DRAFT NOTES: Matrix Calculus (for Machine Learning and Beyond)

Lecturers: Alan Edelman and Steven G. Johnson

Notes by Paige Bright

Based on MIT course 18.S096 in IAP 2023

Contents

Introduction **3**

1 Overview and Motivation **5**
- 1.1 Applications . . . 5
- 1.2 First Derivatives . . . 6
- 1.3 Intro: Matrix and Vector Product Rule . . . 8

2 Derivatives as Linear Operators **9**
- 2.1 Revisiting single-variable calculus . . . 9
- 2.2 Linear operators . . . 10
- 2.3 Revisiting multivariable calculus, Part 1: Scalar-valued functions . . . 11
- 2.4 Revisiting multivariable calculus, Part 2: Vector-valued functions . . . 12

3 Derivatives as Linear Operators, ctd. **13**
- 3.1 Revisiting 18.02, Part 2, ctd. . . . 13
- 3.2 The Chain Rule . . . 14
  - 3.2.1 Cost of Matrix Multiplication . . . 15
- 3.3 Beyond 18.02 Derivatives . . . 16

4 Two-by-two Matrix Jacobians **17**
- 4.1 The Matrix Square Function . . . 17
- 4.2 The Jacobian as a Linear Transformation . . . 17

5 Two-by-two Matrix Jacobians, ctd. **19**
- 5.1 Key Kronecker-Product Identity . . . 19
- 5.2 The Jacobian in Kronecker-Product Notation . . . 20

6 Finite-Difference Approximations **21**
- 6.1 Hand-calculated Derivative Rules: Error Prone . . . 21
- 6.2 Finite-Difference Approximations: Easy Version . . . 21
- 6.3 Example: Matrix squaring . . . 22

6.4 Accuracy of Finite Differences . . . 22
6.5 Order of accuracy . . . 23
6.6 Roundoff error . . . 24
6.7 Other finite-difference methods . . . 24

7 Derivatives in General Vector Spaces 25
7.1 A Simple Matrix Dot Product and Norm . . . 25
7.2 Derivatives, Norms, and Banach spaces . . . 28

8 Nonlinear Root-Finding, Optimization, and Adjoint Differentiation 29
8.1 Newton's Method . . . 29
8.1.1 Scalar Functions . . . 29
8.1.2 Multidimensional Functions . . . 29
8.2 Optimization . . . 30
8.2.1 Nonlinear Optimization . . . 30
8.2.2 Engineering/Physical Optimization . . . 32
8.3 Reverse-mode "Adjoint" Differentiation . . . 32

9 Derivative of Matrix Determinant and Inverse 34
9.1 Two Derivations . . . 34
9.2 Applications . . . 35
9.2.1 Characteristic Polynomial . . . 35
9.2.2 The Logarithmic Derivative . . . 35
9.3 Jacobian of the Inverse . . . 35

10 Forward and Reverse-Mode Automatic Differentiation 37
10.1 Automatic Differentiation via Dual Numbers . . . 37
10.1.1 Example: Babylonian square root . . . 38
10.1.2 Dual numbers . . . 38
10.2 Automatic Differentiation via Computational Graphs . . . 38
10.2.1 Reverse Mode Automatic Differentiation on Graphs . . . 43

11 Differentiating ODE solutions 45
11.1 Ordinary differential equations (ODEs) . . . 45
11.2 Sensitivity analysis of ODE solutions . . . 46
11.2.1 Forward sensitivity analysis of ODEs . . . 48
11.2.2 Reverse/adjoint sensitivity analysis of ODEs . . . 49
11.3 Example . . . 50
11.3.1 Forward mode . . . 51
11.3.2 Reverse mode . . . 51
11.4 Further reading . . . 51

12 Calculus of Variations 53
12.1 Functionals: Mapping functions to scalars . . . 53
12.2 Inner products of functions . . . 53
12.3 Example: Minimizing arc length . . . 54

12.4 Euler–Lagrange equations . . . 55

13 Derivatives of Random Functions **57**
13.1 Introduction . . . 57
13.2 Stochastic programs . . . 57
13.3 Stochastic differentials and the reparameterization trick . . . 59
13.4 Handling discrete randomness . . . 62

14 Second Derivatives, Bilinear Forms, and Hessian Matrices **65**
14.1 Quadratic approximation . . . 67
14.2 Hessians and optimization . . . 68
14.2.1 Sequential quadratic programming . . . 68
14.2.2 Computing Hessians . . . 68
14.2.3 Minima, maxima, and saddle points . . . 69
14.3 Further Reading . . . 69

15 Derivatives of Eigenproblems **71**
15.1 Differentiating on the Unit Sphere . . . 71
15.1.1 Special Case: A Circle . . . 71
15.1.2 On the Sphere . . . 71
15.2 Differentiating on Orthogonal Matrices . . . 72
15.2.1 Differentiating the Symmetric Eigendecomposition . . . 72

16 AD on Computational Graphs, ctd. **74**

17 Where We Go From Here **75**

Introduction

These notes are based on the class as it was run for the second time in January 2023, taught by Professors Alan Edelman and Steven G. Johnson at MIT. The previous version of this course, run in January 2022, can be found on OCW here.

Both Professors Edelman and Johnson use he/him pronouns and are in the Department of Mathematics at MIT; Prof. Edelman is also a Professor in the MIT Computer Science and Artificial Intelligence Laboratory (CSAIL) running the Julia lab, while Prof. Johnson is also a Professor in the Department of Physics.

Here is a description of the course.:

> We all know that typical calculus course sequences begin with univariate and vector calculus, respectively. Modern applications such as machine learning and large-scale optimization require the next big step, "matrix calculus" and calculus on arbitrary vector spaces.
>
> This class covers a coherent approach to matrix calculus showing techniques that allow you to think of a matrix holistically (not just as an array of scalars), generalize and compute derivatives of important matrix factorizations and many other complicated-looking operations, and understand how differentiation formulas must be re-imagined in large-scale computing. We will discuss "reverse" ("adjoint", "backpropagation") differentiation and how modern automatic differentiation is more computer science than calculus (it is neither symbolic formulas nor finite differences).

The class involved numerous example numerical computations using the Julia language, which you can install on your own computer following these instructions. The material for this class is also located on GitHub at https://github.com/mitmath/matrixcalc.

1 Overview and Motivation

Firstly, where does matrix calculus fit into the MIT course catalog? Well, there is 18.01: Single Variable Calculus and 18.02: Vector Calculus that students are required to take at MIT. But it seems as though this sequence of material is being cut off arbitrarily:

$$\text{Scalar} \quad ! \quad \text{Vector} \quad ! \quad \text{Matrices} \quad ! \quad \text{Higher-Order Arrays.}$$

After all, this is how the sequence is portrayed in many computer programming languages, including Julia!

In the last decade, linear algebra has taken on larger and larger importance in numerous areas, such as Machine learning, Statistics, Engineering, etc. In this sense, linear algebra has gradually taken over a much larger part of today's tools for lots of areas of study—now everybody needs linear algebra. So it makes sense that we would *want* to do calculus on these higher-order arrays, and it won't be a simple/obvious generalization (for instance, $\frac{d}{dA}A^2 \neq 2A$ for non-scalar matrices A).

More generally, the subjects of *differentiation* and *sensitivity analysis* are much deeper than one might suspect from the simple rules learned in first- or second-semester calculus. Differentiating functions whose inputs and/or outputs are in more complicated vector spaces (e.g. matrices, functions, or more) is one part of this subject. Another topic is the *efficient* evaluation of derivatives of functions involving very complicated calculations, from neural networks to huge engineering simulations—this leads to the topic of "adjoint" or "reverse-mode" differentiation, also known as "backpropagation". *Automatic differentiation* of computer programs by compilers is another surprising topic, in which the computer does something very different from the typical human process of first writing out an explicit symbolic formula and then passing the chain rule through it. These are only a few examples: the key point is that differentiation is more complicated than you may realize, and that these complexities are increasingly relevant for a wide variety of applications.

Let's quickly talk about some of these applications.

1.1 Applications

Applications: Machine Learning

Machine learning has numerous buzzwords associated with it, including but not limited to: parameter optimization, stochastic gradient descent, automatic differentiation, and backpropagation. In this whole collage you can see a fraction of how matrix calculus applies to Machine Learning. It is recommended that you look into some of these topics yourself if you are interested.

Applications: Physical Problems

Large physical simulations, such as engineering-design problems, are increasingly characterized by *huge* numbers of parameters, and the *derivatives* of simulation outputs with respect to these parameters is crucial in order to evaluate sensitivity to uncertainties as well as to apply large-scale optimization.

For example, the shape of an airplane wing might be characterized by thousands of parameters, and if you can compute the derivative of the drag force (from a large fluid-flow simulation) with respect to these parameters then you could optimize the wing shape to minimize the drag for a given lift or other constraints.

An extreme versions of such parameterization is known as "topology optimization," in which the material at "every point" in space is potentially a degree of freedom, and optimizing over these parameters can discover not only a optimal shape but an optimal *topology* (how materials are connected in space, e.g. how many holes are present).

For example, topology optimization has been applied in mechanical engineering to design the cross sections of airplane wings, artificial hips, and more into a complicated lattice of metal struts (e.g. minimizing weight for a given strength).

Applications: Data Science and Multivariable Statistics

In multivariate statistics, models are often framed in terms of matrix inputs and outputs (or even more complicated objects such as tensors). For example, a "simple" linear multivariate matrix model might be $Y(X) = XB+U$, where B is an unknown matrix of coefficients (to be determined by some form of fit/regression) and U is unknown matrix of random noise (that prevents the model from exactly fitting the data). Regression then involves minimizing some function of the error $U(B) = Y \quad XB$ between the model XB and data Y; for example, a matrix norm $\|U\|_F^2 = \operatorname{tr} U^T U$, a determinant $\det U^T U$, or more complicated functions. Estimating the best-fit coefficients B, analyzing uncertainties, and many other statistical analyses require differentiating such functions with respect to B or other parameters. A recent review article on this topic is Liu et al. (2022): "Matrix differential calculus with applications in the multivariate linear model and its diagnostics" (https://doi.org/10.1016/j.sctalk.2023.100274).

Applications: Automatic differentiation

Typical differential calculus classes are based on symbolic calculus, with students essentially learning to do what Mathematica or Wolfram Alpha can do. Even if you are using a computer to take derivatives symbolically, to use this effectively you need to understand what is going on beneath the hood. But while, similarly, some numerics may show up for a small portion of this class (such as to approximate a derivative using the difference quotient), *today's* automatic differentiation is neither of those two things. It is more in the field of the computer science topic of compiler technology than mathematics. However, the underlying mathematics of automatic differentiation is interesting, and we will learn about this in this class!

Even *approximate* computer differentiation is more complicated than you might expect. For single-variable functions $f(x)$, derivatives are defined as the limit of a difference $[f(x + \delta x) \quad f(x)]/\delta x$ as $\delta x \; ! \; 0$. A crude "finite-difference" approximation is simply to approximate $f^0(x)$ by this formula for a small δx, but this turns out to raise many interesting issues involving balancing truncation and roundoff errors, higher-order approximations, and numerical extrapolation.

1.2 First Derivatives

The derivative of a function of one variable is itself a function of one variable– it simply is (roughly) defined as the linearization of a function. I.e., it is of the form $(f(x) \quad f(x_0)) \quad f^0(x_0)(x \quad x_0)$. In this sense, "everything is easy" with scalar functions of scalars (by which we mean, functions that take in one number and spit out one number).

There are occasionally other notations used for this linearization:

$\delta y \quad f^0(x)\delta x,$

$dy = f^0(x)dx,$

$(y \quad y_0) \quad f^0(x_0)(x \quad x_0),$

and $df = f^0(x)dx.$

This last one will be the preferred of the above for this class. One can think of dx and dy as "really small numbers." In mathematics, they are called infinitesimals, defined rigorously via taking limits. Note that here we do not want

to divide by dx. While this is completely fine to do with scalars, once we get to vectors and matrices you can't always divide!

The numerics of such derivatives are simple enough to play around with. For instance, consider the function $f(x) = x^2$ and the point $(x_0, f(x_0)) = (3, 9)$. Then, we have the following numerical values near $(3, 9)$:

$$f(3.\mathbf{0001}) = 9.\mathbf{0006}0001$$
$$f(3.\mathbf{00001}) = 9.\mathbf{00006}00001$$
$$f(3.\mathbf{000001}) = 9.\mathbf{000006}000001$$
$$f(3.\mathbf{0000001}) = 9.\mathbf{0000006}0000001.$$

Here, the bolded digits on the left are Δx and the bolded digits on the right are Δy. Notice that $\Delta y = 6\Delta x$! Hence, we have that

$$f(3 + \Delta x) = 9 + \Delta y = 9 + 6\Delta x \quad =) \quad f(3 + \Delta x) \quad f(3) = 6\Delta x \quad f^0(3)\Delta x.$$

Therefore, we have that the linearization of x^2 at $x = 3$ is the function $f(x) \quad f(3) \quad 6(x \quad 3)$.

We now leave the world of scalar calculus and enter the world of vector/matrix calculus! Professor Edelman invites us to think about matrices *holistically*—not just as a table of numbers.

The notion of linearizing your function will conceptually carry over as we define the derivative of functions which take in/spit out more than one number. Of course, this means that the derivative will have a different "shape" than a single number. Here is a table on the *shape* of the first derivative. The inputs of the function are given on the left hand side of the table, and the outputs of the function are given across the top.

input # and output !	scalar	vector	matrix
scalar	scalar	vector (for instance, velocity)	matrix
vector	gradient = (column) vector	matrix (called the Jacobian matrix)	higher order array
matrix	matrix	higher order array	higher order array

You will ultimately learn how to do any of these in great detail eventually in this class! The purpose of this table is to plant the notion of differentials as linearization. Let's look at an example.

Example 1

Let $f(x) = x^T x$, where x is a 1 2 matrix and the output is thus a 1 1 matrix. Confirm that $2x_0^T dx$ is indeed the differential of f at $x_0 = \begin{bmatrix} 3 & 4 \end{bmatrix}^T$.

Firstly, let's compute $f(x_0)$:

$$f(x_0) = x_0^T x_0 = 3^2 + 4^2 = 25.$$

Then, suppose $dx = [.001, .002]$. Then, we would have that

$$f(x + dx) = (3.001)^2 + (4.002)^2 = 25.\mathbf{022}005.$$

Then, notice that $2x_0^T\, dx = 2 \begin{bmatrix} 3 & 4 \end{bmatrix}^T dx = .022$. Hence, we have that

$$f(x_0 + dx) \quad f(x_0) \quad 2x_0^T dx = .022.$$

As we will see right now, the $2x_0^T dx$ didn't come from nowhere!

1.3 Intro: Matrix and Vector Product Rule

For matrices, we in fact still have a product rule!

Theorem 2 (Di□erential Product Rule)

Let A, B be two matrices. Then, we have the differential product rule for AB:

$$d(AB) = (dA)B + A(dB).$$

By the differential of the matrix A, we think of it as a small (unconstrained) change in the matrix A. Later, constraints may be places on the allowed perturbations (see Lecture 6).

Notice however, that (by our table) the derivative of a matrix is a matrix! So generally speaking, the products will not commute.

If x is a vector, then by the differential product rule we have

$$d(x^T x) = (dx^T)x + x^T(dx).$$

However, notice that this is a dot product, and dot products commute (since $\sum a_i \;\; b_i = \sum b_i \;\; a_i$), we have that

$$d(x^T x) = (2x)^T dx.$$

Remark 3. *The way the product rule works for vectors as matrices is that transposes "go for the ride." See the next example below.*

Example 4

By the product rule. we have

1. $d(u^T v) = (du)^T v + u^T(dv) = v^T du + u^T dv$ since dot products commute.
2. $d(uv^T) = (du)v^T + u(dv)^T$.

Remark 5. *The way to prove these sorts of statements can be seen in Section 2.*

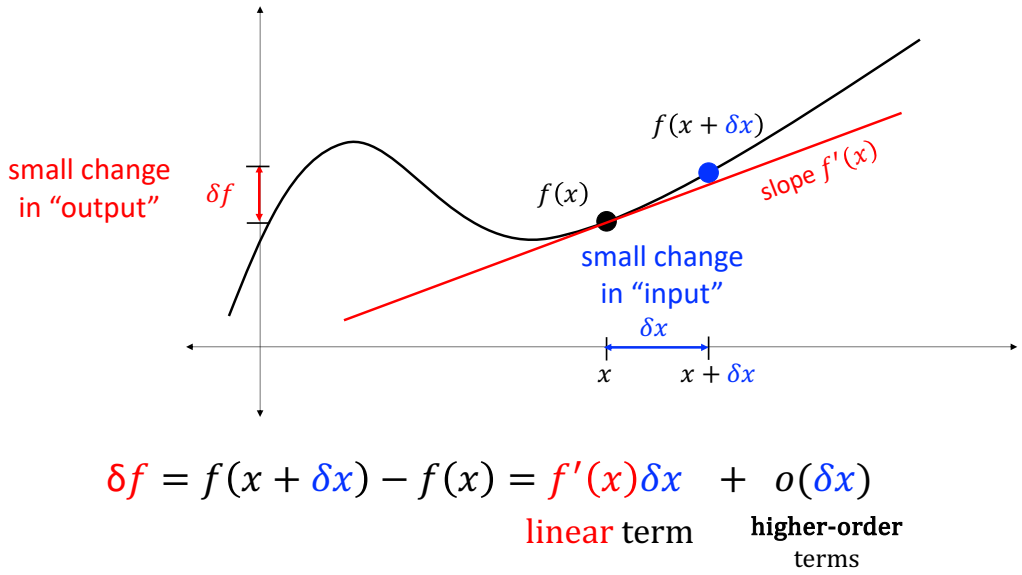


Figure 1: The essence of a derivative is *linearization*: predicting a small change δf in the output $f(x)$ from a small change δx in the input x, to *first order* in δx.

2 Derivatives as Linear Operators

We are now going to revisit the notion of a derivative in a way that we can generalize to higher-order arrays and other vector spaces. We will get into more detail on differentiation as a linear operator, and in particular, will dive deeper into some of the facts we have stated thus far.

