Differentiation

1. Basic idea of differentiation

What is the definition of a derivative?

Can you suggest a way to estimate a derivative numerically?

How can you estimate the level of approximation error for a given choice of $\mathrm{d}x$?

An obvious issue with forward differencing is that there is no reason to choose "forward" over "backward" differencing, yet in general they will yield different answers. This asymmetry is an undesirable feature, since there are many cases where you would like to be able to do the same operation forwards or backwards and get the same answer.

2. Practical differentiation estimates

A better method of approximating the derivative with the same number of function evaluations is the "central difference" algorithm. This approximation is:

$$\left.\frac{\mathrm{d}f}{\mathrm{d}x}\right|_{\mathrm{cd}} = \frac{f(x+\mathrm{d}x/2) - f(x-\mathrm{d}x/2)}{\mathrm{d}x} \tag{1}$$

This is obviously symmetric, and you are performing still only two function evaluations.

Beyond that, it has a nice advantage that is revealed when you estimate the approximation error associated with it.

How can you estimate the approximation error?

The key result is that the approximation error scales as $\mathrm{d}x^2$ instead of $\mathrm{d}x$. This means that for small $\mathrm{d}x$ the approximation is much better. For example, if you happen to be estimating the derivative of a parabola, for which all third and higher derivatives are zero, the estimate of the derivative will have no approximation error.

There is a yet more clever option. Consider the estimate:

$$D_1 = \frac{f(x+\mathrm{d}x/2) - f(x-\mathrm{d}x/2)}{\mathrm{d}x} = \left.\frac{\mathrm{d}f}{\mathrm{d}x}\right|_x = -\frac{1}{3!}\frac{\mathrm{d}x^2}{4}\left.\frac{\mathrm{d}^3 f}{\mathrm{d}x^3}\right|_x + \mathcal{O}(\mathrm{d}x^4) \tag{2}$$

If I reduce $\mathrm{d}x$ by a factor of two I get:

$$D_2 = \frac{f(x+\mathrm{d}x/4) - f(x-\mathrm{d}x/4)}{\mathrm{d}x} = \left.\frac{\mathrm{d}f}{\mathrm{d}x}\right|_x - \frac{1}{4}\frac{1}{3!}\frac{\mathrm{d}x^2}{4}\left.\frac{\mathrm{d}^3 f}{\mathrm{d}x^3}\right|_x + \mathcal{O}(\mathrm{d}x^4) \tag{3}$$

Now if I take a new estimate:

$$D = \frac{4D_2 - D_1}{3} \tag{4}$$

You see that the first terms leave exactly the derivative, and the second order terms cancel leaving:

$$D = \left.\frac{\mathrm{d}f}{\mathrm{d}x}\right|_x + \mathcal{O}(\mathrm{d}x^4) \tag{5}$$

This very good approximation of course comes at the expense of more function evaluations!

3. Error assessment

The question arises as to what to choose for $\mathrm{d}x$. The optimal choice will be about when the round-off error is similar to the approximation error.

For the central difference approximation, this yields:

$$\epsilon_{\mathrm{approx}} = \frac{f'''\mathrm{d}x^2}{24} \tag{6}$$

Assuming the function is of order unity, or at least far enough from overflow or underflow, the round-off error in the numerator is the machine precision ϵ_{m}, and so the final round-off is:

$$\epsilon_{\mathrm{ro}} = \frac{\epsilon_{\mathrm{m}}}{\mathrm{d}x} \tag{7}$$

Setting these two equal to each other yields:

$$\mathrm{d}x = \left(\frac{24\epsilon_{\mathrm{m}}}{f'''}\right)^{1/3} \tag{8}$$

Note a few things. The $\mathrm{d}x$ value goes as the cube root of the machine precision, and the approximation error goes as the square of $\mathrm{d}x$. This means that as you increase machine precision the best approximation error improves as the 2/3 power.

The choice of $\mathrm{d}x$ for double precision ($\epsilon_{\mathrm{m}} \sim 3 \times 10^{-15}$ is about $\mathrm{d}x \sim 10^{-5}$.

Finally, it is important to realize that under most circumstances you do not know beforehand exactly what f''' is. After all, if you did, you would not be doing this derivative calculation!

4. Numpy implementation

In NumPy, the function that implements a derivative is `gradient`, which only works on a function tabulated as an array — i.e. it uses central difference on a set of precomputed points. This is in fact an extremely common use case for derivative calculation.

