4 Finite-Difference Approximations

In this section, we will be referring to this Julia notebook for calculations that are not included here.

4.1 Why compute derivatives approximately instead of exactly?

Working out derivatives by hand is a notoriously error-prone procedure for complicated functions. Even if every individual step is straightforward, there are so many opportunities to make a mistake, either in the derivation or in its implementation on a computer. Whenever you implement a derivatives, you should **always double-check** for mistakes by comparing it to an independent calculation. The simplest such check is a *finite-difference approximation*, in which we *estimate* the derivative(s) by comparing $f(x)$ and $f(x + \delta x)$ for one or more "finite" (non-infinitesimal) perturbations δx.

There are many finite-difference techniques at varying levels of sophistication, as we will discuss below. They all incur an intrinsic **truncation error** due to the fact that δx is not infinitesimal. (And we will also see that you can't make δx too small, either, or *roundoff errors* start exploding!) Moreover, finite differences become expensive for higher-dimensional x (in which you need a separate finite difference for each input dimension to compute the full Jacobian). This makes them an approach of last resort for computing derivatives accurately. On the other hand, they are the *first* method you generally employ in order to *check* derivatives: if you have a bug in your analytical derivative calculation, usually the answer is completely wrong, so even a crude finite-difference approximation for a single small δx (chosen at random in higher dimensions) will typically reveal the problem.

Another alternative is **automatic differentiation** (AD), software/compilers perform *analytical* derivatives for you. This is extremely reliable and, with modern AD software, can be very efficient. Unfortunately, there is still lots of code, e.g. code calling external libraries in other languages, that AD tools can't comprehend. And there are other cases where AD is inefficient, typically because it misses some mathematical structure of the problem. Even in such cases, you can often fix AD by defining the derivative of one small piece of your program by hand,[2] which is much easier than differentiating the whole thing. In such cases, you still will typically want a finite-difference check to ensure that you have not made a mistake.

It turns out that finite-difference approximations are a surprisingly complicated subject, with rich connections to many areas of numerical analysis; in this lecture we will just scratch the surface.

4.2 Finite-Difference Approximations: Easy Version

The simplest way to check a derivative is to recall that the definition of a differential:

$$df = f(x + dx) - f(x) = f'(x)dx$$

came from dropping higher-order terms from a small but finite difference:

$$\delta f = f(x + \delta x) - f(x) = f'(x)\delta x + o(\|\delta x\|)\,.$$

So, we can just compare the **finite difference** $\boxed{f(x + \delta x) - f(x)}$ to our **(directional) derivative operator** $f'(x)\delta x$ (i.e. the derivative in the direction δx). $f(x+\delta x)-f(x)$ is also called a **forward difference** approximation. The antonym of a forward difference is a **backward difference** approximation $f(x) - f(x - \delta x) \approx f'(x)\delta x$. If you just want to compute a derivative, there is not much practical distinction between forward and backward differences.

[2]In some Julia AD software, this is done with by defining a "ChainRule", and in Python autograd/JAX it is done by defining a custom "vJp" (row-vector—Jacobian product) and/or "Jvp" (Jacobian–vector product).

The distinction becomes more important when discretizing (approximating) differential equations. We'll look at other possibilities below.

Remark 29. *Note that this definition of forward and backward difference is **not** the same as forward- and backward-mode differentiation—these are **unrelated** concepts.*

If x is a scalar, we can also divide both sides by δx to get an approximation for $f'(x)$ instead of for df:

$$f'(x) \approx \frac{f(x+\delta x) - f(x)}{\delta x} + (\text{higher-order corrections}) \,.$$

This is a more common way to write the forward-difference approximation, but it only works for scalar x, whereas in this class we want to think of x as perhaps belonging to some other vector space.

Finite-difference approximations come in many forms, but they are generally a **last resort** in cases where it's too much effort to work out an analytical derivative and AD fails. But they are also useful to **check** your analytical derivatives and to quickly **explore**.

4.3 Example: Matrix squaring

Let's try the finite-difference approximation for the square function $f(A) = A^2$, where here A is a square matrix in $\mathbb{R}^{m,m}$. By hand, we obtain the product rule

$$df = A\,dA + dA\,A,$$

i.e. $f'(A)$ is the **linear operator** $\boxed{f'(A)[\delta A] = A\,\delta A + \delta A\,A.}$ This is *not equal to* $2A\,\delta A$ because *in general* A and δA do not commute. So let's check this difference against a finite difference. We'll try it for a *random* input A and a *random small* perturbation δA.

Using a random matrix A, let $dA = A \cdot 10^{-8}$. Then, you can compare $f(A + dA) - f(A)$ to $A\,dA + dA\,A$. If the matrix you chose was really random, you would get that the approximation minus the exact equality from the product rule has entries with order of magnitude around 10^{-16}! However, compared to $2AdA$, you'd obtain entries of order 10^{-8}.

