Numerical Differentiation

Phy 379

Dave Pawlowski

Data isn't provided in a nice continuous fashion

Data doesn't usually correspond to an exact analytic expression

But we still have to operate on data: addition, subtraction, multiplication... EASY

Differentiation and integration is a little more tricky.

Numerical Differentiation is a technique that produces an **estimate** of the derivative of a mathematical function.

In order to do this, the function must be discretized, or broken up into chunks and evaluated at various grid points.

Several ways to do this. We will use finite differencing.

Analytically, we know that the derivative of a function, f, is

$$f' = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Computationally, we know that we are unable to let h = 0, as that will return, at worst, an error, or at best, infinity, which is still not the correct answer.

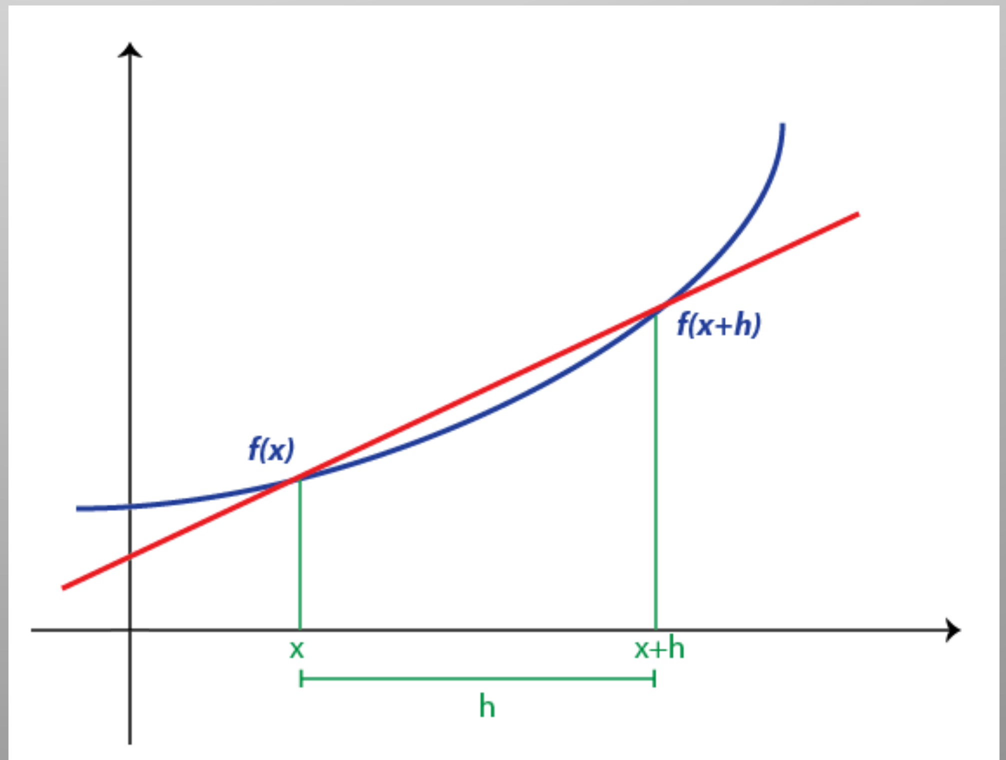
However, we are able to let h become very small. In the computer world, *h* is known as the step size or the grid size.

In order to estimate the derivative of the function, we need to evaluate the function at each step:

$$\ldots x-3h, x-2h, x-h, x, x+1, x+2h, x+3h, \ldots$$

Then, on a grid point by grid point basis, you can take the derivative utilizing finite differencing:

$$f' = \frac{f(x+h)-f(x)}{h}$$



Where did this equation come from?

Take the Taylor series of *f(x+h)*:

$$f(x+h) = f(x) + \frac{f'(x)h}{1!} + \frac{f''(x)h^2}{2!} + \frac{f'''(x)h^3}{3!} + ...$$

Keeping only the first 2 terms on the right hand side of the equation you get:

$$f(x+h) = f(x) + f'(x)h$$

and solving for *f'*:

$$f'(x) = \frac{f(x+h) - f(x)}{h}$$

This is referred to as a **forward differencing scheme**.