2.1 Revisiting single-variable calculus

In a first-semester single-variable calculus course (like 18.01 at MIT), the derivative $f'(x)$ is introduced as the slope of the tangent line at the point $(x, f(x))$, which can also be viewed as a *linear approximation* of f near x. In particular, as depicted in Fig. 1, this is equivalent to a prediction of the *change δf in the "output"* of $f(x)$ from a small *change δx in the "input"* to first order (*linear*) in δx:

$$\delta f = f(x+\delta x) \quad f(x) = f'(x)\,\delta x + \underbrace{(\text{higher-order terms})}_{o(\delta x)}.$$

We can more precisely express these higher-order terms using asymptotic "little-o" notation "$o(\delta x)$", which denotes any function whose magnitude shrinks much faster than $j\delta xj$ as δx ! 0, so that for sufficiently small δx it is negligible compared to the linear $f'(x)\,\delta x$ term. (Variants of this notation are commonly used in computer science, and there is a formal definition that we omit here.[1]) Examples of such higher-order terms include $(\delta x)^2$, $(\delta x)^3$, $(\delta x)^{1.001}$, and $(\delta x)/\log(\delta x)$.

Remark 6. *Here, δx is not an infinitesimal but rather a small number.*

This notion of a derivative may remind you of the first two terms in a Taylor series $f(x+\delta x) = f(x)+f'(x)\,\delta x+$ (though in fact it is much more basic than Taylor series!), and the notation will generalize nicely to higher dimensions

[1] Briefly, a function $g(\delta x)$ is $o(\delta x)$ if $\lim_{\delta x\to 0} \frac{\|g(\delta x)\|}{\|\delta x\|} = 0$. We will return to this subject in Section 7.2.

and other vector spaces. In differential notation, we can express the same idea as:

$$df = f(x + dx) \qquad f(x) = f'(x)\,dx,$$

where in this notation we implicitly drop the $o(\delta x)$ term that vanishes in the limit as δx becomes infinitesimally small.

We will use this as the more generalized definition of a derivative. In this formulation, we avoid *dividing* by dx, because soon we will allow x (and hence dx) to be something other than a number—if dx is a vector, we won't be *able* to divide by it!

2.2 Linear operators

From the perspective of linear algebra, given a function f, we consider the differential of f to be the *linear operator* such that

$$df = f(x + dx) \qquad f(x) = f'(x)[dx].$$

As above, you should think of the differential notation dx as representing an *arbitrary small* change in x, where we are implicitly dropping any $o(dx)$ terms, i.e. terms that decay faster than linearly as dx ! 0. Often, we will omit the square brackets and write simply $f'(x)dx$ instead of $f'(x)[dx]$, but this should be understood as the linear operator $f'(x)$ *acting on* dx—don't write $dx\,f'(x)$, which will generally be nonsense!

This definition will allow us to extend differentiation to *arbitrary vector spaces* of inputs x and outputs $f(x)$. (More technically, we will require vector spaces with a norm *kxk*, called "Banach spaces," in order to precisely define the $o(\delta x)$ terms that are dropped. We will come back to the subject of vector norms and Banach spaces later.)

> **Recall 7 (Linear Operator)**
> Recall that a linear operator is a map L from a vector v in vector space V to a vector $L[v]$ (sometimes denoted simply Lv) in some other vector space. Specifically, L is linear if
>
> $$L[v_1 + v_2] = Lv_1 + Lv_2 \quad \text{and} \quad L[\alpha v] = \alpha L[v]$$
>
> for scalars α *2* R.

Some examples of linear operators include

Multiplication by scalars α, i.e. $Lv = \alpha v$. Also multiplication of column vectors v by matrices A, i.e. $Lv = Av$.

Some functions like $f(x) = x^2$ are obviously nonlinear. But what about $f(x) = x + 1$? This may *look* linear if you plot it, but it is *not* a linear operation, because $f(2x) = 2x + 1$ *6=* $2f(x)$—such functions, which are linear *plus a nonzero constant*, are known as *a ne*.

There are also many other examples of linear operations that are not so convenient or easy to write down as matrix–vector products. For example, if A is a 3 3 matrix, then $L[A] = AB + CA$ is a linear operator given 3 3 matrices B, C. The transpose $f(x) = x^T$ of a column vector x is linear, but is not given by any matrix multiplied by x. Or, if we consider vector spaces of *functions*, then the calculus operations of differentiation and integration are linear operators too!

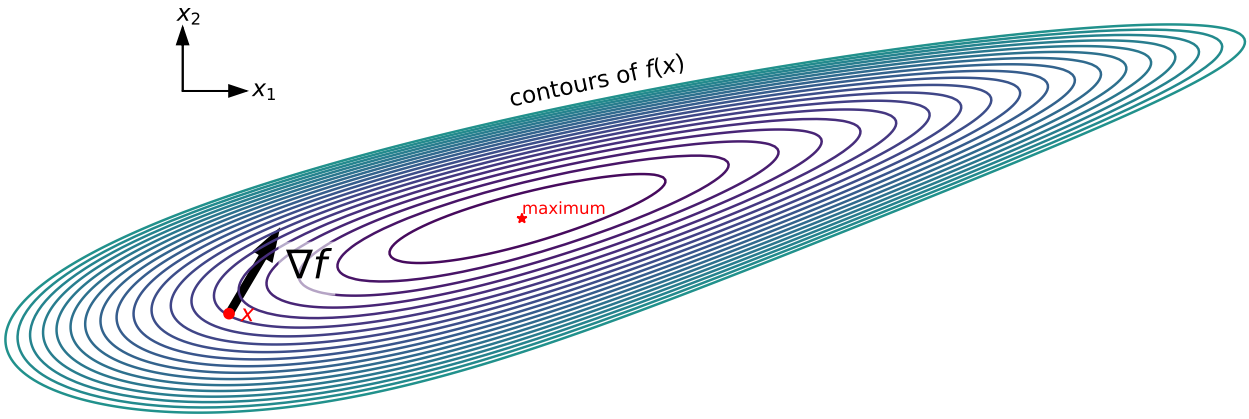


Figure 2: For a real-valued $f(x)$, the gradient ∇f is defined so that it corresponds to the "uphill" direction at a point x, which is perpendicular to the contours of f. Although this may not point exactly towards the nearest local maximum of f (unless the contours are circular), "going uphill" is nevertheless the starting point for many computational-optimization algorithms to search for a maximum.

2.3 Revisiting multivariable calculus, Part 1: Scalar-valued functions

Let f be a scalar-valued function, which takes in "column" vectors $x \in \mathbb{R}^n$ and produces a scalar (in $\mathbb{R}$). Then,

$$df = f(x + dx) \quad f(x) = f'(x)dx = \text{scalar}.$$

Therefore, since dx is a column vector (in an arbitrary direction, representing an arbitrary small change in x), the linear operator $f'(x)$ that produces a scalar df must be a **row vector** (a "1-row matrix", or more formally something called a *covector* or "dual" vector or "linear form")! We call this row vector the *transpose of the gradient* $(\nabla f)^T$, so that df is the *dot product of* dx *with the gradient*. So we have that

$$df = \nabla f \quad dx = \underbrace{(\nabla f)^T}_{f'(x)} dx.$$

It should be noted that there are a few conventions in the literature for this sort of thing, but treating the gradient as a "column vector" is a common choice because it can be viewed as the "uphill" (*steepest-ascent*) direction in the x space, as depicted in Fig. 2. In this class, we will always define ∇f to *have the same "shape" as* x, so that df is a dot product ("inner product") of dx with the gradient.

This is perfectly consistent with the viewpoint of the gradient that you may remember from multivariable calculus, in which the gradient was a vector of components

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix};$$

or, equivalently,

$$df = f(x + dx) \quad f(x) = \nabla f \quad dx = \frac{\partial f}{\partial x_1} dx_1 + \frac{\partial f}{\partial x_2} dx_2 + \quad + \frac{\partial f}{\partial x_n} dx_n \,.$$

While a component-wise viewpoint may sometimes be convenient, we want you to encourage you to view the vector x as a *whole*, not simply a collection of components, and to learn that it is often more convenient and elegant to differentiate expressions *without* taking the derivative component-by-component, learning a new approach that will generalize better to more complicated inputs/output vector spaces.

Let's look at an example to see how we compute this differential.

Example 8

Consider $f(x) = x^T A x$ where $x \in \mathbb{R}^n$ and A is a square $n \times n$ matrix, and thus $f(x) \in \mathbb{R}$. Compute df, $f'(x)$, and ∇f.

We can do this directly from the definition.

$$\begin{aligned} df &= f(x+dx) - f(x) \\ &= (x+dx)^T A(x+dx) - x^T A x \\ &= x^T A x + dx^T A x + x^T A dx + \underbrace{dx^T A dx}_{\text{higher order}} - x^T A x \\ &= \underbrace{x^T (A + A^T)}_{f'(x) = (\nabla f)^T} dx \implies \nabla f = (A + A^T) x \,. \end{aligned}$$

Here, we dropped terms with more than one dx factor as these are asymptotically negligible. Another trick was to combine $dx^T A x$ and $x^T A dx$ by realizing that these are *scalars* and hence equal to their own transpose: $dx^T A x = (dx^T A x)^T = x^T A^T dx$. Hence, we have found that $f'(x) = x^T (A + A^T) = (\nabla f)^T$, or equivalently $\nabla f = [x^T (A + A^T)]^T = (A + A^T) x$.

It is, of course, also possible to compute the same gradient component-by-component, the way you probably learned to do in multivariable calculus. First, you would need to write $f(x)$ explicitly in terms of the components of x, as $f(x) = x^T A x = \sum_{i,j} x_i A_{i,j} x_j$. Then, you would compute $\partial f / \partial x_k$ for each k, taking care that x appears twice in the f summation. However, this approach is awkward, error-prone, labor-intensive, and quickly becomes worse as we move on to more complicated functions. It is much better, we feel, to get used to treating vectors and matrices *as a whole*, not as mere collections of numbers.

2.4 Revisiting multivariable calculus, Part 2: Vector-valued functions

Next time, we will revisit multi-variable calculus (18.02 at MIT) again in a Part 2, where now f will be a vector-valued function, taking in vectors $x \in \mathbb{R}^n$ and giving vector outputs $f(x) \in \mathbb{R}^m$. Then, df will be a m-component column vector, dx will be an n-component column vector, and we must get a linear operator $f'(x)$ satisfying

$$\underbrace{df}_{m \text{ components}} = \underbrace{f'(x)}_{m \times n} \underbrace{dx}_{n \text{ components}} ,$$

so $f'(x)$ must be an $m \times n$ *matrix* called the *Jacobian* of f! This will be discussed in the next lecture.

3 Derivatives as Linear Operators, ctd.

3.1 Revisiting 18.02, Part 2, ctd.

Professor Johnson picks up where he left off last time: with the notion of a Jacobian and some computational examples.

The Jacobian matrix J represents the linear operator that takes dx to df:

$$df = J dx\,.$$

The matrix J has entries $J_{ij} = \frac{\partial f_i}{\partial x_j}$ (corresponding to the i-th row and the j-th column of J).

So now, suppose that $f : \mathsf{R}^2 \;!\; \mathsf{R}^2$. Let's understand how we would compute the differential of f:

$$df = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{bmatrix} \begin{bmatrix} dx_1 \\ dx_2 \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} dx_1 + \frac{\partial f_1}{\partial x_2} dx_2 \\ \frac{\partial f_2}{\partial x_1} dx_1 + \frac{\partial f_2}{\partial x_2} dx_2 \end{bmatrix}.$$

Let's compute an example.

Example 9

Consider the function $f(x) = Ax$ where A is a constant $m \quad n$ matrix. Then, by applying the distributive law for matrix–vector products, we have

$$\begin{aligned} df = f(x + dx) \quad f(x) &= A(x + dx) \quad Ax \\ = Ax + Adx \quad Ax &= Adx = f^{0}(x)dx. \end{aligned}$$

Therefore, $f^{0}(x) = A$.

Notice then that the linear operator A is its own Jacobian matrix!

Let's now consider some derivative rules.

Sum Rule: Given $f(x) = g(x) + h(x)$, we get that

$$df = dg + dh \;\; =) \;\; f^{0}(x)dx = g^{0}(x)dx + h^{0}(x)dx.$$

Hence, $f^{0} = g^{0} + h^{0}$ as we should expect.

Product Rule: Suppose $f(x) = g(x)h(x)$. Then,

$$\begin{aligned} df = f(x + dx) \quad f(x) & \\ &= g(x + dx)h(x + dx) \quad g(x)h(x) \\ &= (g(x) + \underbrace{g^{0}(x)dx}_{dg})(h(x) + \underbrace{h^{0}(x)dx}_{dh}) \quad g(x)h(x) \\ &= dg\, h + g\, dh\,, \end{aligned}$$

where the $dg\, dh$ term is higher-order and hence dropped in infinitesimal notation. Note, as usual, that dg and h may not commute now as they may no longer be scalars!

Let's look at some short examples of how we can apply the product rule nicely.

Example 10

Let $f(x) = Ax$. Then, $df = dA\overset{*}{x}^{0} + Adx = Adx \implies f'(x) = A$. We set $dA = 0$ here because A does not change when we change x.

Example 11

Let $f(x) = x^T Ax$. Then,

$$df = dx^T(Ax) + x^T d(Ax) = \underbrace{dx^T Ax}_{= x^T A^T dx} + x^T Adx = x^T(A + A^T)dx = (\nabla f)^T dx\,,$$

and hence $\nabla f = (A + A^T)x$ as before.

3.2 The Chain Rule

One of the most important rules from differential calculus is the chain rule, because it allows us to differentiate complicated functions built out of compositions of simpler functions. This chain rule can also be generalized to our differential notation in order to work for functions on arbitrary vector spaces:

Chain Rule: Let $f(x) = g(h(x))$. Then,

$$df = f'(x)[dx] = g'(h(x))(h'(x)[dx]).$$

In other words, $f'(x) = g'(h(x))h'(x)$: the Jacobian (linear operator) f' is simply the *product (composition) of the Jacobians*, $g'h'$. Ordering matters because linear operators do not generally commute: left-to-right = outputs-to-inputs.

Let's look more carefully at the *shapes* of these Jacobian matrices in an example where each function maps a column vector to a column vector:

Example 12

Let $x \in \mathbb{R}^n$, $h(x) \in \mathbb{R}^p$, and $g(h(x)) \in \mathbb{R}^m$. Then, let $f(x) = g(h(x))$ mapping from $\mathbb{R}^n$ to $\mathbb{R}^m$. The chain rule then states that

$$f'(x) = g'(h(x))h'(x),$$

which makes sense as g' is an $m \times p$ matrix and h' is a $p \times n$ matrix, so that the product gives an $m \times n$ matrix f'! However, notice that this is *not* the same as $h'(x)g'(h(x))$ as you cannot (if $n \neq m$) multiply a $p \times n$ and an $m \times p$ matrix together, and even if $n = m$ you will get the wrong answer since they probably won't commute.

Not only does the order of the multiplication matter, but the associativity of matrix multiplication matters *practically*. Let's consider a function

$$f(x) = a(b(c(x)))$$

where $c : \mathbb{R}^n \to \mathbb{R}^p$, $b : \mathbb{R}^p \to \mathbb{R}^q$, and $a : \mathbb{R}^q \to \mathbb{R}^m$. Then, we have that, by the chain rule,

$$f'(x) = a'(b(c(x)))b'(c(x))c'(x).$$

Notice that this is the same as

$$f' = (a'b')c' = a'(b'c')$$

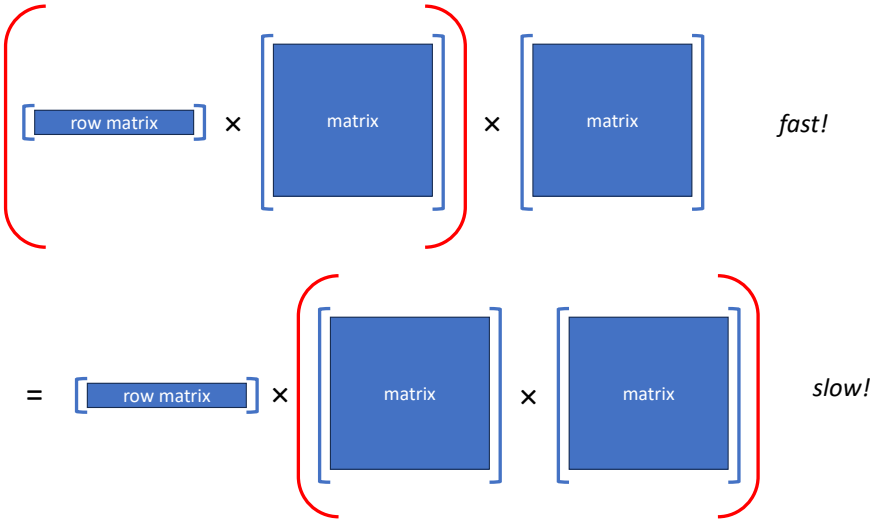


Figure 3: Matrix multiplication is *associative*—that is, $(AB)C = A(BC)$ for all A, B, C—but multiplying left-to-right can be much more efficient than right-to-left if the leftmost matrix has only one (or few) rows, as shown here. Correspondingly, the order in which you carry out the chain rule has dramatic consequences for the computational effort required. Left-to-right is known as "reverse mode" or "backpropagation", and is best suited to situations where there are many fewer outputs than inputs.

by associativity (omitting the function arguments for brevity). The left-hand side is multiplication from left to right, and the right-hand side is multiplication from right to left.

But who cares? Well it turns out that associativity is deeply important. So important that the two orderings have names: multiplying left-to-right is called "reverse mode" and multiplying right-to-left is called "forward mode" in the field of *automatic di erentiation* (AD). Reverse-mode differentation is also known as an "adjoint method" or "backpropagation" in some contexts, which we will explore in more detail later. Why does this matter? Let's think about the computational cost of matrix multiplication.

3.2.1 Cost of Matrix Multiplication

If you multiply a m q matrix by a q p matrix, you normally do it by computing mp dot products of length q (or some equivalent re-ordering of these operations). To do a dot product of length q requires q multiplications and q 1 additions of scalars. Overall, this is approximately $2mpq$ scalar operations in total. In computer science, you would write that this is "$\Theta(mpq)$": the computational effort is *asymptotically proportional* to mpq for large m, p, q.

So why does the order of the chain rule matter? Consider the following two examples.

Example 13

Suppose you have a lot of inputs n 1, and only one output $m = 1$, with lots of intermediate values, i.e. $q = p = n$. Then reverse mode (left-to-right) will cost $\Theta(n^2)$ scalar operations while forward mode (right-to-left) would cost $\Theta(n^3)$! This is a huge cost difference, depicted schematically in Fig. 3.