5. But can't we just take derivatives of functions? "Autodiff"

Evaluating finite differences is unavoidable in many cases; e.g. if I just have a function pre-evaluated on a grid or (as we will see later) in the context of numerically integrating differential equations. But for many cases of interest, the function we are evaluating is just that: a function composed of a bunch of individually differentiable functions.

For example:

$$f(x) = \sin(x^2) \tag{9}$$

is the result of:

$$f(x) = f_1(f_0(x))) \tag{10}$$

where $f_0(x) = x^2$ and $f_1(y) = \sin y$.

When we take the derivative of this function analytically we use the chain rule:

$$\frac{\partial f}{\partial x} = \frac{\partial f_0}{\partial x}\frac{\partial f_1}{\partial y} = 2x\cos(x^2) \tag{11}$$

That is, the derivatives of the constituent functions of the composite function just multiply. Of course sometimes the function has terms (like $f = x^2 + 3x$) but this is even simpler and still is linear (the derivatives of x^2 and $3x$ just add).

This means that when you calculate the function, in principle the computer can calculate its derivative at the same time, as long as it knows the derivatives of all the intermediate functions. The notebooks show an example of implementing this by hand.

The critical point is that this operation is very straightforward to make automated. If every function returns not only its value but its derivative, then you just multiply all of those derivatives in a perfectly mechanical way.

`jax` is one example (but not the only one!) of a numerical library that performs autodiff. `jax` also is designed to compile against GPUs, which are good at highly parallelized computations, like doing the same operation on a bunch of different elements of an array. This is an important machine learning use case, as is autodiff. The notebooks show a simple version of autodiff with `jax`. A good example of how it generalizes is showing that it can take the *second* derivative in the same way.

But the examples so far are pretty trivial in that they are derivatives of a 1-dimensional function. That's not really so impressive, or that useful for the machine learning or other applications (which fit many dimensional functions so need derivatives in *lots* of dimensions). But `jax` and other autodiff implementations keep track of all the derivatives of all the input functions.

See the tutorial in Kyle Cranmer's book for an extensive description.

6. Scaling a problem

Note that in a lot of analyses like that above we assume the quantities we are dealing with are far away from overflow or underflow. Indeed, it is good practice to maximize one's dynamic range by dealing in units that are transformed.

For example, in a gravitational context we might calculate a force from the gradient of the potential.

What is the force equation for gravity in spherical symmetry?

Imagine we are calculating the acceleration on an object near the surface of the Sun. Near the surface of the Sun we have $G - 6.67 \times 10^{-11}$ m^3 kg^{-1} s^{-2}, $M = 2 \times 10^{30}$ kg, $r \sim 10^9$ m. So $\phi \sim 10^{28}$, and $F \sim 10^{19}$.

Two things to note: first, it would be "nicer" if these numbers were closer to unity. In the course of a full calculation of, say, an orbit, these numbers will vary. Also, we will be calculating other numbers (like the position and velocity of the object). We want to minimize the chance that any of these numbers will overflow or underflow. So we should make the natural units of the problem that the *computer* sees as close to unity as we can.

This is especially true if we are working in single rather than double precision. It is not *usually* a good reason to work in double precision just to prevent overflow and underflow. Usually single precision is sufficient from that point of view with a wise choice of units.

Second, there are scaling relations that this set of equations has to obey, as we will see in a second. We can solve one problem and for a large set of situations can scale our results to that new situation.

Specifically we can redefine:

$$\begin{aligned} r' &= \frac{r}{R_\odot} \\ t' &= \sqrt{\frac{GM_\odot}{R_\odot}} \end{aligned} \tag{12}$$

and we find:

$$\frac{\mathrm{d}^2 r'}{\mathrm{d}t'^2} = \frac{\mathrm{d}}{\mathrm{d}r'}\frac{1}{r'} \tag{13}$$

with $r' \sim 1$ (given that we are working near the surface of the Sun). Now there are no stray units. If the derivative on the right hand side yields something far from unity, then this is an unavoidable aspect of the problem, but we have done our best to keep things in range.

Also, if we tabulate results in terms of r' and t', we can adjust to a different length scale $R_{\rm new}$ and mass $M_{\rm new}$ through the above equations, instead of recalculating the whole problem.

Pulling the dimensions out of a problem like this is a generic strategy in numerical physics.