δx. That is, $x_k = x_0 + k\Delta x$. Everything we do below still works fine if the x values are not evenly spaced, but it becomes much more tedious to write down the formulas.

For example, suppose we are given the data

[ ] ↳ *27 cells hidden*