Conversely, suppose you have a lot of outputs m 1 and only one input $n = 1$, with lots of intermediate values $q = p = m$. Then reverse mode would cost $\Theta(m^3)$ operations but forward mode would be only $\Theta(m^2)$!

Moral: If you have a lot of inputs and few outputs (the usual case in machine learning and optimization), compute the chain rule left-to-right (reverse mode). If you have a lot of outputs and few inputs, compute the chain rule right-to-left (forward mode).

3.3 Beyond 18.02 Derivatives

Now let's compute some derivatives that go beyond first-year calculus, where the inputs and outputs are in more general vector spaces. For instance, consider the following examples:

Example 14

Let A be an n n matrix. You could have the following matrix-valued functions. For example:

$f(A) = A^3$,

$f(A) = A^{-1}$ if A is invertible,

or U, where U is the resulting matrix after applying Gaussian elimination to A!

You could also have scalar outputs. For example:

$f(A) = \det A$,

$f(A) = \text{trace } A$,

or $f(A) = \sigma_1(A)$, the largest singular value of A.

Let's focus on two simpler examples for this lecture.

Example 15

Let $f(A) = A^3$ where A is a square matrix. Compute df.

Here, we apply the chain rule one step at a time:

$$df = dA\,A^2 + A\,dA\,A + A^2\,dA = f'(A)[dA].$$

Notice that this is not equal to $3A^2$ (unless dA and A commute, which won't generally be true since dA represents an *arbitrary* small change in A). The right-hand side is a linear operator $f'(A)$ acting on dA, but it is not so easy to interpret it as simply a single "Jacobian" matrix multiplying dA!

Example 16

Let $f(A) = A^{-1}$ where A is a square invertible matrix. Compute $df = d(A^{-1})$.

Here, we use a slight trick. Notice that $AA^{-1} = \mathrm{I}$, the identity matrix. Thus, we can compute the differential using the product rule (noting that $d\,\mathrm{I} = 0$, since changing A does not change I) so

$$d(AA^{-1}) = dA\,A^{-1} + A\,d(A^{-1}) = d(\mathrm{I}) = 0 \implies d(A^{-1}) = -A^{-1}\,dA\,A^{-1}.$$

4 Two-by-two Matrix Jacobians

For this section of the notes, we refer to the following Pluto Notebook for a demonstration of the material we are discussing.

In this notebook we emphasize the multiple views of Jacobians using 2x2 matrix functions as examples. In particular, we discuss the Jacobian using a linear transformation viewpoint and what is known as the Kronecker product notation.

4.1 The Matrix Square Function

Suppose we have the following matrix

$$X = \begin{bmatrix} p & r \\ q & s \end{bmatrix},$$

and consider the function $f(X) = X^2$. We can think of this as a function from four variables to one with four variables, i.e. $\mathbb{R}^4 \to \mathbb{R}^4$, but what we want to start doing is viewing this as a function from 2×2 matrices to 2×2 matrices, i.e. $\mathbb{R}^{2,2} \to \mathbb{R}^{2,2}$. To do so, consider the following question:

Question 17. *What is the size of the Jacobian of $f(X)$?*

Well, it would be a 4×4 matrix– which one can obtain by making a matrix formed by the derivatives of each coordinate with respect to each variable. Now let's think about the general square matrix– i.e. an $n \times n$ matrix. If we want to find the Jacobian of f, we can do so by the same process and (symbolically) receive an $n^2 \times n^2$ matrix. This symbolic computation is one that Julia and Mathematica can do. In fact, as seen in the Notebook, Julia spits out the Jacobian quite easily:

$$J_{f(X)}(X) = \begin{bmatrix} 2p & r & q & 0 \\ q & p+s & 0 & q \\ r & 0 & p+s & r \\ 0 & r & q & 2s \end{bmatrix}.$$

However, we urge you not to do this too instinctively– rather, we want to think about the Jacobian as a linear transformation.

4.2 The Jacobian as a Linear Transformation

As we have already discussed, by the product rule we have that

$$d(X^2) = X\,dX + dX\,X\,.$$

This is, in fact, a linear transformation of dX. If the differential is distracting you, realize that for an arbitrary square matrix M (of the same size as X), the following is a linear operation on M:

$$M \mapsto XM + MX.$$

This is a perfectly good way to express a linear operation. However, if you want to express it in "matrix-times-vector" operation, you can use a technique involving the ***Kronecker product*** (denoted).

Consider the two matrices

$$A = \begin{bmatrix} a & c \\ b & d \end{bmatrix} \quad \text{and} \quad P = \begin{bmatrix} p & r \\ q & s \end{bmatrix}.$$

Then $A \quad P$ is given by taking all possible products of the entries in the first matrix with the entries of the second. Symbolically, we would obtain

$$A \quad P = \begin{bmatrix} aP & cP \\ bP & dP \end{bmatrix} = \begin{bmatrix} ap & ar & cp & cr \\ aq & as & cq & cs \\ bp & br & dp & dr \\ bq & bs & dq & ds \end{bmatrix}.$$

Similarly,

$$P \quad A = \begin{bmatrix} pA & rA \\ qA & sA \end{bmatrix} = \begin{bmatrix} pa & pc & ra & rc \\ pb & pd & rb & rd \\ qa & qc & sa & sc \\ qb & qd & sb & sd \end{bmatrix}.$$

See the Notebook for more examples of Kronecker products of matrices (including some with pictures rather than numbers!). Using this, we will find that we can represent the Jacobian of $f(X) = X^2$ at $X = \begin{bmatrix} p & r \\ q & s \end{bmatrix}$ as an ***equivalent*** operation on a 'vectorized" column vector consisting of the ' entries of X via a Kronecker product:

$$\mathrm{I}_2 \quad X + X^T \quad \mathrm{I}_2 \quad ! \underbrace{\begin{bmatrix} 2p & r & q & 0 \\ q & p+s & 0 & q \\ r & 0 & p+s & r \\ 0 & r & q & 2s \end{bmatrix}}_{J_{f(X)}} \underbrace{\begin{pmatrix} p \\ q \\ r \\ s \end{pmatrix}}_{\mathrm{vec}(X)}.$$

We will pick back up here on Monday.

5 Two-by-two Matrix Jacobians, ctd.

Consider the function $f(x) = \|x\|_2 := \sqrt{x_1^2 + \quad + x_n^2}$ from $\mathbb{R}^n$ to $\mathbb{R}$.

Question 18. *How do we calculate $\nabla_x f$?*

The 18.02 way to solve this question is to do it one component at a time, but they are all realistically the same for each variable. We have:

$$\partial_{x_1} \sqrt{x_1^2 + \quad + x_n^2} = \frac{2x_1}{2\sqrt{x_1^2 + \ldots x_n^2}} = \frac{x_1}{\sqrt{x_1^2 + \ldots x_n^2}}.$$

Doing this for each variable would lead to

$$\nabla_x f = \frac{x}{\|x\|_2}.$$

But in the 18.S096 way, we want to compute these derivatives without indices! So consider $r = \|x\|_2 \implies r^2 = x^T x$. Therefore, we have

$$2r\,dr = 2x^T\,dx \implies dr = \frac{x^T}{r}\,dx.$$

This implies that $\nabla_x f = \frac{x}{r}$ which is the same desired result, but without the indices! Now this example is a scalar function of a vector, but we are really interested in functions which take in and produce matrices. So let's go back to our Julia notebook from last time.

5.1 Key Kronecker-Product Identity

Remark 19. *Note that we started going through this notebook last time, so the material will not be repeated here.*

So last time we picked up, we noticed that

$$J_{f(X)=X^2}(X) = \mathrm{I}_2 \quad X + X^T \quad \mathrm{I}_2\,.$$

The reason that this is true follows from a nice identity.

Definition 20

Given a matrix A, we define its "vectorization" $\mathrm{vec}(A)$ to be the column vector whose entries are obtained by vertically "stacking" the columns of A, taken from left to right.

For example,

$$\mathrm{vec}\left(\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}\right) = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}.$$

Then,

Proposition 21

Given A, B, C (compatible) matrices, we have

$$(A \quad B)\,\mathrm{vec}(C) = \mathrm{vec}(BCA^T).$$

This will be particularly important when it comes to calculating the Jacobian in terms of the Kronecker product. Before moving onto that, however, we note the following useful Kronecker product identities for your use:

Proposition 22

We have the following:

1. $(A \quad B)^T = A^T \quad B^T$,
2. $(A \quad B)^{-1} = A^{-1} \quad B^{-1}$,
3. $\det(A \quad B) = \det(A)^m \det(B)^n$, where $A \in \mathbb{R}^{n,n}$ and $B \in \mathbb{R}^{m,m}$,
4. $A \quad B$ is orthogonal if A and B are orthogonal,
5. $(A \quad B)(C \quad D) = (AC) \quad (BD)$,
6. and if $Au = \lambda u$ and $Bv = \mu v$, then if $X = vu^T$ then $BXA^T = \lambda\mu X$.

5.2 The Jacobian in Kronecker-Product Notation

So now we want to use Proposition 21 to calculate the Jacobian of $f(X) = X^2$ in terms of the Kronecker product. Let dX be our C in Proposition 21. Then, we can notice (by our definition of the Jacobian and by the product rule) that

$$\operatorname{vec}(X\,dX + dX\,X) = (\mathrm{I}_2 \quad X + X^T \quad \mathrm{I}_2)\operatorname{vec}(dX) = J_{f(X)}(X)\operatorname{vec}(dX),$$

which is precisely the formula we had obtained! In fact, we can use this to compute the Jacobian of functions from matrices to matrices quite easily.

Let's do the same for the matrix cube function. Sure, we could symbolically check what the Jacobian is (which is done nicely in the notebook), but let's show a similar identity as before. Notice that we have

$$\operatorname{vec}(dX\,X^2 + X\,dX\,X + X^2\,dX) = \left(\mathrm{I}_2 \quad X^2 + X^T \quad X + (X^2)^T \quad \mathrm{I}_2\right)\vec{(}dX) = J_{f(X)=X^3}(X)\operatorname{vec}(dX).$$

Remark 23. *You can do a similar example relating the Jacobian from LU decomposition in terms of the Kronecker product, which is an exercise on your problem set.*

Notice that we could use Automatic Differentiation to obtain the Jacobian of matrices, as can be seen in the Pluto notebook as well. However, the way automatic differentiation works *is not* via symbolic calculation, nor is it by calculating finite differences (like in the difference quotient). What precisely automatic differentiation is doing will be talked about later on in the course. For now though, Professor Johnson will discuss finite differences, which comes up a lot in computing derivatives on a computers.

6 Finite-Difference Approximations

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

6.1 Hand-calculated Derivative Rules: Error Prone

Rather than calculating derivatives by hand, we *could* ideally use **automatic differentiation** (AD).

AD lets software/compilers perform the 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 "needs help"—for example, if you are computing an answer *approximately* (e.g. solving a nonlinear equation by Newton's method), AD can waste a lot of effort trying to *exactly differentiate the error* in your approximation. Often, you can compute the approximate answer (to the same accuracy) much more efficiently. Though even in cases where AD falls down, often you only need to give it a **little help**: define a differentiation rule for a *small piece* of your program and let AD handle the rest. In Julia, 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).

An important fallback algorithm, when AD fails, 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. Even if you have implemented an exact analytical derivative, finite-difference approximations are extremely *useful as a check* to make sure that you have done it correctly.

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.

6.2 Finite-Difference Approximations: Easy Version

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

$$df = f(x+dx) \quad f(x) = f'(x)dx$$

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

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

So, we can just compare the **finite difference** $\boxed{f(x+\delta x) \quad f(x)}$ to our **(directional) derivative operator** $f'(x)\delta x$ (i.e. the derivative in the direction δx). $f(x+\delta x) \quad f(x)$ is also called a **forward difference** approximation. The antonym of a forward difference is a **backward difference** approximation $f(x) \quad f(x \quad \delta x) \quad f'(x)\delta x$. If you just want to compute a derivative, there is not much practical distinction between forward and backward differences. The distinction becomes more important when discretizing (approximating) differential equations. We'll look at other possibilities below.

Remark 24. *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:

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

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**.

6.3 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 $\mathsf{R}^{m,m}$. 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 $dA = A \quad 10^{\ 8}$. Then, you can compare $f(A + dA) \quad f(A)$ 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 entreis of order $10^{\ 8}$.

To be more quantitative, we might compute that "norm" $k$approx $\quad$ exact$k$ 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

$$\text{relative error} = \frac{k\text{approx} \quad \text{exact}k}{k\text{exact}k}.$$

Here, $k\,k$ is a norm, like the length of a vector.

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 alwyas 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.

Definition 25

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:

$$kAk := \sum_{i,j} jA_{ij}j^2 = \sqrt{\mathrm{tr}(A^T A)}\,.$$

This is called the Frobenius norm.

6.4 Accuracy of Finite Di□erences

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 $k\delta Ak$. This plot will be done *logarithmically* (on a log–log scale) so that we can see power-law relationships as straight lines.

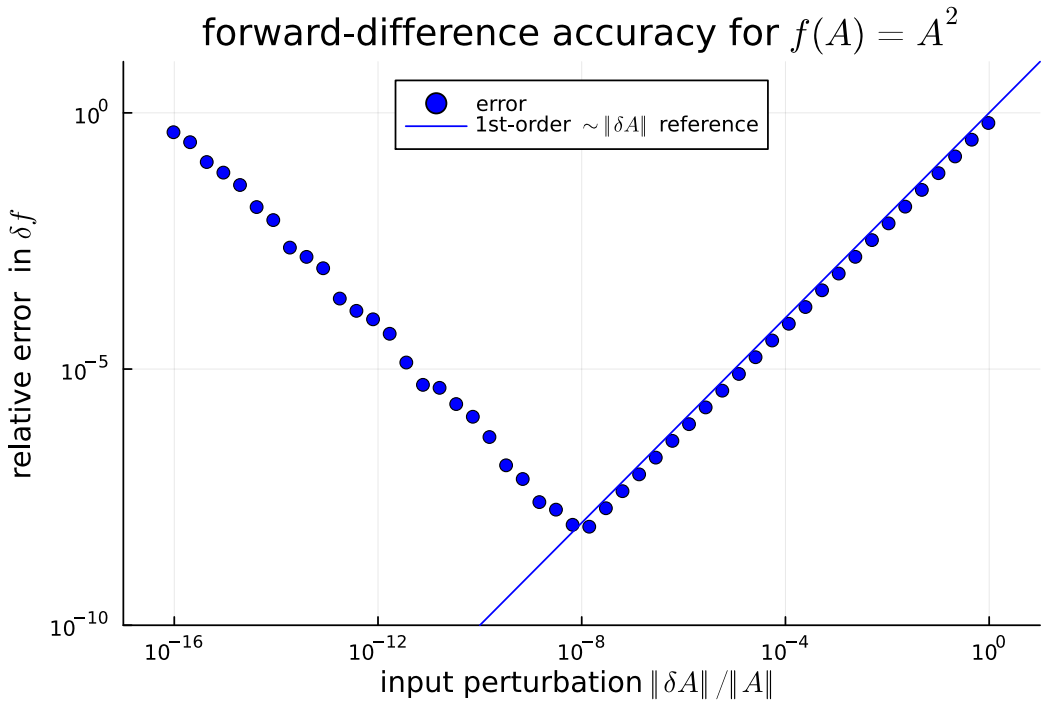


Figure 4: Forward-difference accuracy for $f(A) = A^2$, showing the relative error in $\delta f = f(A + \delta A) \quad f(A)$ versus the linearization $f^0(A)\delta A$, as a function of the magnitude $k\delta Ak$. A is a 4 4 matrix with unit-variance Gaussian random entries, and δA is similarly a unit-variance Gaussian random perturbation scaled by a factor s ranging from 1 to 10 ^{16}.

We notice two main features as we decrease δA:

1. The relative error at first decreases linearly with $k\delta Ak$. This is called **first-order accuracy**. Why?
2. When δA gets too small, the error increases. Why?

6.5 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 $k\delta xk^n$, 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

$$f(x + \delta x) = f(x) + f^0(x)\delta x + (\text{terms proportional to } k\delta xk^2) + \underbrace{o(k\delta xk^2)}_{\text{i.e. higher-order terms}}$$

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

$$\begin{aligned}\text{relative error} &= \frac{kf(x + \delta x) \quad f(x) \quad f^0(x)\delta xk}{kf^0(x)\delta xk} \\ &= \frac{(\text{terms proportional to } k\delta xk^2) + o(k\delta xk^2)}{\text{proportional to } k\delta xk} = (\text{terms proportional to } k\delta xk) + o(k\delta xk)\end{aligned}$$

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?

6.6 Roundo 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 . 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 $\log_{10} \epsilon$ decimal digits, or really $53 = 1$ $\log_2 \epsilon$ *binary* digits, for each number.

In our finite-difference example, for $\|\delta A\|/\|A\| \approx 10^{-8} \sqrt{\epsilon}\|x\|$, 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. In general, $\|\delta x\| \approx \sqrt{\epsilon}\|x\|$ is a good rule of thumb, but the precise crossover point of minimum error depends on the function f and the finite-difference method.

6.7 Other finite-di erence 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'. 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.

7 Derivatives in General Vector Spaces

Matrix calculus requires us to generalize concepts of derivative and gradient further, to functions whose inputs and/or outputs are not simply scalars or column vectors. To achieve this, we extend the notion of the ordinary vector **dot product** and ordinary Euclidean vector length to general **inner products** and **norms** on **vector spaces**. Our first example will consider familiar matrices from this point of view.

Recall from linear algebra that we can call any set V a "vector space" if its elements can be added/subtracted $x \quad y$ and multiplied by scalars αx (subject to some basic arithmetic axioms, e.g. the distributive law). For example, the set of $m \quad n$ matrices themselves form a vector space, or even the set of continuous functions $u(x)$ (mapping $\mathsf{R} \; ! \; \mathsf{R}$)—the key fact is that we can add/subtract/scale them and get elements of the same set. It turns out to be extraordinarily useful to extend differentiation to such spaces, e.g. for functions that map matrices to matrices or functions to numbers. Doing so crucially relies on our input/output vector spaces V having a **norm** and, ideally, an **inner product**.