A note on Error

Since we threw out all the terms after the 1st derivative in the Taylor series, we have induced error into our equation. This is referred to **truncation error**, since we truncated the infinite series.

There are two primary sources of error in a numerical problem: truncation error and round-off error (ignoring *human stupidity error)*.

Round-off error is caused by the fact that computers cannot store numbers to infinite decimal points. Think of it this way: If asked to store the number

number = 1/3.0

one expects the answer to be, represented in decimal notation, 0.33333333..., where the 3 repeats infinitely. A computer cannot store the number this way, and therefore at some point must round:

print number

number = 0.3333330

The number of decimal places that a computer uses depend on the precision of the variable and the number of memory addressing registers, or bytes, that the computer uses for storage.

Typically, round of error is more than small enough for scientific calculations (~ 10^-16 for double precision numbers).

It turns out that round-off error can be quite important when performing numerical differentiation, since we are subtracting two numbers that are likely to be quite similar: f(x) and f(x+h) for forward differencing. This means that while we want h to get as small as possible, if it gets too small, our round-off error can start to dominate.

In python, floats are precise to 16 digits-> $C=10^{-16}$ Subtracting 2 numbers results in a worst case error of 2C * f(x). The total error is then:

$$\epsilon = \frac{2C|f(x)|}{h} + \frac{1}{2}h|f''(x)|$$

Taking the derivative and setting it equal to zero gives:

$$-\frac{2C|f(x)|}{h^2} + \frac{1}{2}|f''(x)| = 0$$

or

$$h = \sqrt{4C|\frac{f(x)}{f''(x)}|}$$

Subbing that back in to our error equation, we get:

$$\epsilon = \sqrt{4C|f(x)f''(x)|}$$

So the best that we can hope for when performing numerical differentiation is error that goes as the square root of C, or 10^{-8}, as opposed to 10^{-16}.

In the case of truncation error, the numerical method that is used determines the amount of error involved. It is customary to describe the error in this manner:

$$f'(x) = \frac{f(x+h) - f(x)}{h} + O(h^2)$$

Where $O(h^2)$ tells us that this method is 1st order accurate (we got rid of the terms that had h^2): if we increase the step size by a factor of 2, the solution will be twice as accurate.

It turns out that we can do better.

So, keep the term that has an h^2 in the Taylor series:

$$f(x+h) = f(x) + \frac{f'(x)h}{1!} + \frac{f''(x)h^2}{2!} \quad (1)$$

Since we don't know the 2nd derivative, we need to get rid of it somehow. Luckily, we are able to do that:

$$f(x-h) = f(x) - \frac{f'(x)h}{1!} + \frac{f''(x)h^2}{2!} \quad (2)$$

Subtracting 2 from 1, then rearranging, we get:

$$f(x+h) - f(x-h) = f'(x)h + f'(x)h$$

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^3)$$

This is called a central difference, and it is 2nd order.

So now the question is, how do you solve this equation on a computer? Let's use the central difference as an example.

In order to get the derivative at a point, x, we need to evaluate the function at the points x +/- h. That's easy, except at x_0.

At the first data point, we don't know x – h. Similarly, at the last data point, we don't know x + h.

Point number, n	x	f'(x_n) = (x(n+1)-x(n-1))/(2 delta(n))
0	10	
1	20	10
2	30	10
3	40	10
4	50	10
5	60	10
6	70	10
7	80	10
8	90	10
9	100	10
10	110	

In some cases, we may not care about those points, and can just evaluate the solution in the range from n = 1 to n – 1. In other cases, something may have to be said about the **boundary.**

For example, we may determine that the function should be continuous at the boundary, or alternatively, the derivative of the function should be continuous.

There are many possibilities for the boundary conditions of a numerical problem. Its always a good idea to try and choose boundary conditions that are physically realistic.

Once we figure out what to do about the boundary, we can use a computer to solve the problem. Clearly we will be iterating over a range of values, in the example above, from n = 0 to n = 10, or something similar. The general framework of your solver will look something like this:

```
stepsize = 0.01
i = 1
while i<10:
    x_prime[i] = (x[i+1] – x[i-1])/(2*stepsize)
     i = i + stepsize
```

You will write a solver in your next HW assignment, so I won't do more work for you here.