To be more quantitative, we might compute that "norm" $\|\text{approx} - \text{exact}\|$ which we want to be small. But small **compared to what**? The natural answer is **small compared to the correct answer.** This is called the relative error (or "fractional error") and is computed via

$$\text{relative error} = \frac{\|\text{approx} - \text{exact}\|}{\|\text{exact}\|}.$$

Here, $\|\cdot\|$ is a norm, like the length of a vector. This allows us to understand the size of the error in the finite difference approximation, i.e. it allows us to answer how accurate this approximation is (recall Sec. 4.1).

So, as above, you can compute that the relative error between the approximation and the exact answer is about 10^{-8}, where as the relative error between $2AdA$ and the exact answer is about 10^0. This shows that our exact answer is likely correct! Getting a good match up between a random input and small displacement isn't a proof of correctness of course, but it is always a good thing to check. This kind of randomized comparison will almost always **catch major bugs** where you have calculated the symbolic derivative incorrectly, like in our $2AdA$ example.

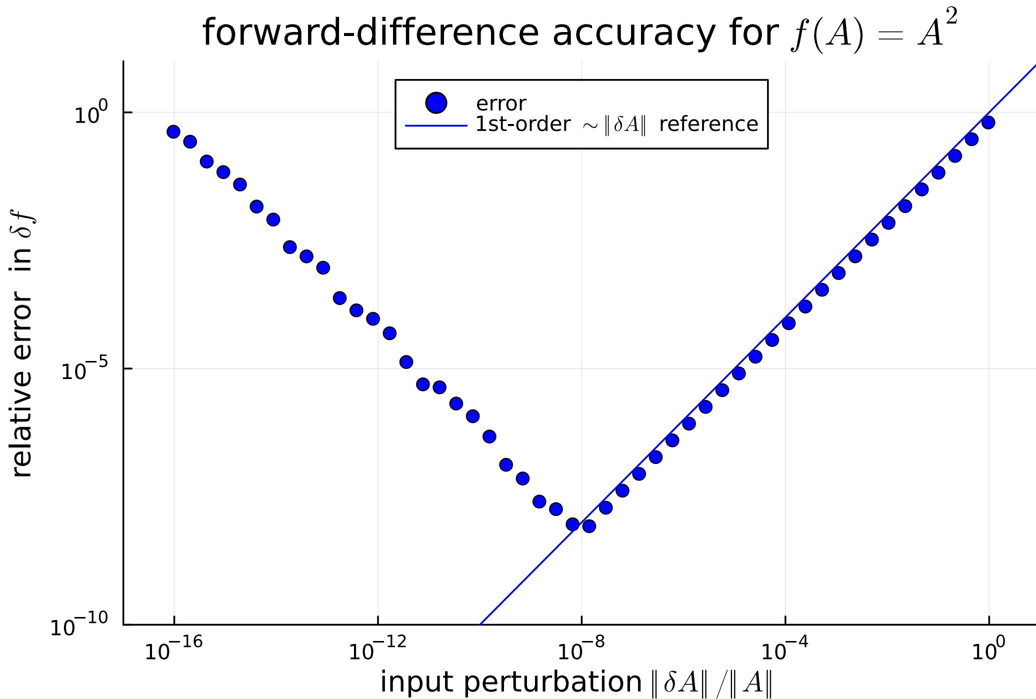

Figure 4: Forward-difference accuracy for $f(A) = A^2$, showing the relative error in $\delta f = f(A + \delta A) - f(A)$ versus the linearization $f'(A)\delta A$, as a function of the magnitude $\|\delta A\|$. A is a 4×4 matrix with unit-variance Gaussian random entries, and δA is similarly a unit-variance Gaussian random perturbation scaled by a factor s ranging from 1 to 10^{-16}.

Definition 30

Note that the norm of a matrix that we are using, computed by `norm(A)` in Julia, is just the direct analogue of the familiar Euclidean norm for the case of vectors. It is simply the square root of the sum of the matrix entries squared:

$$\|A\| := \sqrt{\sum_{i,j} |A_{ij}|^2} = \sqrt{\operatorname{tr}(A^T A)}\,.$$

This is called the Frobenius norm.

4.4 Accuracy of Finite Differences

Now how accurate is our finite-difference approximation above? How should we choose the size of δx?

Let's again consider the example $f(A) = A^2$, and plot the relative error as a function of $\|\delta A\|$. This plot will be done *logarithmically* (on a log–log scale) so that we can see power-law relationships as straight lines.