7.1 A Simple Matrix Dot Product and Norm

Recall that for *scalar-valued* functions $f(x) \; 2 \; \mathsf{R}$ with *vector inputs* $x \; 2 \; \mathsf{R}^n$ (i.e. n-component "column vectors") we have that

$$df = f(x + dx) \quad f(x) = f^0(x)[dx] \; 2 \; \mathsf{R}.$$

Therefore, $f^0(x)$ is a linear operator taking in the vector dx in and giving a scalar value out. Another way to view this, is that $f^0(x)$ is the row vector (also called a "covector") $(\mathsf{r} \, f)^T$. Under this viewpoint, it follows that df is the dot product:

$$df = \mathsf{r} \, f \quad dx$$

We can generalize this to any vector space V with inner products! Given $x \; 2 \; V$, and a scalar-valued function f, we obtain the linear operator $f^0(x)[dx] \; 2 \; \mathsf{R}$, called a "linear form." In order to define the gradient $\mathsf{r} \, f$, we need an inner product for V, the vector-space generalization of the familiar dot product!

Given $x, y \; 2 \; V$, the inner product $\quad$ is a map $(\;)$ such that $x \quad y \; 2 \; \mathsf{R}$. This is also commonly denoted $\hbar x, y \imath$ or $\hbar x \, \jmath \, y \imath$. More technically, an inner product is a map $(\;)$ that is

1. **Symmetric**: i.e. $x \quad y = y \quad x$,
2. **Linear**: i.e. $x \quad (\alpha y + \beta z) = \alpha(x \quad y) + \beta(x \quad z)$, and
3. **Non-negative**: i.e. $x \quad x := \|x\|^2 \quad 0$, and $= 0$ if and only if $x = 0$.

Definition 26 (Hilbert Space)

A (complete) vector space with an inner product is called a *Hilbert space*. (The technical requirement of "completeness" essentially means that you can take limits in the space, and is important for rigorous proofs.[a])

[a] Completeness means that any Cauchy sequence of points in the vector space—any sequence of points that gets closer and closer together—has a limit lying within the vector space. This criterion usually holds in practice for vector spaces over real or complex scalars, but can get trickier when talking about vector spaces of functions, since e.g. the limit of a sequence of continuous functions can be a discontinuous function.

Once we have a Hilbert space, we can define the gradient for scalar-valued functions. Given $x \; 2 \; V$ a Hilbert space, and $f(x)$ scalar, then we have the linear form $f^0(x)[dx] \; 2 \; \mathsf{R}$. Then, under these assumptions, there is a theorem known as the "Riesz representation theorem" stating that *any* linear form (including f^0) must be an inner

product with *something*:

$$f'(x)[dx] = \underbrace{(\text{some vector})}_{\text{gradient } \nabla f\big|_x} dx = df.$$

That is, the gradient ∇f is *defined* as the thing you take the inner product of dx with to get df. Note that ∇f always has the "same shape" as x.

The first few examples we look at involve the usual Hilbert space $V = \mathbb{R}^n$ with different inner products.

Example 27

Given $V = \mathbb{R}^n$ with n-column vectors, we have the familiar Euclidean dot product $x \cdot y = x^T y$. This leads to the usual ∇f.

Example 28

We can have different inner products on $\mathbb{R}^n$. For instance,

$$x \cdot_W y = w_1 x_1 y_1 + w_2 x_2 y_2 + \ldots w_n x_n y_n = x^T \underbrace{\begin{bmatrix} w_1 & & \\ & \ddots & \\ & & w_n \end{bmatrix}}_{W} y$$

for weights $w_1, \ldots, w_n > 0$.

More generally we can define a weighted dot product $x \cdot_W y = x^T W y$ for any symmetric-positive-definite matrix W ($W = W^T$ and W is positive definite). Note that we need these requirements on W in order for the inner product to satisfy the symmetric and non-negative properties.

If we change the definition of the inner product, then we change the definition of the gradient! For example, with $f(x) = x^T A x$ we previously found that $df = x^T(A + A^T)dx$. With the ordinary Euclidean inner product, this gave a gradient $\nabla f = (A + A^T)x$. However, if we use the weighted inner product $x^T W y$, then we would obtain a different "gradient" $\nabla^{(W)} f = W^{-1}(A + A^T)x$ so that $df = (\nabla^{(W)} f) \cdot_W dx$.

In these notes, we will employ the Euclidean inner product for $x \in \mathbb{R}^n$, and hence the usual ∇f, unless noted otherwise. However, weighted inner products are useful in lots of cases, especially when the components of x have different scales/units.

We can also consider the space of $m \times n$ matrices $V = \mathbb{R}^{m \times n}$. There, is of course, a vector-space isomorphism from $V \ni A \mapsto \text{vec}(A) \in \mathbb{R}^{mn}$. Thus, in this space we have the analogue of the familiar ("Frobenius") Euclidean inner product, which is convenient to rewrite in terms of matrix operations via the trace:

$$A : B = \sum_{ij} A_{ij} B_{ij} = \text{vec}(A)^T \text{vec}(B) = \text{tr}(A^T B).$$

Given this inner product, we have the Frobenius norm induced

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

. Using this, we can now take the gradient of scalar functions with *matrix inputs*! (This will be our default matrix inner product, and hence our default matrix gradient, in these notes.)

Example 29

Consider the function

$$f(A) = \|A\|_F = \sqrt{\mathrm{tr}(A^T A)}.$$

What is df?

Firstly, by the familiar scalar-differentiation chain and power rules we have that

$$df = \frac{1}{2\sqrt{\mathrm{tr}(A^T A)}} d(\mathrm{tr}\, A^T A).$$

Then, note that (by linearity of the trace)

$$d(\mathrm{tr}\, B) = \mathrm{tr}(B + dB) \quad \mathrm{tr}(B) = \mathrm{tr}(B) + \mathrm{tr}(dB) \quad \mathrm{tr}(B) = \mathrm{tr}(dB).$$

Hence,

$$\begin{aligned} df &= \frac{1}{2\|A\|_F} \mathrm{tr}(d(A^T A)) \\ &= \frac{1}{2\|A\|_F} \mathrm{tr}(dA^T\, A + A^T\, dA) \\ &= \frac{1}{2\|A\|_F} (\mathrm{tr}(dA^T\, A) + \mathrm{tr}(A^T\, dA)) \\ &= \frac{1}{\|A\|_F} \mathrm{tr}(A^T\, dA) = \frac{A}{\|A\|_F} \quad dA. \end{aligned}$$

Here, we used the fact that $\mathrm{tr}\, B = \mathrm{tr}\, B^T$, and in the last step we connected df with a Frobenius inner product. In other words,

$$\nabla f = \nabla \|A\|_F = \frac{A}{\|A\|_F}.$$

Let's consider another simple example:

Example 30

Fix some constant $x \in \mathbb{R}^m$, $y \in \mathbb{R}^n$, and consider the function $f : \mathbb{R}^{m \times n} \to \mathbb{R}$ given by

$$f(A) = x^T A y.$$

What is ∇f?

We have that

$$\begin{aligned} df &= x^T\, dA\, y \\ &= \mathrm{tr}(x^T\, dA\, y) \\ &= \mathrm{tr}(y x^T\, dA) \\ &= \underbrace{x y^T}_{\nabla f} \quad dA. \end{aligned}$$

More generally, for any scalar-valued function $f(A)$, from the definition of Frobenius inner product it follows that:

$$df = f(A + dA) \quad f(A) = \nabla f \quad dA = \sum_{i,j} (\nabla f)_{i,j}\, dA_{i,j}\,,$$

and hence the components of the gradient are exactly the elementwise derivatives

$$(\nabla f)_{i,j} = \frac{\partial f}{\partial A_{i,j}},$$

similar to the component-wise definition of the gradient vector from multivariable calculus! But for non-trivial matrix-input functions $f(A)$ it can be extremely awkward to take the derivative with respect to each entry of A individually. Using the "holistic" matrix inner-product definition, we will soon be able to compute even more complicated matrix-valued gradients, including $\nabla(\det A)$!

7.2 Derivatives, Norms, and Banach spaces

We have been using the term "norm" throughout this class, but what technically is a norm? Of course, there are familiar examples such as the Euclidean ("ℓ^2") norm $\|x\| = \sqrt{\sum_k x_k^2}$ for $x \in \mathbb{R}^n$, but it is useful to consider how this concept generalizes to other vector spaces. It turns out, in fact, that norms are crucial to the definition of a derivative!

Given a vector space V, a norm $\|\cdot\|$ on V is a map $\|\cdot\| : V \to \mathbb{R}$ satisfying the following three properties:

1. **Non-negative**: i.e. $\|v\| \geq 0$ and $\|v\| = 0 \iff v = 0$,
2. **Homogeneity**: $\|\alpha v\| = |\alpha| \|v\|$ for any $\alpha \in \mathbb{R}$, and
3. **Triangle inequality**: $\|u + v\| \leq \|u\| + \|v\|$.

An vector space that has a norm is called an *normed vector space*. Often, mathematicians technically want a slightly more precise type of normed vector space with a less obvious name: a *Banach* space.

Definition 31 (Banach Space)
A (complete) vector space with a norm is called a *Banach space*. (As with Hilbert spaces, "completeness" is a technical requirement for some types of rigorous analysis, essentially allowing you to take limits.)

For example, given any inner product $u \cdot v$, there is a corresponding norm $\|u\| = \sqrt{u \cdot u}$. (Thus, every Hilbert space is also a Banach space.[2])

To define derivatives, we technically both the input *and* the output to be Banach spaces. To see this, recall our formalism

$$f(x + \delta x) - f(x) = \underbrace{f'(x)[\delta x]}_{\text{linear}} + \underbrace{o(\delta x)}_{\text{smaller}}.$$

To precisely define the sense in which the $o(\delta x)$ terms are "smaller" or "higher-order," we need norms. In particular, the "little-o" notation $o(\delta x)$ denotes any function such that

$$\lim_{\delta x \to 0} \frac{\|o(\delta x)\|}{\|\delta x\|} = 0,$$

i.e. which goes to zero faster than linearly in δx. This requires both the input δx and the output (the function) to have norms. This extension of differentiation to arbitrary normed/Banach spaces is sometimes called the **Fréchet derivative**.

[2] Proving the triangle inequality for an arbitrary inner product is not so obvious; one uses a result called the Cauchy–Schwarz inequality.

8 Nonlinear Root-Finding, Optimization, and Adjoint Differentiation

The next part are based on these slides. Today, we want to talk about why we are computing derivatives in the first place. In particular, we will drill down on this a little bit and then talk about computation of derivatives.

8.1 Newton's Method

One common application of derivatives is to solve nonlinear equations via linearization. For instance, suppose we has a scalar function $f : \mathsf{R} \; ! \; \mathsf{R}$ and we wanted to solve $f(x) = 0$ for a root x. Of course, we could solve such an equation explicitly in simple cases, such as when f is linear or quadratic, but if the function is something more arbitrary like $f(x) = x^3 \quad \sin(\cos x)$ you might not be able to obtain closed-form solutions. However, there is a nice way to obtain the solution approximately to any accuracy you want, as long if you know approximately where the root is. The method we are talking about is known as *Newton's method*, which is really a linear-algebra technique. It takes in the function and a guess for the root, approximates it by a straight line (whose root is easy to find), which is then an approximate root that we can use as a new guess. In particular, the method (depicted in Fig. 5) is as follows:

> Linearize $f(x)$ near some x using the approximation
>
> $$f(x + \delta x) \quad f(x) + f^0(x)\delta x,$$
>
> solve the linear equation $f(x) + f^0(x)\delta x = 0 \; =) \; \delta x = \; \frac{f(x)}{f^0(x)}$,
>
> and then use this to update the value of x we linearized near—i.e., letting the new x be
>
> $$x_{\mathsf{new}} = x \quad \delta x = x \quad \frac{f(x)}{f^0(x)}\,.$$

Once you are close to the root, Newton's method converges amazingly quickly. As discussed below, it asympotically *doubles* the number of correct digits on every step!

8.1.1 Scalar Functions

8.1.2 Multidimensional Functions

We can generalize Newton's method to multidimensional functions! Let $f : \mathsf{R}^n \; ! \; \mathsf{R}^n$ be a function which takes in a vector and spits out a vector of the same size n. We can then apply a Newton approach in higher dimensions:

> Linearize $f(x)$ near some x using the first-derivative approximation
>
> $$f(x + \delta x) \quad f(x) + \underbrace{f^0(x)}_{\mathsf{Jacobian}} \delta x,$$
>
> solve the linear equation $f(x) + f^0(x)\delta x = 0 \; =) \; \delta x = \; \underbrace{f^0(x) \; ^1}_{\mathsf{inverse\ Jacobian}} \quad f(x)$,
>
> and then use this to update the value of x we linearized near—i.e., letting the new x be
>
> $$x_{\mathsf{new}} = x_{\mathsf{old}} \quad f^0(x) \; ^1 f(x)\,.$$

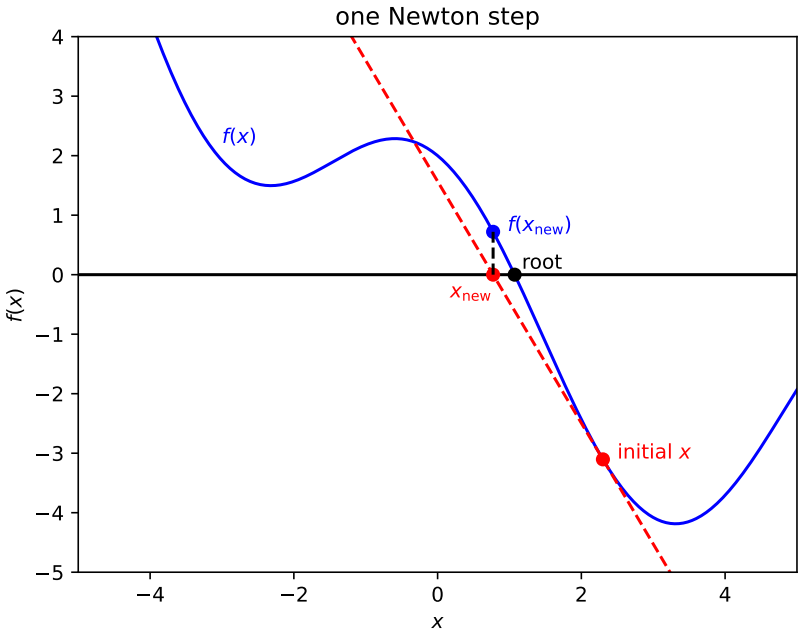


Figure 5: Single step of the scalar Newton's method to solve $f(x) = 0$ for an example nonlinear function $f(x) = 2\cos(x) \quad x + x^2/10$. Given a starting guess ($x = 2.3$ in this example), we use $f(x)$ and $f^0(x)$ to form a linear (affine) approximation of f, and then our next step x_{new} is the root of this approximation. As long as the initial guess is not too far from the root, Newton's method converges extremely rapidly to the exact root (black dot).

That's it! Once we have the Jacobian, we can just solve a linear system on each step. This again converges amazingly fast, doubling the number of digits of accuracy in each step. (This is known as "quadratic convergence.") However, there is a caveat: we *need* some starting guess for x, and the guess needs to be sufficiently close to the root for the algorithm to make reliable progress. (If you start with an initial x far from a root, Newton's method can fail to converge and/or it can jump around in intricate and surprising ways—google "Newton fractal" for some fascinating examples.) This is a widely used and very practical application of Jacobians and derivatives!

8.2 Optimization

8.2.1 Nonlinear Optimization

A perhaps even more famous application of large-scale differentiation is to nonlinear optimization. Suppose we have a scalar-valued function $f : \mathsf{R}^n \; ! \; \mathsf{R}$, and suppose we want to minimize (or maximize) f. For instance, in machine learning, we could have a big neural network (NN) with a vector x of a million parameters, and one tries to minimize a "loss" function f that compares the NN output to the desired results on "training" data. The most basic idea in optimization is to go "downhill" (see diagram) to make f as small as possible. If we can take the gradient of this function f, to go "downhill" we consider $\quad r\, f$, the direction of *steepest descent*, as depicted in Fig. 6.

Then, even if we have a million parameters, we can evolve all of them simultaneously in the downhill direction. It turns out that calculating all million derivatives costs about the same as evaluating the function at a point once (using reverse-mode/adjoint/left-to-right/backpropagation methods). Ultimately, this makes large-scale optimization practical for training neural nets, optimizing shapes of airplane wings, optimizing portfolios, etc.

Of course, there are many practical complications that make nonlinear optimization tricky (far more than can

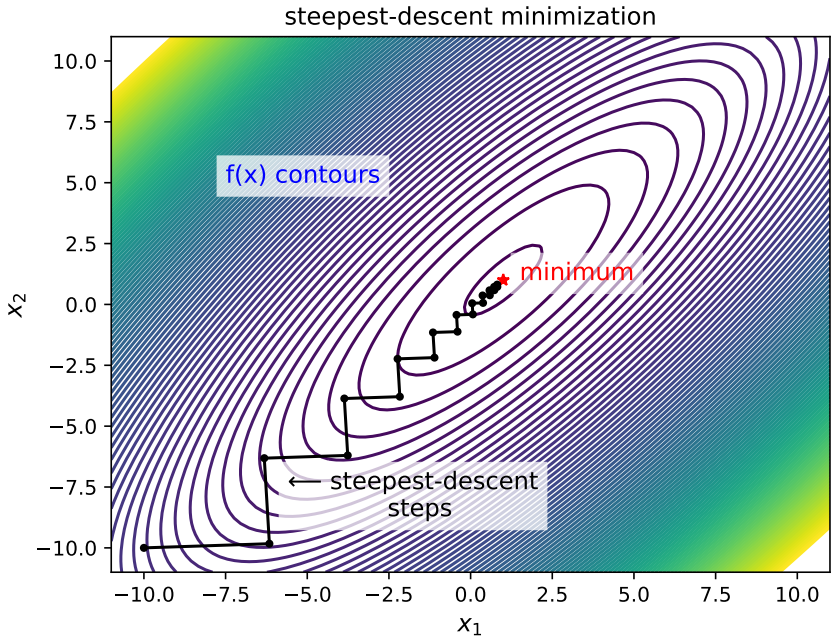


Figure 6: A *steepest-descent algorithm* minimizes a function $f(x)$ by taking successive "downhill" steps in the direction $-\nabla f$. (In the example shown here, we are minimizing a quadratic function in two dimensions $x \in \mathbb{R}^2$, performing an exact 1d minimization in the downhill direction for each step.) Steepest-descent algorithms can sometimes "zig-zag" along narrow valleys, slowing convergence (which can be counteracted in more sophisticated algorithms by "momentum" terms, second-derivative information, and so on).

be covered in a single lecture, or even in a whole course!), but we give some examples of now.