We notice two main features as we decrease δA:

1. The relative error at first decreases linearly with $\|\delta A\|$. This is called **first-order accuracy**. Why?

2. When δA gets too small, the error increases. Why?

4.5 Order of accuracy

The **truncation error** is the inaccuracy arising from the fact that the input perturbation δx is not infinitesimal: we are computing a difference, not a derivative. If the truncation error in the derivative scales proportional $\|\delta x\|^n$, we call the approximation **n-th order accurate**. For forward differences, here, the order is **n=1**. Why?

For any $f(x)$ with a nonzero second derivative (think of the Taylor series), we have

$$f(x+\delta x) = f(x) + f'(x)\delta x + (\text{terms proportional to } \|\delta x\|^2) + \underbrace{o(\|\delta x\|^2)}_{\text{i.e. higher-order terms}}$$

That is, the terms we *dropped* in our forward-difference approximations are proportional to $\|\delta x\|^2$. But that means that the **relative error is linear**:

$$\begin{aligned}\text{relative error} &= \frac{\|f(x+\delta x) - f(x) - f'(x)\delta x\|}{\|f'(x)\delta x\|} \\ &= \frac{(\text{terms proportional to } \|\delta x\|^2) + o(\|\delta x\|^2)}{\text{proportional to } \|\delta x\|} = (\text{terms proportional to } \|\delta x\|) + o(\|\delta x\|)\end{aligned}$$

This is **first-order accuracy**. Truncation error in a finite-difference approximation is the **inherent** error in the formula for **non-infinitesimal** δx. Does that mean we should just make δx as small as we possibly can?

4.6 Roundoff error

The reason why the error *increased* for very small δA was due to **roundoff errors**. The computer only stores a **finite number of significant digits** (about 15 decimal digits) for each real number and rounds off the rest on each operation — this is called floating-point arithmetic. If δx is too small, then the difference $f(x+\delta x) - f(x)$ gets rounded off to zero (some or all of the *significant digits cancel*). This is called catastrophic cancellation.

Floating-point arithmetic is much like scientific notation $*.***** \times 10^e$: a finite-precision coefficient $*.*****$ scaled by a power of 10 (or, on a computer, a power of 2). The number of digits in the coefficient (the "significant digits") is the "precision," which in the usual 64-bit floating-point arithmetic is charactized by a quantity $\epsilon = 2^{-52} \approx 2.22 \times 10^{-16}$, called the machine epsilon. When an arbitrary real number $y \in \mathbb{R}$ is rounded to the closest floating-point value $\tilde{y}$, the roundoff error is bounded by $|\tilde{y} - y| \le \epsilon|y|$. Equivalently, the computer keeps only about 15–16 $\approx -\log_{10}\epsilon$ decimal digits, or really $53 = 1 - \log_2 \epsilon$ *binary* digits, for each number.

In our finite-difference example, for $\|\delta A\|/\|A\|$ of roughly $10^{-8} \approx \sqrt{\epsilon}\|A\|$ or larger, the approximation for $f'(A)$ is dominated by the truncation error, but if we go smaller than that the relative error starts increasing due to roundoff. Experience has shown that $\|\delta x\| \approx \sqrt{\epsilon}\|x\|$ is often a good rule of thumb—about half the significant digits is the most that is reasonably safe to rely on—but the precise crossover point of minimum error depends on the function f and the finite-difference method. But, like all rules of thumb, this may not always be completely reliable.

4.7 Other finite-difference methods

There are more sophisticated finite-difference methods, such as Richardson extrapolation, which consider a sequence of progressively smaller δx values in order to adaptively determine the best possible estimate for f' (*extrapolating* to $\delta x \to 0$ using progressively higher degree polynomials). One can also use higher-order difference formulas than the simple forward-difference method here, so that the truncation error decreases faster than than linearly with δx. The most famous higher-order formula is the "centered difference" $f'(x)\delta x \approx [f(x+\delta x) - f(x-\delta x)]/2$, which has *second*-order accuracy (relative truncation error proportional to $\|\delta x\|^2$).

Higher-dimensional inputs x pose a fundamental computational challenge for finite-difference techniques, because if you want to know what happens for every possible direction δx then you need many finite differences: one for each dimension of δx. For example, suppose $x \in \mathbb{R}^n$ and $f(x) \in \mathbb{R}$, so that you are computing $\nabla f \in \mathbb{R}^n$; if you want to know the whole gradient, you need n *separate* finite differences. The net result is that finite differences in higher dimensions are expensive, quickly becoming impractical for high-dimensional optimization (e.g. neural networks) where n might be huge. On the other hand, if you are just using finite differences as a check for bugs in your code, it is usually sufficient to compare $f(x + \delta x) - f(x)$ to $f'(x)[\delta x]$ in a few random directions, i.e. for a few random small δx.

MIT OpenCourseWare
https://ocw.mit.edu

18.S096 Matrix Calculus for Machine Learning and Beyond
Independent Activities Period (IAP) 2023

For information about citing these materials or our Terms of Use, visit: https://ocw.mit.edu/terms.