For instance, even though we can compute the "downhill direction", how far do we need to step in that direction? (In machine learning, this is sometimes called the "learning rate.") Often, you want to take "as big of a step as you can" to speed convergence, but you don't want the step to be too big because ∇f only tells you a *local* approximation of f. There are many different ideas of how to determine this:

- Line search: using a 1D minimization to determine how far you step.
- A "trust region" bounding the step size (where we trust the derivative-based approximation of f). There are many techniques to evolve the size of the trust region as optimization progresses.

We may also need consider constraints, for instance minimizing $f(x)$ subject to $g_k(x) \leq 0$ (points x satisfying the constraints are called "feasible"). One typically uses a combination of ∇f and ∇g_k to approximate (e.g. linearize) the problem and make progress towards the best feasible point.

If you just go straight downhill, you might "zig-zag" along narrow valleys, making convergence very slow. There are a few options to combat this, such as "momentum" terms and conjugate gradients. Even fancier than these techniques, one might estimate second-derivative "Hessian matrices" from a sequence of ∇f values—a famous version of this is known as the BFGS algorithm—and use it to take approximate Newton steps (for the root $\nabla f = 0$). (We'll return to Hessians in a later lecture.)

Ultimately, there are a lot of techniques and a zoo of competing algorithms that you might need to experiment with to find the best approach for a given problem. (There are many books on optimization algorithms, and even a whole book can only cover a small slice of what is out there!)

Some parting advice: Often the main trick is less about the choice of algorithms than it is about finding the right mathematical formulation of your *problem*—e.g. what function, what constraints, and what parameters should you be considering—to match your problem to a good algorithm. However, if you have *many* (10) parameters, *try hard* to use an analytical gradient (not finite differences), computed efficiently in reverse mode.

8.2.2 Engineering/Physical Optimization

There are many, many applications of optimization besides machine learning (fitting models to data). It is interesting to also consider engineering/physical optimization. (For instance, suppose you want to make an airplane wing that is as strong as possible.) The general outline of such problems is typically:

1. You start with some design parameters $\mathbf{p}$, e.g. describing the geometry, materials, forces, or other degrees of freedom.
2. These $\mathbf{p}$ are then used in some physical model(s), such as solid mechanics, chemical reactions, heat transport, electromagnetism, acoustics, etc. For example, you might have a linear model of the form $A(\mathbf{p})x = b(\mathbf{p})$ for some matrix A (typically very large and sparse).
3. The solution of the physical model is a solution $x(\mathbf{p})$. For example, this could the mechanical stresses, chemical concentrations, temperatures, electromagnetic fields, etc.
4. The physical solution $x(\mathbf{p})$ is the input into some design objective $f(x(\mathbf{p}))$ that you want to improve/optimize. For instance, strength, speed power, efficiency, etc.
5. To maximize/minimize $f(x(\mathbf{p}))$, one uses the gradient $\nabla_{\mathbf{p}} f$, computed using reverse-mode/"adjoint" methods, to update the parameters $\mathbf{p}$ and improve the design.

As a fun example, researchers have even applied "topology optimization" to design a chair, optimizing every voxel of the design—the parameters $\mathbf{p}$ represent the material present (or not) in every voxel, so that the optimization discovers not just an optimal shape but an optimal *topology* (how materials are connected in space, how many holes there are, and so forth)—to support a given weight with *minimal material*. To see it in action, watch this chair-optimization video. (People have applied such techniques to much more practical problems as well, from airplane wings to optical communications.)

8.3 Reverse-mode "Adjoint" Differentiation

But what is adjoint differentiation—the method of differentiating that makes these applications actually feasible to apply? Ultimately, it is yet another example of left-to-right/reverse-mode differentiation, essentially applying the chain rule from outputs to inputs. Consider, for example, trying to compute the gradient of the scalar function $f(x(p))$ where $A(p)x = b$. Then,

$$df = f'(x)[dx] = f'(x)d(A^{-1})b = \underbrace{f'(x)A^{-1}}_{v^T} dA\, A^{-1}b.$$

Grouping the terms left-to-right, we first solve the "adjoint" (transposed) equation $A^T v = f(x)^T$ for v, and then we obtain $df = v^T dA\, x$. For any given parameter p_k, $\partial f/\partial p_k = v^T \partial A \partial p_k x$ (and usually $\partial A/\partial p_k$ is very sparse). That is, it takes only *two solves* to get both f and ∇f—one for solving $Ax = b$ to find $f(x)$, and another with A^T for v, after which all of the derivatives $\partial f/\partial p_k$ are just some cheap dot products.

Note that you should *not* use right-to-left "forward-mode" derivatives with lots of parameters, because

$$\frac{\partial f}{\partial p_k} = f'(x)\left(A^{-1}\frac{\partial A}{\partial p_k}x\right)$$

represents one solve per parameter p_k! Right-to-left (a.k.a. forward mode) is only better when there is one (or few) input parameters p_k and many outputs, while left-to-right "adjoint" differentiation is better when there is one (or few) output values and many input parameters. (In a later class, we will discuss using dual numbers for differentiation, and this also corresponds to forward mode.)

Another possibility that might come to mind is to use finite differences, but you should not use this if you have lots of parameters! Finite differences would involve a calculation of something like

$$\frac{\partial f}{\partial p_k} \approx [f(p + \epsilon e_k) - f(p)]/\epsilon,$$

where e_k is a unit vector in the k-th direction and ϵ is a small number. This, however, requires one solve for each parameter p_k, just like forward-mode differentiation. (It becomes even more expensive if you use fancier higher-order finite-difference approximations in order to obtain higher accuracy.)

You could also consider adjoint/reverse differentiation for nonlinear equations. For instance, consider the gradient of the scalar function $f(x(p))$ where $x(p) \in \mathbb{R}^n$ solves $g(p, x) = 0 \in \mathbb{R}^n$. By the Implicit Function Theorem,

$$g(p,x) = 0 \implies \frac{\partial g}{\partial p}dp + \frac{\partial g}{\partial x}dx = 0 \implies dx = -\left(\frac{\partial g}{\partial x}\right)^{-1}\frac{\partial g}{\partial p}dp\,.$$

Hence,

$$df = f'(x)dx = \underbrace{-f'(x)\left(\frac{\partial g}{\partial x}\right)^{-1}}_{v^T}\frac{\partial g}{\partial p}dp\,.$$

Associating left-to-right again leads to a single "adjoint" equation: $(\partial g/\partial x)^T v = f'(x)^T$. In other words, it again only takes two solves to get both f and ∇f—one nonlinear "forward" solve for x and one linear "adjoint" solve for v! Thereafter, all derivatives $\partial f/\partial p_k$ are cheap dot products. (Note that the linear "adjoint" solve involves the transposed Jacobian $\partial g/\partial x$. Except for the transpose, this is very similar to the cost of a single Newton step to solve $g = 0$ for x. So the adjoint problem should be cheaper than the forward problem.)

Lastly, note that you need to understand adjoint methods even if you use automatic differentiation. Firstly, it helps you understand when to use forward- vs. reverse-mode automatic differentiation. Secondly, many physical models call large software packages written over the decades in various languages that ***cannot be differentiated automatically*** by AD. You can typically correct this by just supplying a "vector–Jacobian product" $y^T dx$ for this physics, or even just part of the physics, and then AD will differentiate the rest and apply the chain rule for you. Lastly, often models involve approximate calculations (e.g. for the iterative solution of linear or nonlinear equations, numerical integration, and so forth), but AD tools don't know this and spend extra effort trying to differentiate the error in your approximation; in such cases, manually written derivative rules can sometimes be much more efficient. (For example, suppose your model involves solving a nonlinear system $g(x, p) = 0$ by an iterative approach like Newton's method. AD will be very inefficient because it will attempt to differentiate through all your Newton steps. Assuming that you converge your Newton solver to enough accuracy that the error is negligible, it is much more efficient to perform differentiation via the implicit-function theorem as described above, leading to a single linear adjoint solve.)

9 Derivative of Matrix Determinant and Inverse

9.1 Two Derivations

This section of notes follows this Julia notebook. This notebook is a little bit short, but is an important and useful calculation.

Theorem 32

Given A is a square matrix, we have

$$\nabla(\det A) = \text{cofactor}(A) = (\det A)A^{-T} := \text{adj}(A^T) = \text{adj}(A)^T$$

where adj is the "adjugate". (You may not have heard of the matrix adjugate, but this formula tells us that it is simply $\text{adj}(A) = \det(A)A^{-1}$, or $\text{cofactor}(A) = \text{adj}(A^T)$.) Furthermore,

$$d(\det A) = \text{tr}(\det(A)A^{-1}dA) = \text{tr}(\text{adj}(A)dA) = \text{tr}(\text{cofactor}(A)^T dA).$$

You may remember that each entry (i, j) of the cofactor matrix is $(-1)^{i+j}$ times the determinant obtained by deleting row i and column j from A. Some short calculations to obtain some intuition about these functions:

$$M = \begin{bmatrix} a & c \\ b & d \end{bmatrix} \tag{1}$$

$$\implies \text{cofactor}(M) = \begin{bmatrix} d & -c \\ -b & a \end{bmatrix} \tag{2}$$

$$\text{adj}(M) = \begin{bmatrix} d & -b \\ -c & a \end{bmatrix} \tag{3}$$

$$(M)^{-1} = \frac{1}{ad - bc}\begin{bmatrix} d & -b \\ -c & a \end{bmatrix}. \tag{4}$$

Numerically, as is done in the notebook, you can construct a random $n \times n$ matrix A (say, 9×9), consider $dA = .00001A$, and see that numerically,

$$\det(A + dA) - \det(A) \approx \text{tr}(\text{adj}(A)dA),$$

which numerically supports our claim for the theorem.

We now prove the theorem in two ways. Firstly, there is a direct proof where you just differentiate the scalar with respect to every input using the cofactor expansion of the determinant based on the ith row. Recall that

$$\det(A) = A_{i1}C_{i1} + A_{i2}C_{i2} + \cdots + A_{in}C_{in}.$$

Thus,

$$\frac{\partial \det A}{\partial A_{ij}} = C_{ij} \implies \nabla(\det A) = C,$$

the cofactor matrix.

There is also a fancier proof of the theorem using linearization near the identity. Firstly, note that it is easy to see from the properties of determinants that

$$\det(I + dA) - 1 \approx \text{tr}(dA),$$

and thus

$$\begin{aligned}\det(A + A(A^{-1}dA)) \quad \det(A) &= \det(A)(\det(I + A^{-1}dA) \quad 1) \\ &= \det(A)\operatorname{tr}(A^{-1}dA) = \operatorname{tr}(\det(A)A^{-1}dA) \\ &= \operatorname{tr}(\operatorname{adj}(A)dA.\end{aligned}$$

This also implies the theorem.

9.2 Applications

9.2.1 Characteristic Polynomial

We now use this as an application to find the derivative of a characteristic polynomial evaluated at x. Let $p(x) = \det(xI \quad A)$, a scalar function of x. Recall that through factorization, $p(x)$ may be written in terms of eigenvalues λ_i. So we may ask: what is the derivative of $p(x)$, the characteristic polynomial at x? Using freshman calculus, we could simply compute

$$\frac{d}{dx}\prod_i(x \quad \lambda_i) = \sum_i \prod_{j \neq i}(x \quad \lambda_j) = \prod(x \quad \lambda_i)\sum_i(x \quad \lambda_i)^{-1},$$

as long as $x \neq \lambda_i$.

This is a perfectly good simply proof, but with our new technology we have a new proof:

$$\begin{aligned}d(\det(xI \quad A)) &= \det(xI \quad A)\operatorname{tr}((xI \quad A)^{-1}d(xI \quad A)) \\ &= \det(xI \quad A)\operatorname{tr}(xI \quad A)^{-1}dx.\end{aligned}$$

Note that here we used that $d(xI \quad A) = dx\, I$ when A is constant and $\operatorname{tr}(Adx) = \operatorname{tr}(A)dx$ since dx is a scalar.

We may again check this computationally as we do in the notebook.

9.2.2 The Logarithmic Derivative

We can similarly compute using the chain rule that

$$d(\log(\det(A))) = \frac{d(\det A)}{\det A} = \det(A^{-1})d(\det(A)) = \operatorname{tr}(A^{-1}dA).$$

The logarithmic derivative shows up a lot in applied mathematics.

For instance, recall Newton's method to find roots $f(x) = 0$ of single-variable real-valued functions $f(x)$ by taking a sequence of steps $x \mapsto x + \delta x$. The key formula in Newton's method is $\delta x = f'(x)^{-1}f(x)$, but this is the same as $\frac{1}{(\log f(x))'}$. So, derivatives of log determinants show up in finding *roots of determinants*, i.e. for $f(x) = \det M(x)$. When $M(x) = A \quad xI$, roots of the determinant are eigenvalues of A. For more general functions $M(x)$, solving $\det M(x) = 0$ is therefore called a *nonlinear eigenproblem*.

9.3 Jacobian of the Inverse

Lastly, we compute the derivative (as both a linear operator and an explicit Jacobian matrix) of the inverse of a matrix. There is a neat trick to obtain this derivative, simply from the property $A^{-1}A = I$ of the inverse. By the product rule, this implies that

$$d(A^{-1}A) = d(I) = 0 = d(A^{-1})A + A^{-1}dA = (A^{-1})'[dA]\,.$$

Here, the right-hand side defines a perfectly good linear operator for the derivative $(A^{-1})'$, but if we want we can rewrite this as an explicit Jacobian matrix by using Kronecker products acting on the "vectorized" matrices as we did in Sec. 4:

$$\mathrm{vec}\left(d(A^{-1})\right) = \mathrm{vec}\left(-A^{-1}(dA)A^{-1}\right) = \underbrace{-(A^{-T}\otimes A^{-1})}_{\text{Jacobian}}\,\mathrm{vec}(dA)\,,$$

where A^{-T} denotes $(A^{-1})^T = (A^T)^{-1}$. One can check this formula numerically, as is done in the notebook.

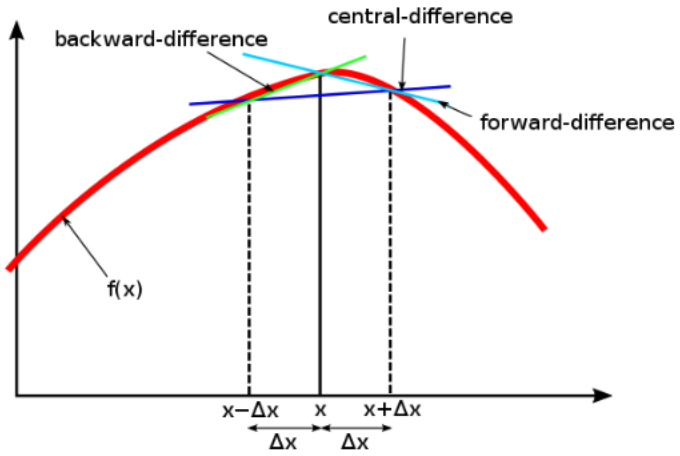
10 Forward and Reverse-Mode Automatic Differentiation

10.1 Automatic Di□erentiation via Dual Numbers

The first time that Professor Edelman had heard about automatic differentiation (AD), it was easy for him to imagine what it was, though what he imagined was wrong. In his head, he thought it was straightforward symbolic differentiation applied to code—sort of like executing Mathematica or Maple, or even just automatically doing what he learned to do in his calculus class. For instance, just plugging in functions and their domains in the following table:

Derivative	Domain
$(\sin x)' = \cos x$	$-1 < x < 1$
$(\cos x)' = -\sin x$	$-1 < x < 1$
$(\tan x)' = \sec^2 x$	$x \neq \frac{\pi}{2} + \pi n, n \in \mathbb{Z}$
$(\cot x)' = -\csc^2 x$	$x \neq \pi n, n \in \mathbb{Z}$
$(\sec x)' = \tan x \sec x$	$x \neq \frac{\pi}{2} + \pi n, n \in \mathbb{Z}$
$(\csc x)' = -\cot x \csc x$	$x \neq \pi n, n \in \mathbb{Z}$

And in any case, if it wasn't just like executing Mathematica or Maple, then it must be finite differences, like one learns in a numerical computing class (or as we did in Sec. 6). That is, something like the below diagram:



It turns out that it is definitely not the latter—AD algorithms are generally exact (in exact arithmetic, neglecting roundoff errors). But it also doesn't look much like the former: the computer doesn't quite construct a big "unrolled" symbolic expression and then differentiate it, the way you might imagine doing by hand or via computer-algebra software. For example, imagine a computer program that computes $\det A$ for an $n \times n$ matrix—writing down the "whole" symbolic expression isn't possible until the program runs and n is known (e.g. input by the user), and in any case a naive symbolic expression would require $n!$ terms. AD systems have to deal with computer-programming constructs like loops, recursion, and problem sizes n that are unknown until the program runs, while at the same time avoiding constructing symbolic expressions whose size becomes prohibitively large. The design of AD systems often ends up being as much about compilers as it is about calculus.

One AD approach that can be explained relatively simply is "forward-mode" AD, which can be implemented by carrying out the computation of f' in *tandem* with the computation of f. One augments every intermediate value a in the computer program with another value b that represents its derivative, along with chain rules to propagate

these derivatives as values in the program are combined. It turns out that this can be thought of as replacing real numbers (values) with a new kind of "dual number" $D(a, b)$ (values + derivatives) and corresponding arithmetic rules, as explained below.

10.1.1 Example: Babylonian square root

We start with a simple example, the square-root function, where a practical method of automatic differentiation came as both a mathematical surprise and a computing wonder for Professor Edelman. The example is the Babylonian algorithm to compute $\sqrt{x}$, known for millennia (and later revealed as a special case of Newton's method applied to $t \quad x^2 = 0$): simply repeat $(t + x/t)/2 \;!\; t$ until t converges to $\sqrt{x}$. Each iteration has one addition and two divisions. For illustration purposes, 10 iterations suffice. We can also check our answer using ForwardDiff, and can use this to compute the square root of various numbers.

We can use the Babylonian algorithm to compute derivatives as well using dual numbers of the form $D(a, b)$, explained below.

10.1.2 Dual numbers

We can think of a dual number $D(a, b)$ as $a + b\epsilon$, where ϵ is a new "infinitesimal" quantity that satisfies $\epsilon^2 = 0$. This is similar to imaginary numbers, where i is introduced as a non-zero number such that $i^2 = \quad 1$, which is how some people imagine this notation. Others like to think of how engineers just drop the $O(\epsilon^2)$ terms when ϵ is simply a very small number. The four algebraic rules for these "dual numbers" are as follows:

$$\begin{aligned}(a + b\epsilon) \quad (c + d\epsilon) &= (a \quad c_+(b \quad d)\epsilon \\ (a + b\epsilon) \;\; (c + d\epsilon) &= (ac) + (bc + ad)\epsilon \\ \frac{a + b\epsilon}{c + d\epsilon} &= \frac{a}{c} + \frac{bc \quad ad}{c^2\epsilon}.\end{aligned}$$

The ϵ coefficients of these rules correspond to the sum/difference, product, and quotient rules of differential calculus! We can implement these rules into Julia—defining a new `D` (dual) numeric type and corresponding arithmetic operations—and use our code to do computations, as can be seen in the notebook. Simply by plugging a dual number as the input into the Babylonian algorithm, we thereby obtain both the value (the "real" part) and the derivative (the ϵ coefficient) of $f(x) = \sqrt{x}$ at different points x! (This is not a finite-difference approximation, it is an *exact* computation of the derivative of the Babylonian-algorithm result, limited only by computer-arithmetic roundoff errors.)

10.2 Automatic Di□erentiation via Computational Graphs

Let's now get into automatic differentiation via computational graphs. For this section, we consider the following simple motivating example.

Example 33

Define the following functions:

$$\begin{cases} a(x, y) = \sin x \\ b(x, y) = \frac{1}{y} \quad a(x, y) \\ z(x, y) = b(x, y) + x. \end{cases}$$

Compute $\frac{\partial z}{\partial x}$ and $\frac{\partial z}{\partial y}$.

There are a few ways to solve this problem. Firstly, of course, one can compute this symbolically, noting that

$$z(x,y) = b(x,y) + x = \frac{1}{y}a(x,y) + x = \frac{\sin x}{y} + x,$$

which implies

$$\frac{\partial z}{\partial x} = \frac{\cos x}{y} + 1 \quad \text{and} \quad \frac{\partial z}{\partial y} = \quad \frac{\sin x}{y^2}.$$

However, one can also use a Computational Graph (see Figure of Computational Graph below) where the edge from node A to node B is labelled with $\frac{\partial B}{\partial A}$.

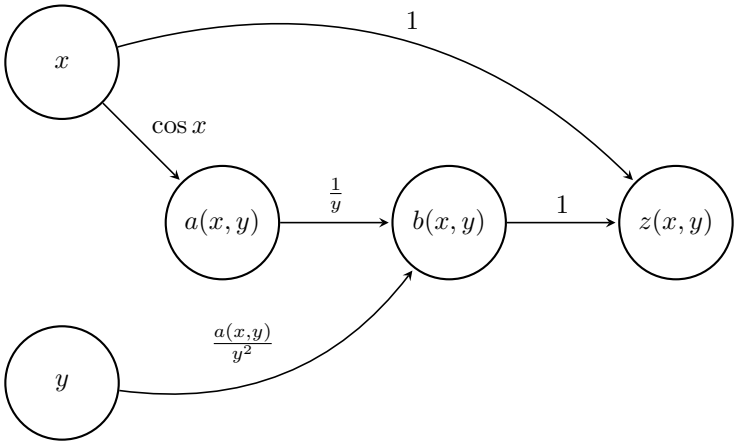


Figure 7: Figure of Computational Graph

Now how do we use this directed acyclic graph (DAG) to find the derivatives? Well one view (called the "forward view") is given by following the paths from the inputs to the outputs and (left) multiplying as you go, adding together multiple paths. For instance, following this procedure for paths from x to $z(x,y)$, we have

$$\frac{\partial z}{\partial x} = 1 \quad \frac{1}{y} \quad \cos x + 1 = \frac{\cos x}{y} + 1.$$

Similarly, for paths from y to $z(x,y)$, we have

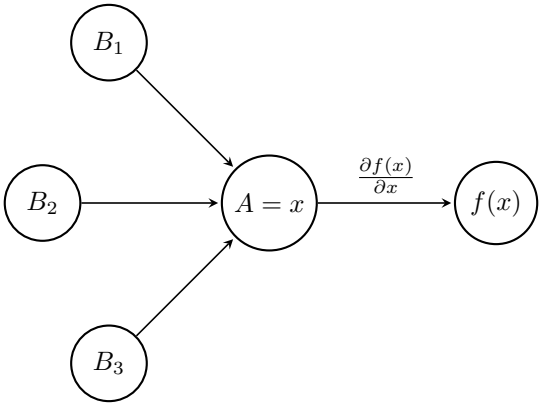
$$\frac{\partial z}{\partial y} = 1 \quad \frac{a(x,y)}{y^2} = \frac{\sin x}{y^2},$$

and if you have numerical derivatives on the edges, this algorithm works. Alternatively, you could follow a reverse view and follow the paths backwards (multiplying right to left), and obtain the same result. Note that there is nothing magic about these being scalar here– you could imagine these functions are the type that we are seeing in this class and do the same computations! The only thing that matters here fundamentally is the associativity. However, the when considering vector-valued functions, the order in which you multiply the edge weights is vitally important (as vector/matrix valued functions are not generally commutative).

The graph-theoretic way of thinking about this is to consider "path products". A path product is the product of edge weights as you traverse a path. In this way, we are interested in the sum of path products from inputs to outputs to compute derivatives using computational graphs. Clearly, we don't particularly care which order we traverse the paths as long as the *order* we take the product in is correct. In this way, forward and reverse-mode automatic differentiation is not so mysterious.

Let's take a closer view of the implementation of forward-mode automatic differentiation. Suppose we have at

the node x in computing the derivative as seen in the figure below:



Suppose we know the path product before x on the LHS, suppose it is called P. Then what is the new path product as we move to the right?: $f'(x)\ \ P$! So we need a data structure that maps in the following way:

$$(\text{value}, \text{path product}) \mapsto (f(\text{value}), f'\ \ \text{path product}).$$

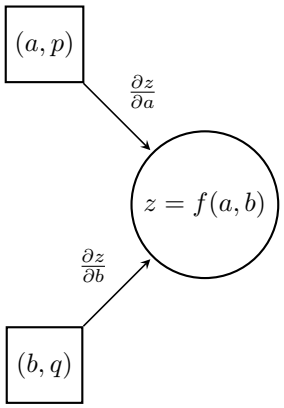
In some sense, this is another way to look at the Dual Numbers– taking in our path products and spitting out values. In any case, we overload our program which can easily calculate $f(\text{value})$ and tack-on $f'\ \ $(path product).

One might ask how our program starts– this is how the program works in the "middle", but what should our starting value be? Well the only thing it can be for this method to work is $(x, 1)$. Then, at every step you do the following map listed above:

$$(\text{value}, \text{path product}) \mapsto (f(\text{value}), f'\ \ \text{path product}),$$

and at the end we obtain our derivatives.

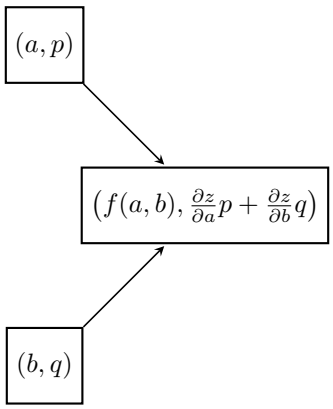
Now how do we combine arrows? In other words, suppose at the two notes on the LHS we have the values (a, p) and (b, q), as seen in the diagram below: So here, we aren't thinking of a, b as numbers, but as variables. What



should the new output value be? We want to add the two path products together, obtaining

$$\left(f(a, b), \frac{\partial z}{\partial a}p + \frac{\partial z}{\partial b}q\right).$$

So really, our overloaded data structure looks like this:



This diagram of course generalizes if we may many different nodes on the left side of the graph.

If we come up with such a data structure for all of the simple computations (addition/subtraction, multiplication, and division), and if this is all we need for our computer program, then we are set! Here is how we define the structure for addition/subtraction, multiplication, and division.

Addition/Subtraction:

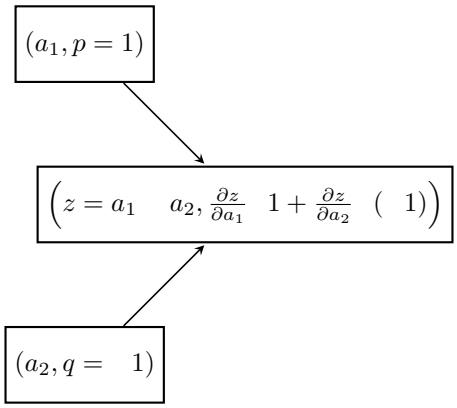


Figure 8: Figure of Addition/Subtraction Computational Graph

Multiplication: See figure below.

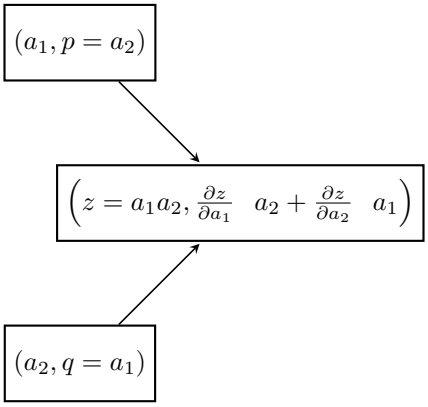


Figure 9: Figure of Multiplication Computational Graph

Division: See figure below.

In theory, these three graphs are all we need, and we can use Taylor series expansions for more complicated functions. But in practice, we throw in what the derivatives of more complicated functions are so that we don't waste our time trying to compute something we already know, like the derivative of sine or of a logarithm.

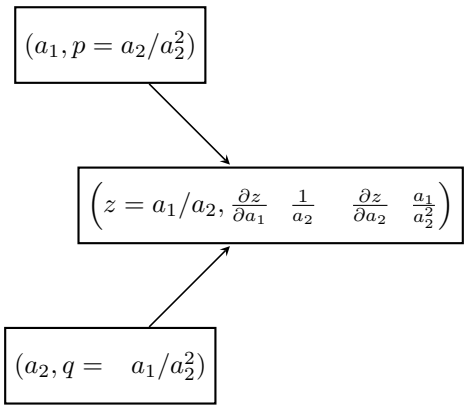
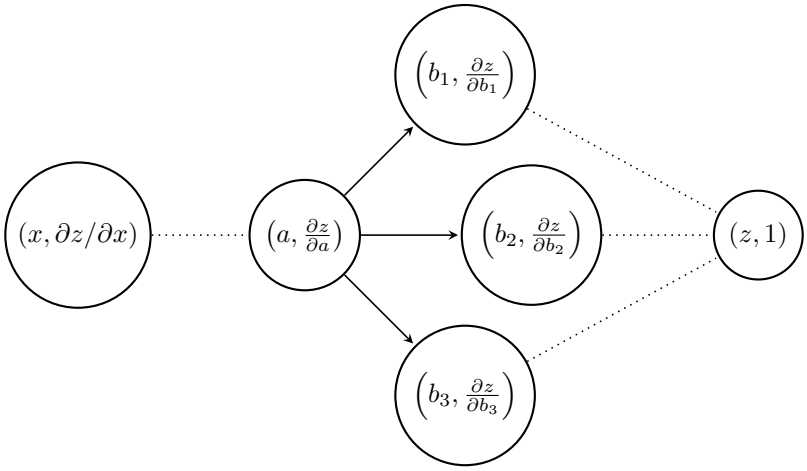


Figure 10: Figure of Division Computational Graph

10.2.1 Reverse Mode Automatic Differentiation on Graphs

When we do reverse mode, we have arrows going the other direction, which we will understand in this section of the notes. In forward mode it was all about "what do we depend on", i.e. computing the derivative on the right hand side of the above diagram using the functions in the nodes on the left. In reverse mode, the question is really "what are we influenced by?", or "what do we influence later"?

When going "backwards", we need know what nodes a given node influences. For instance, given a node A, we want to know the nodes B_i that is influenced by, or depends on, node A. So now our diagram looks like this:



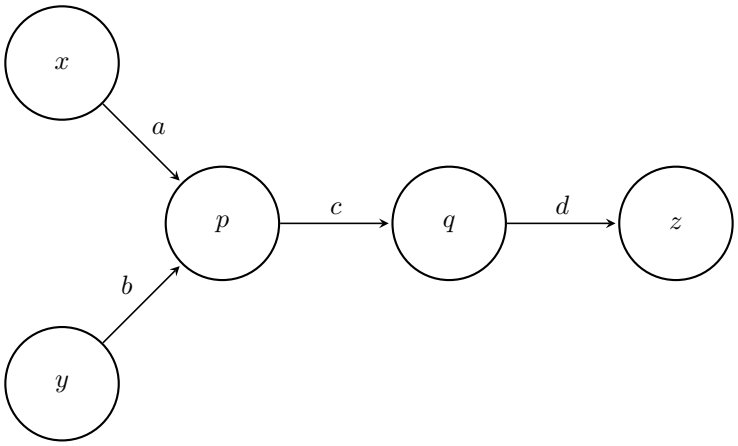
So now, we eventually have a final node $(z, 1)$ (far on the right hand side) where everything starts. This time, all of our multiplications take place from right to left as we are in reverse mode. Our goal is to be able to calculate the node $(x, \partial z/\partial x)$. So if we know how to fill in the $\frac{\partial z}{\partial a}$ term, we will be able to go from right to left in these computational graphs (i.e., in reverse mode). In fact, the formula for getting $\frac{\partial z}{\partial a}$ is given by

$$\frac{\partial z}{\partial a} = \sum_{i=1}^{s} \frac{\partial b_i}{\partial a} \frac{\partial z}{\partial b_i}$$

where the b_is come from the nodes that are influenced by the node A. This is again just another chain rule like from calculus, but you can also view this as multiplying the sums of all the weights in the graph influenced by A.

Why can reverse mode be more efficient than forward mode? One reason it because it can save data and use it

later. Take, for instance, the following sink/source computational graph.



If x, y here are our sources, and z is our sink, we want to compute the sum of products of weights on paths from sources to sinks. If we were using forward mode, we would need to compute the paths dca and dcb, which requires four multiplications (and then you would add them together). If we were using reverse mode, we would only need compute $a\underline{cd}$ and $b\underline{cd}$ and sum them; notice reverse mode (since we need only compute cd once), only takes 3 multiplications. In general, this can more efficiently resolve certain types of problems, such as the source/sink one.

11 Differentiating ODE solutions

In this lecture, we will consider the problem of differentiating the *solution* of ordinary differential equations (ODEs) with respect to parameters that appear in the equations and/or initial conditions. This is as important topic in a surprising number of practical applications, such as evaluating the effect of uncertainties, fitting experimental data, or machine learning (which is increasingly combining ODE models with neural networks). As in previous lectures, we will find that there are crucial practical distinctions between "forward" and "reverse" ("adjoint") techniques for computing these derivatives, depending upon the number of parameters and desired outputs.

Although a basic familiarity with the concept of an ODE will be helpful to readers of this lecture, we will begin with a short review in order to establish our notation and terminology.

The video lecture on this topic for IAP 2023 was given by Dr. Frank Schäfer (MIT). These notes follow the same basic approach, but differ in some minor notational details.

11.1 Ordinary di□erential equations (ODEs)

An **ordinary differential equation** (**ODE**) is an equation for a function $u(t)$ of "time"[3] $t \in \mathbb{R}$ in terms of one or more derivatives, most commonly in the **first-order** form

$$\frac{du}{dt} = f(u, t)$$

for some right-hand-side function f. Note that $u(t)$ need not be a scalar function—it could be a column vector $u \in \mathbb{R}^n$, a matrix, or any other differentiable object. One could also write ODEs in terms of higher derivatives d^2u/dt^2 and so on, but it turns out that one can write any ODE in terms of first derivatives alone, simply by making u a vector with more components.[4] To uniquely determine a solution of a first-order ODE, we need some additional information, typically an **initial value** $u(0) = u_0$ (the value of u at $t = 0$), in which case it is called an **initial-value problem**. These facts, and many other properties of ODEs, are reviewed in detail by many textbooks on differential equations, as well as in classes like 18.03 at MIT.

ODEs are important for a huge variety of applications, because the behavior of many realistic systems is defined in terms of rates of change (derivatives). For example, you may recall Newton's laws of mechanics, in which acceleration (the derivative of velocity) is related to force (which may be a function of time, position, and/or velocity), and the solution $u = [\text{position}, \text{velocity}]$ of the corresponding ODE tells us the trajectory of the system. In chemistry, u might represent the concentrations of one or more reactant molecules, with the right-hand side f providing reaction rates. In finance, there are ODE-like models of stock or option prices. *Partial* differential equations (PDEs) are more complicated versions of the same idea, for example in which $u(x, t)$ is a function of space x as well as time t and one has $\frac{\partial u}{\partial t} = f(u, x, t)$ in which f may involve some spatial derivatives of u.

In linear algebra (e.g. 18.06 at MIT), we often consider initial-value problems for *linear* ODEs of the form $du/dt = Au$ where u is a column vector and A is a square matrix; if A is a constant matrix (independent of t or u), then the solution $u(t) = e^{At}u(0)$ can be described in terms of a matrix exponential e^{At}. More generally, there are many tricks to find explicit solutions of various sorts of ODEs (various functions f). However, just as one cannot find explicit formulas for the integrals of most functions, there is no explicit formula for the solution of *most* ODEs,

[3] Of course, the independent variable need not be time, it just needs to be a real scalar. But in a generic context it is convenient to imagine ODE solutions as evolving in time.

[4] For example, the second-order ODE $\frac{d^2v}{dt^2} + \frac{dv}{dt} = h(v, t)$ could be re-written in first-order form by defining $u = \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} = \begin{pmatrix} v \\ dv/dt \end{pmatrix}$, in which case $du/dt = f(u, t)$ where $f = \begin{pmatrix} u_2 \\ h(u_1, t) - u_2 \end{pmatrix}$.

and in many practical applications one must resort to approximate numerical solutions. Fortunately, if you supply a computer program that can compute $f(u,t)$, there are mature and sophisticated software libraries[5] which can compute $u(t)$ from $u(0)$ for any desired set of times t, to any desired level of accuracy (for example, to 8 significant digits).

For example, the most basic numerical ODE method computes the solution at a sequence of times $t_n = n\Delta t$ for $n = 0, 1, 2, \ldots$ simply by approximating $\frac{du}{dt} = f(u,t)$ using the finite difference $\frac{u(t_{n+1}) \quad u(t_n)}{\Delta t} \quad f(u(t_n), t_n)$, giving us the "explicit" timestep algorithm:

$$u(t_{n+1}) \quad u(t_n) + \Delta t\, f(u(t_n), t_n).$$

Using this technique, known as "Euler's method," we can march the solution forward in time: starting from our initial condition u_0, we compute $u(t_1) = u(\Delta t)$, then $u(t_2) = u(2\Delta t)$ from $u(\Delta t)$, and so forth. Of course, this might be rather inaccurate unless we make Δt very small, necessitating many timesteps to reach a given time t, and there can arise other subtleties like "instabilities" where the error may accumulate exponentially rapidly with each timestep. It turns out that Euler's method is mostly obsolete: there are much more sophisticated algorithms that robustly produce accurate solutions with far less computational cost. However, they all resemble Euler's method in the conceptual sense: they use evaluations of f and u at a few nearby times t to "extrapolate" u at a subsequent time somehow, and thus march the solution forwards through time.

Relying on a computer to obtain numerical solutions to ODEs is practically essential, but it can also make ODEs a lot more fun to work with. If you ever took a class on ODEs, you may remember a lot of tedious labor (tricky integrals, polynomial roots, systems of equations, integrating factors, etc.) to obtain solutions by hand. Instead, we can focus here on simply setting up the correct ODEs and integrals and trust the computer to do the rest.

11.2 Sensitivity analysis of ODE solutions

Often, ODEs depend on some additional parameters $p \in \mathbb{R}^N$ (or some other vector space). For example, these might be reaction-rate coefficients in a chemistry problem, the masses of particles in a mechanics problem, the entries of the matrix A in a linear ODE, and so on. So, you really have a problem of the form

$$\frac{\partial u}{\partial t} = f(u, p, t),$$

where the solution $u(p,t)$ depends both on time t and the parameters p, and in which the initial condition $u(p,0) = u_0(p)$ may also depend on the parameters.

The question is, how can we compute the derivative $\partial u/\partial p$ of the solution with respect to the parameters of the ODE? By this, as usual, we mean the linear operator that gives the first-order change in u for a change in p:

$$u(p + dp, t) \quad u(p,t) = \frac{\partial u}{\partial p} dp,$$

where of course $\partial u/\partial p$ depends on p and t. This kind of question is commonplace. For example, it is important in:

> Uncertainty quantification (UQ): if you have some uncertainty in the parameters of your ODE (for example, you have a chemical reaction in which the reaction rates are only known experimentally some measurement errors), the derivative $\partial u/\partial p$ tells you (to first order, at least) how sensitive your answer is to each of these uncertainties.

[5] For a modern and full-featured example, see the DifferentialEquations.jl suite of ODE solvers in the Julia language.

Optimization and fitting: often, you want to choose the parameters p to maximize or minimize some objective (or "loss" in machine learning). For example, if your ODE models some chemical reaction with unknown reaction rates or other parameters p, you might want to *fit* the parameters p to minimize the difference between $u(p,t)$ and some experimentally observed concentrations.

In the latter case of optimization, you have a **scalar objective function** of the solution, since to minimize or maximize something you need a real number (and u might be a vector). For example, this could take on one of the following two forms:

1. A real-valued function $g(u(p,T),T)$ *2* R that depends on the solution $u(p,T)$ at a particular time T. For example, if you have an experimental solution u (t) that you are are trying to match at $t = T$, you might minimize $g(u(p,T),T) = ku(p,T)$ u $(T)k^2$.
2. A real-valued function $G(p) = \int_0^T g(u(p,t),t)dt$ that depends on an average (here scaled by T) over many times t *2* $(0,T)$ of our time-dependent g. In the example of fitting experimental data u (t), minimizing $G(p) = \int_0^T ku(p,t)$ u $(t)k^2dt$ corresponds to a least-square fit to minimize the error averaged over a time T (e.g. the duration of your experiment).

In both cases, since these are scalar-valued functions, for optimization/fitting one would like to know the gradient $r_p g$ or $r_p G$, such that, as usual,

$$g(u(p+dp,t),t) \quad g(u(p,t),t) = (r_p g)^T dp$$

so that $r_p g$ is the steepest ascent/descent direction for maximization/minimization of g, respectively.

These are "just derivatives," but probably you can see the difficulty: if we don't have a formula (explicit solution) for $u(p,t)$, only some numerical software that can crank out numbers for $u(p,t)$ given any parameters p and t, how do we apply differentiation rules to find $\partial u/\partial p$ or $r_p g$? Of course, we could use finite differences as in Sec. 6—just crank through numerical solutions for p and $p + \delta p$ and subtract them—but that will be quite slow if we want to differentiate with respect to many parameters (N 1), not to mention giving potentially poor accuracy. In fact, people often have *huge* numbers of parameters inside an ODE that they want to differentiate. Nowadays, our right-hand-side function $f(u,p,t)$ can even contain a *neural network* (this is called a "neural ODE") with thousands or millions (N) of parameters p, and we need all N of these derivatives $r_p g$ or $r_p G$ to minimize the "loss" function g or G. So, not only do we need to find a way to differentiate our ODE solutions (or scalar functions thereof), but these derivatives must be obtained *e ciently*. It turns out that there are two ways to do this, and both of them hinge on the fact that the derivative is obtained by *solving another ODE*:

Forward mode: $\frac{\partial u}{\partial p}$ turns out to solve *another* ODE that we can integrate with the same numerical solvers for u. This gives us all of the derivatives we could want, but the drawback is that the ODE for $\frac{\partial u}{\partial p}$ is larger by a factor of N than the original ODE for u, so it is only practical for small N (few parameters).

Reverse ("**adjoint**") mode: for scalar objectives, it turns out that $r_p g$ or $r_p G$ can be computed by solving a different ODE for an "adjoint" solution $v(p,t)$ of the *same size* as u, and then computing some simple integrals involving u (the "forward" solution) and v. This has the advantage of giving us all N derivatives with only about *twice* the cost of solving for u, regardless of the number N of parameters. The disadvantage is that, since it turns out that v must be integrated "backwards" in time (starting from an "initial" condition at $t = T$ and working back to $t = 0$) and depends on u, it is necessary to store $u(p,t)$ for all t *2* $[0,T]$ (rather than marching u forwards in time and discarding values from previous times when they are no longer needed), which can require a vast amount of computer memory for large ODE systems integrated over long times.

We will now consider each of these approaches in more detail.

11.2.1 Forward sensitivity analysis of ODEs

Let us start with our ODE $\frac{\partial u}{\partial t} = f(u, p, t)$, and consider what happens to u for a small change dp in p:

$$d\left(\frac{\partial u}{\partial t}\right) = \frac{\partial}{\partial p}\left(\frac{\partial u}{\partial t}\right) dp = \frac{\partial}{\partial t}\left(\frac{\partial u}{\partial p}\right) dp = d(f(u,p,t)) = \left(\frac{\partial f}{\partial u}\frac{\partial u}{\partial p} + \frac{\partial f}{\partial p}\right) dp,$$

where we have used the familiar rule (from multivariable calculus) of interchanging the order of partial derivatives—a property that we will re-derive explicitly for our generalized linear-operator derivatives in our lecture on Hessians and second derivatives. Equating the two sides, we see that we have an ODE

$$\boxed{\frac{\partial}{\partial t}\left(\frac{\partial u}{\partial p}\right) = \frac{\partial f}{\partial u}\frac{\partial u}{\partial p} + \frac{\partial f}{\partial p}}$$

for the derivative $\frac{\partial u}{\partial p}$, whose initial condition is obtained simply by differentiating the initial condition $u(p, 0) = u_0(p)$ for u:

$$\left.\frac{\partial u}{\partial p}\right|_{t=0} = \frac{\partial u_0}{\partial p}.$$

We can therefore plug this into any ODE solver technique (usually numerical methods, unless we are extremely lucky and can solve this ODE analytically for a particular f) to find $\frac{\partial u}{\partial p}$ at any desired time t. Simple, right?

The only thing that might seem a little weird here is the *shape* of the solution: $\frac{\partial u}{\partial p}$ is a linear operator, but how can the solution of an ODE be a linear operator? It turns out that there is nothing wrong with this, but it is helpful to think about a few examples:

If $u, p \in \mathbb{R}$ are scalars (that is, we have a single scalar ODE with a single scalar parameter), then $\frac{\partial u}{\partial p}$ is just a (time-dependent) number, and our ODE for $\frac{\partial u}{\partial p}$ is an ordinary scalar ODE with an ordinary scalar initial condition.

If $u \in \mathbb{R}^n$ (a "system" of n ODEs) and $p \in \mathbb{R}$ is a scalar, then $\frac{\partial u}{\partial p} \in \mathbb{R}^n$ is another column vector and our ODE for $\frac{\partial u}{\partial p}$ is another system of n ODEs. So, we solve two ODEs of the same size n to obtain u and $\frac{\partial u}{\partial p}$.

If $u \in \mathbb{R}^n$ (a "system" of n ODEs) and $p \in \mathbb{R}^N$ is a vector of N parameters, then $\frac{\partial u}{\partial p} \in \mathbb{R}^{n \times N}$ is an $n \times N$ Jacobian *matrix*. Our ODE for $\frac{\partial u}{\partial p}$ is effectivly system of nN ODEs for all the components of this matrix, with a matrix $\frac{\partial u_0}{\partial p}$ of nN initial conditions! Solving this "matrix ODE" with numerical methods poses no conceptual difficulty, but will generally require about N times the computational work of solving for u, simply because there are N times as many unknowns. This could be expensive if N is large!

This reflects our general observation of forward-mode differentiation: it is expensive when the number N of "input" parameters being differentiated is large. However, forward mode is straightforward and, especially for $N \lesssim 100$ or so, is often the first method to try when differentiating ODE solutions. Given $\frac{\partial u}{\partial p}$, one can then straightforwardly differentiate scalar objectives by the chain rule:

$$\nabla_p g\big|_{t=T} = \underbrace{\frac{\partial u}{\partial p}^T}_{\text{Jacobian}^T} \underbrace{\frac{\partial g}{\partial u}^T}_{\text{vector}}\Bigg|_{t=T},$$

$$\nabla_p G = \int_0^T \nabla_p g \, dt.$$

11.2.2 Reverse/adjoint sensitivity analysis of ODEs

For large N 1 and scalar objectives g or G (etc.), we can in principle compute derivatives *much* more efficiently, with about the same cost as computing u, by applying a "reverse-mode" or "adjoint" approach. In other lectures, we've obtained analogous reverse-mode methods simply by evaluating the chain rule left-to-right (outputs-to-inputs) instead of right-to-left. Conceptually, the process for ODEs is similar,[6] but algebraically the derivation is rather trickier and less direct. The key thing is that, if possible, we want to avoid computing $\frac{\partial u}{\partial p}$ explicitly, since this could be a prohibitively large Jacobian matrix if we have many parameters (p is large), especially if we have many equations (u is large).

In particular, let's start with our forward-mode sensitivity analysis, and consider the derivative $G' = (\nabla_p G)^T$ where G is the integral of a time-varying objective $g(u, p, t)$ (which we allow to depend explicitly on p for generality). By the chain rule,

$$G' = \int_0^T \left(\frac{\partial g}{\partial p} + \frac{\partial g}{\partial u} \frac{\partial u}{\partial p} \right) dt,$$

which involves our unwanted factor $\frac{\partial u}{\partial p}$. To get rid of this, we're going to use a weird trick (much like Lagrange multipliers) of adding *zero* to this equation:

$$G' = \int_0^T \left[\left(\frac{\partial g}{\partial p} + \frac{\partial g}{\partial u} \frac{\partial u}{\partial p} \right) + v^T \underbrace{\left(\frac{\partial}{\partial t} \left(\frac{\partial u}{\partial p} \right) \quad \frac{\partial f}{\partial u} \frac{\partial u}{\partial p} \quad \frac{\partial f}{\partial p} \right)}_{=0} \right] dt$$

for some function $v(t)$ of the *same shape as u* that multiplies our "forward-mode" equation for $\partial u/\partial p$. (If $u \in \mathbb{R}^n$ then $v \in \mathbb{R}^n$; more generally, for other vector spaces, read v^T as an inner product with v.) The new term $v^T(\quad)$ is zero because the parenthesized expression is precisely the ODE satisfied by $\frac{\partial u}{\partial p}$, as obtained in our forward-mode analysis above, *regardless* of $v(t)$. This is important because it allows us the freedom to *choose* $v(t)$ to *cancel* the unwanted $\frac{\partial u}{\partial p}$ term. In particular, if we first *integrate by parts* on the $v^T \frac{\partial}{\partial t} \left(\frac{\partial u}{\partial p} \right)$ term to change it to $\left(\frac{\partial v}{\partial t} \right)^T \frac{\partial u}{\partial p}$ plus a boundary term, then re-group the terms, we find:

$$G' = v^T \frac{\partial u}{\partial p} \Big|_0^T + \int_0^T \left[\frac{\partial g}{\partial p} \quad v^T \frac{\partial f}{\partial p} + \underbrace{\left(\frac{\partial g}{\partial u} \quad v^T \frac{\partial f}{\partial u} \quad \left(\frac{\partial v}{\partial t} \right)^T \right)}_{\text{want to be zero!}} \frac{\partial u}{\partial p} \right] dt \, .$$

If we could now set the $(\quad)$ term to zero, then the unwanted $\frac{\partial u}{\partial p}$ would vanish from the integral calculation in G'. We can accomplish this by *choosing* $v(t)$ (which could be *anything* up to now) to satisfy the **"adjoint" ODE**:

$$\boxed{\frac{\partial v}{\partial t} = \left(\frac{\partial g}{\partial u} \right)^T \quad \left(\frac{\partial f}{\partial u} \right)^T v}.$$

What initial condition should we choose for $v(t)$? Well, we can use this choice to get rid of the boundary term we obtained above from integration by parts:

$$v^T \frac{\partial u}{\partial p} \Big|_0^T = v(T)^T \underbrace{\frac{\partial u}{\partial p} \Big|_T}_{\text{unknown}} \quad v(0)^T \underbrace{\frac{\partial u_0}{\partial p}}_{\text{known}} .$$

[6] This "left-to-right" picture can be made very explicit if we imagine discretizing the ODE into a recurrence, e.g. via Euler's method for an arbitrarily small Δt, as described in the MIT course notes *Adjoint methods and sensitivity analysis for recurrence relations* by S. G. Johnson (2011).

Here, the unknown $\left.\frac{\partial u}{\partial p}\right|_T$ term is a problem—to compute that, we would be forced to go back to integrating our big $\frac{\partial u}{\partial p}$ ODE from forward mode. The other term is okay: since the initial condition u_0 is always given, we should know its dependence on p explicitly (and we will simply have $\frac{\partial u_0}{\partial p} = 0$ in the common case where the initial conditions don't depend on p). To eliminate the $\left.\frac{\partial u}{\partial p}\right|_T$ term, therefore, we make the choice

$$\boxed{v(T) = 0}.$$

Instead of an *initial* condition, our adjoint ODE has a **final condition**. That's no problem for a numerical solver: it just means that the **adjoint ODE is integrated *backwards* in time**, starting from $t = T$ and working down to $t = 0$. Once we have solved the adjoint ODE for $v(t)$, we can plug it into our equation for G' to obtain our gradient by a simple integral:

$$\nabla_p G = (G')^T = \left(\frac{\partial u_0}{\partial p}\right)^T v(0) + \int_0^T \left[\left(\frac{\partial g}{\partial p}\right)^T - \left(\frac{\partial f}{\partial p}\right)^T v\right] dt \, .$$

(If you want to be fancy, you can compute this $\int_0^T$ simultaneously with v itself, by augmenting the adjoint ODE with an additional set of unknowns and equations representing the G' integrand. But that's mainly just a computational convenience and doesn't change anything fundamental about the process.)

The only remaining annoyance is that the adjoint ODE depends on $u(p,t)$ for all $t \in [0,T]$. Normally, if we are solving the "forward" ODE for $u(p,t)$ numerically, we can "march" the solution u forwards in time and only store the solution at a few of the most recent timesteps. Since the adjoint ODE starts at $t = T$, however, we can only start integrating v after we have completed the calculation of u. This requires us to save essentially *all* of our previously computed $u(p,t)$ values, so that we can evaluate u at arbitrary times $t \in [0,T]$ during the integration of v (and G'). This can require a lot of computer memory if u is large (e.g. it could represent *millions* of grid points from a spatially discretized PDE, such as in a heat-diffusion problem) and many timesteps t were required. To ameliorate this challenge, a variety of strategies have been employed, typically centered around "checkpointing" techniques in which u is only saved at a subset of times t, and its value at other times is obtained during the v integration by *re-computing* u as needed (numerically integrating the ODE starting at the closest "checkpoint" time). A detailed discussion of such techniques lies outside the scope of these notes, however.

11.3 Example

Let us illustrate the above techniques with a simple example. Suppose that we are integrating the scalar ODE

$$\frac{\partial u}{\partial t} = f(u,p,t) = p_1 + p_2 u + p_3 u^2 = p^T \begin{pmatrix} 1 \\ u \\ u^2 \end{pmatrix}$$

for an initial condition $u(p,0) = u_0 = 0$ and three parameters $p \in \mathbb{R}^3$. (This is probably simple enough to solve in closed form, but we won't bother with that here.) We will also consider the scalar function

$$G(p) = \int_0^T \underbrace{[u(p,t) - u_*(t)]^2}_{g(u,p,t)} dt$$

that (for example) we may want to minimize for some given $u_*(t)$ (e.g. experimental data or some given formula like $u_* = t^3$), so we are hoping to compute $\nabla_p G$.

11.3.1 Forward mode

The Jacobian matrix $\frac{\partial u}{\partial p} = \begin{pmatrix} \frac{\partial u}{\partial p_1} & \frac{\partial u}{\partial p_2} & \frac{\partial u}{\partial p_3} \end{pmatrix}$ is simply a row vector, and satisfies our "forward-mode" ODE:

$$\frac{\partial}{\partial t}\left(\frac{\partial u}{\partial p}\right) = \frac{\partial f}{\partial u}\frac{\partial u}{\partial p} + \frac{\partial f}{\partial p} = (p_2 + 2p_3 u)\frac{\partial u}{\partial p} + \begin{pmatrix} 1 & u & u^2 \end{pmatrix}$$

for the initial condition $\left.\frac{\partial u}{\partial p}\right|_{t=0} = \frac{\partial u_0}{\partial p} = 0$. This is an inhomogeneous system of three coupled *linear* ODEs, which might look more conventional if we simply transpose both sides:

$$\underbrace{\frac{\partial}{\partial t}\begin{pmatrix} \frac{\partial u}{\partial p_1} \\ \frac{\partial u}{\partial p_2} \\ \frac{\partial u}{\partial p_3} \end{pmatrix}}_{(\partial u/\partial p)^T} = (p_2 + 2p_3 u)\begin{pmatrix} \frac{\partial u}{\partial p_1} \\ \frac{\partial u}{\partial p_2} \\ \frac{\partial u}{\partial p_3} \end{pmatrix} + \begin{pmatrix} 1 \\ u \\ u^2 \end{pmatrix}.$$

The fact that this depends on our "forward" solution $u(p,t)$ makes it not so easy to solve by hand, but a computer can solve it numerically with no difficulty. We can then plug this into the chain rule for G:

$$\nabla_p G = 2\int_0^T \left[u(p,t) \quad u\ (t)\right]\frac{\partial u}{\partial p}^T dt$$

(again, an integral that a computer could evaluate numerically).

11.3.2 Reverse mode

In reverse mode, we have an adjoint solution $v(t) \in \mathbb{R}$ (the same shape as u) which solves our adjoint equation

$$\frac{\partial v}{dt} = \left(\frac{\partial g}{\partial u}\right)^T \quad \left(\frac{\partial f}{\partial u}\right)^T v = 2\left[u(p,t) \quad u\ (t)\right] \quad (p_2 + 2p_3 u)\, v$$

with a *final* condition $v(T) = 0$. Again, a computer can solve this numerically without difficulty (given the numerical "forward" solution u) to find $v(t)$ for $t \in [0,T]$. Finally, our gradient is the integrated product:

$$\nabla_p G = \int_0^T \begin{pmatrix} 1 \\ u \\ u^2 \end{pmatrix} v\, dt\,.$$

11.4 Further reading

A classic reference on reverse/adjoint differentiation of ODEs (and generalizations thereof), using notation similar to that used today (except that the adjoint solution v is denoted $\lambda(t)$, in homage to Lagrange multipliers), is Cao et al. (2003) (https://doi.org/10.1137/S106482750138063). See also the SciMLSensitivity.jl package (https://github.com/SciML/SciMLSensitivity.jl) for sensitivity analysis with Chris Rackauckas's amazing DifferentialEquations.jl software suite for numerical solution of ODEs in Julia. There is a nice 2021 YouTube lecture on adjoint sensitivity of ODEs (https://youtu.be/k6s2G5MZv-I), again using a similar notation. A discrete version of this process arises for recurrence relations, in which case one obtains a reverse-order "adjoint" recurrence relation as described in MIT course notes by S. G. Johnson (https://math.mit.edu/~stevenj/18.336/recurrence2.pdf).

The differentiation methods in this chapter (e.g. for $\partial u/\partial p$ or $\nabla_p G$) are derived assuming that the ODEs are solved exactly: given the exact ODE for u, we derived an exact ODE for the derivative. On a computer, you will solve these forward and adjoint ODEs approximately, and in consequence the resulting derivatives will only be

approximately correct (to the tolerance specified by your ODE solver). This is known as a **differentiate-then-discretize** approach, which has the advantage of simplicity (it is independent of the numerical solution scheme) at the expense of slight inaccuracy (your approximate derivative will not exactly predict the first-order change in your approximate solution u). The alternative is a **discretize-then-differentiate** approach, in which you first approximate ("discretize") your ODE into a discrete-time recurrence formula, and then *exactly* differentiate the recurrence. This has the advantage of exactly differentiating your approximate solution, at the expense of complexity (the derivation is specific to your discretization scheme). Various authors discuss these tradeoffs and their implications, e.g. in chapter 4 of M. D. Gunzburger's *Perspectives in Flow Control and Optimization* (2002) or in papers like Jensen et al. (2014).

12 Calculus of Variations

In this lecture, we will apply our derivative machinery to a new type of input: neither scalars, nor column vectors, nor matrices, but rather the **inputs will be functions** $u(x)$, which form a perfectly good vector space (and can even have norms and inner products).[7] It turns out that there are lots of amazing applications for differentiating with respect to *functions*, and the resulting techniques are sometimes called the "calculus of variations" and/or "Frechét" derivatives.

12.1 Functionals: Mapping functions to scalars

Example 34

For example, consider functions $u(x)$ that map $x \in [0,1]$! $u(x) \in \mathbb{R}$. We may then define the function f:

$$f(u) = \int_0^1 \sin(u(x))\,\mathrm{d}x.$$

Such a function, mapping an input *function* u to an output *number*, is sometimes called a "functional." What is f' or ∇f in this case?

Recall that, given any function f, we always define the derivative as a linear operator $f'(u)$ via the equation:

$$df = f(u+du) \quad f(u) = f'(u)[du]\,,$$

where now du denotes an arbitrary "small-valued" *function* $du(x)$ that represents a small change in $u(x)$, as depicted in Fig. **??** for the analogous case of a non-infinitesimal $\delta u(x)$. Here, we may compute this via linearization of the integrand:

$$\begin{aligned} df &= f(u+du) \quad f(u) \\ &= \int_0^1 \sin(u(x)+du(x)) \quad \sin(u(x))\,dx \\ &= \int_0^1 \cos(u(x))\,du(x)\,dx = f'(u)[du]\,, \end{aligned}$$

where in the last step we took $du(x)$ to be arbitrarily small[8] so that we could linearize $\sin(u+du)$ to first-order in $du(x)$. That's it, we have our derivative $f'(u)$ as a perfectly good linear operation acting on du!

12.2 Inner products of functions

In order to define a gradient ∇f when studying such "functionals" (maps from functions to $\mathbb{R}$), it is natural to ask if there is an inner product on the input space. In fact, there are perfectly good ways to define inner products of functions! Given functions $u(x), v(x)$ defined on $x \in [0,1]$, we could define a "Euclidean" inner product:

$$\langle u, v \rangle = \int_0^1 u(x)v(x)\,\mathrm{d}x.$$

[7] Being fully mathematically rigorous with vector spaces of functions requires a lot of tedious care in specifying a well-behaved set of functions, inserting annoying caveats about functions that di□er only at isolated points, and so forth. In this lecture, we will mostly ignore such technicalities—we will implicitly assume that our functions are integrable, di□erentiable, etcetera, as needed. The subject of *functional analysis* exists to treat such matters with more care.

[8] Technically, it only needs to be small "almost everywhere" since jumps that occur only at isolated points don't a□ect the integral.

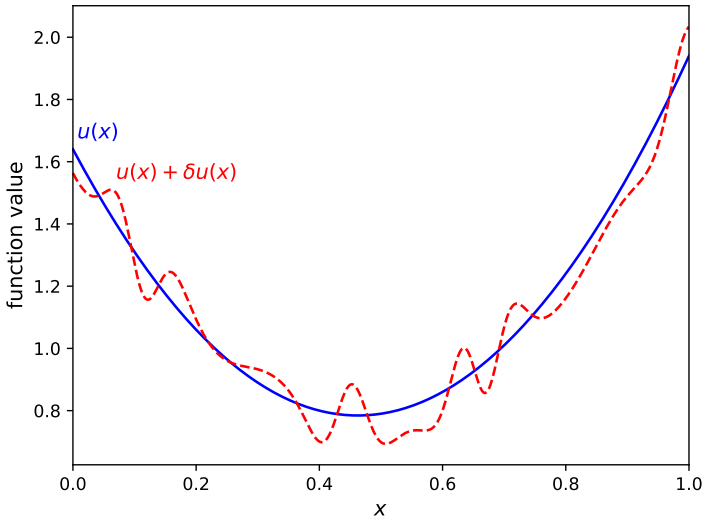


Figure 11: If our $f(u)$'s inputs u are *functions* $u(x)$ (e.g., mapping $[0,1] \to \mathbb{R}$), then the essence of differentiation is linearizing f for small perturbations $\delta u(x)$ that are themselves functions, in the limit where $\delta u(x)$ becomes arbitrarily small. Here, we show an example of a $u(x)$ and a perturbation $u(x) + \delta u(x)$.

Notice that this implies

$$\|u\| := \sqrt{\langle u, u \rangle} = \sqrt{\int_0^1 u(x)^2}.$$

Recall that the gradient ∇f is *defined* as whatever we take the inner product of du with to obtain df. Therefore, we obtain the gradient as follows:

$$df = f'(u)[du] = \int_0^1 \cos(u(x))\, du(x)\, dx = \langle \nabla f, du \rangle \implies \nabla f = \cos(u(x)).$$

The gradient ∇f is just another function, $\cos(u(x))$! As usual, ∇f has the same "shape" as u.

12.3 Example: Minimizing arc length

We now consider a more tricky example with an intuitive geometric interpretation.

Example 35

Let u be a differentiable function on $[0,1]$ and consider the functional

$$f(u) = \int_0^1 \sqrt{1 + u'(x)^2}\, dx.$$

Solve for ∇f when $u(0) = u(1) = 0$.

Geometrically, you learned in first-year calculus that this is simply the **length of the curve** $u(x)$ from $x = 0$ to $x = 1$. To differentiate this, first notice that ordinary single-variable calculus gives us the linearization

$$d\left(\sqrt{1+v^2}\right) = \sqrt{1 + (v+dv)^2} \quad \sqrt{1+v^2} = \left(\sqrt{1+v^2}\right)' dv = \frac{v}{\sqrt{1+v^2}} dv.$$

Therefore,

$$\begin{aligned} df &= f(u+du) \quad f(u) \\ &= \int_0^1 \left(\sqrt{1+(u+du)'^2} \quad \sqrt{1+u'^2}\right) dx \\ &= \int_0^1 \frac{u'}{\sqrt{1+u'^2}}\, du'dx. \end{aligned}$$

However, this is a linear operator on du' and not (directly) on du. Abstractly, this is fine, because du' is itself a linear operation on du, so we have $f'(u)[du]$ as the composition of two linear operations. However, it is more revealing to rewrite it explicitly in terms of du, for example in order to define ∇f. To accomplish this, we can apply *integration by parts* to obtain

$$f'(u)[du] = \int_0^1 \frac{u'}{\sqrt{1+u'^2}}\, du'dx = \left.\frac{u'}{\sqrt{1+u'^2}} du\right|_0^1 \quad \int_0^1 \left(\frac{u'}{\sqrt{1+u'^2}}\right)' du\, dx\,.$$

Notice that up until now we did not need utilize the "boundary conditions" $u(0) = u(1) = 0$ for this calculation. However, if we want to restrict ourselves to such functions $u(x)$, then our perturbation du cannot change the endpoint values, i.e. we must have $du(0) = du(1) = 0$. (Geometrically, suppose that we want find the u that minimizes arc length between $(0,0)$ and $(1,0)$, so that we need to fix the endpoints.) This implies that the boundary term in the above equation is zero. Furthermore, note that the u that minimizes the functional f has the property that $\nabla f|_u = 0$. Hence, we have that

$$df = \int_0^1 \underbrace{\left(\frac{u'}{\sqrt{1+u'^2}}\right)'}_{\nabla f} du\, dx = \langle \nabla f, du \rangle\,.$$

Therefore, for a u that minimizes the functional f (the *shortest curve*), we must have the following result:

$$\begin{aligned} 0 = \nabla f &= \left(\frac{u'}{\sqrt{1+u'^2}}\right)' \\ &= \frac{u''\sqrt{1+u'^2} \quad u' \frac{u''u'}{\sqrt{1+u'^2}}}{1+u'^2} \\ &= \frac{u''(1+u'^2) \quad u''u'^2}{(1+u'^2)^{3/2}} \\ &= \frac{u''}{(1+u'^2)^{3/2}}. \end{aligned}$$

Hence, $\nabla f = 0 \implies u''(x) = 0 \implies u(x) = ax + b$ for constants a, b; and for these boundary conditions $a = b = 0$. In other words, u is a straight line! Thus, we have recovered the familiar result that that straight lines in $\mathbb{R}^2$ are the shortest curves between two points!

12.4 Euler–Lagrange equations

This style of calculation is part of the subject known as the **calculus of variations**. Of course, the final answer in this example above (a straight line) may have been obvious, but a similar approach can be applied to many more interesting problems. We can generalize the approach as follows:

Example 36

Let $f(u) = \int_a^b F(u, u', x)\, dx$ where u is a differentiable function on $[a, b]$. Suppose the endpoints of u are fixed (i.e. its values at $x = a$ and $x = b$ are constants), and calculate df and ∇f.

This calculation

$$\begin{aligned} df &= f(u + du) \quad f(u) \\ &= \int_a^b \left(\frac{\partial F}{\partial u} du + \frac{\partial F}{\partial u'} du' \right) dx \\ &= \underbrace{\frac{\partial F}{\partial u'} du \Big|_a^b}_{=0} + \int_a^b \left(\frac{\partial F}{\partial u} \quad \left(\frac{\partial F}{\partial u'} \right)' \right) du\, dx \end{aligned}$$

where we used the fact that $du = 0$ at a or b if the endpoints $u(a)$ and $u(b)$ are fixed. Hence,

$$\nabla f = \frac{\partial F}{\partial u} \quad \left(\frac{\partial F}{\partial u'} \right)',$$

which equals zero at extremum. Notice that this gives rise to a 2nd-order differential equation in u, known as the Euler–Lagrange equations!

There are many wonderful applications of this idea. For example, search online for information about the "brachistochrone problem" and/or the "principle of least action". A classic textbook on the topic is *Calculus of Variations* by Gelfand and Fomin.

13 Derivatives of Random Functions

These notes are from a guest lecture by Gaurav Arya in IAP 2023.

13.1 Introduction

In this class, we've learned how to take derivatives of all sorts of crazy functions. Recall one of our first examples:

$$f(A) = A^2, \tag{5}$$

where A is a matrix. To differentiate this function, we had to go back to the drawing board, and ask:

Question 37. *If we perturb the input slightly, how does the output change?*

To this end, we wrote down something like:

$$\delta f = (A + \delta A)^2 \quad A^2 = A(\delta A) + (\delta A)A + \underbrace{(\delta A)^2}_{\text{neglected}}. \tag{6}$$

We called δf and δA *differentials* in the limit where δA became arbitrarily small. We then had to ask:

Question 38. *What terms in the differential can we neglect?*

We decided that $(\delta A)^2$ should be neglected, justifying this by the fact that $(\delta A)^2$ is "higher-order". We were left with the derivative operator $\delta A \mapsto A(\delta A) + (\delta A)A$: the best possible *linear* approximation to f in a neighbourhood of A. At a high level, the main challenge here was dealing with complicated input and output spaces: f was matrix-valued, and also matrix-accepting. We had to ask ourselves: in this case, what should the notion of a derivative even mean?

In this lecture, we will face a similar challenge, but with an even weirder type of function. This time, the output of our function will be *random*. Now, we need to revisit the same questions. If the output is random, how can we describe its response to a change in the input? And how can we form a useful notion of derivative?

13.2 Stochastic programs

More precisely, we will consider random, or *stochastic*, functions X with real input $p \in \mathbb{R}$ and real-valued random-variable output. As a map, we can write X as

$$p \mapsto X(p), \tag{7}$$

where $X(p)$ is a random variable. (To keep things simple, we'll take $p \in \mathbb{R}$ and $X(p) \in \mathbb{R}$ in this chapter, though of course they could be generalized to other vector spaces as in the other chapters. For now, the randomness is complication enough to deal with.)

The idea is that we can only *sample* from $X(p)$, according to some distribution of numbers with probabilities that depend upon p. One simple example would be sampling real numbers uniformly (equal probabilities) from the interval $[0, p]$. As a more complicated example, suppose $X(p)$ follows the *exponential distribution* with scale p, corresponding to randomly sampled real numbers $x \geq 0$ whose probability decreases proportional to $e^{-x/p}$. This can be denoted $X(p) \sim \text{Exp}(p)$, and implemented in Julia by:

```
julia> using Distributions
```

```
julia> sample_X(p) = rand(Exponential(p))
sample_X (generic function with 1 method)
```

We can take a few samples:

```
julia> sample_X(10.0)
1.7849785709142214

julia> sample_X(10.0)
4.435847397169775

julia> sample_X(10.0)
0.6823343897949835

julia> mean(sample_X(10.0) for i = 1:10^9) # mean = p
9.999930348291866
```

If our program gives a different output each time, what could a useful notion of derivative be? Before we try to answer this, let's ask *why* we might want to take a derivative. The answer is that we may be very interested in *statistical properties* of random functions, i.e. values that can be expressed using *averages*. Even if a function is stochastic, its *average* ("expected value") can be a deterministic function of its parameters that has a conventional derivative.

So, why not take the average *first*, and then take the ordinary derivative of this average? This simple approach works for very basic stochastic functions (e.g. the exponential distribution above has expected value p, with derivative 1), but runs into practical difficulties for more complicated distributions (as are commonly implemented by large computer programs working with random numbers).

Remark 39. *It is often much easier to produce an "unbiased estimate" $X(p)$ of a statistical quantity than to compute it exactly. (Here, an unbiased estimate means that $X(p)$ averages out to our statistical quantity of interest.)*

For example, in deep learning, the "variational autoencoder" (VAE) is a very common architecture that is inherently stochastic. It is easy to get a stochastic *unbiased estimate* of the loss function by running a random simulation $X(p)$: the loss function $L(p)$ is then the "average" value of $X(p)$, denoted by the *expected value* $\mathsf{E}[X(p)]$. However, computing the loss $L(p)$ exactly would require integrating over all possible outcomes, which usually impractical. Now, to train the VAE, we also need to differentiate $L(p)$, i.e. differentiate $\mathsf{E}[X(p)]$ with respect to p!

Perhaps more intuitive examples can be found in the physical sciences, where randomness may be baked into your model of a physical process. In this case, it's hard to get around the fact that you need to deal with stochasticity! For example, you may have two particles that interact with an *average* rate of r. But in reality, the times when these interactions actually occur follow a stochastic process. (In fact, the time until the first interaction might be exponentially distributed, with scale $1/r$.) And if you want to (e.g.) fit the parameters of your stochastic model to real-world data, it's once again very useful to have derivatives.

If we can't compute our statistical quantity of interest exactly, it seems unreasonable to assume we can compute its derivative exactly. However, we could hope to stochastically *estimate* its derivative. That is, if $X(p)$ represents the full program that produces an unbiased estimate of our statistical quantity, here's one property we'd definitely like our notion of derivative to have: we should be able to construct from it an unbiased gradient estimator[9] $X'(p)$

[9] For more discussion of these concepts, see (e.g.) the review article "Monte Carlo gradient estimation in machine learning" (2020) by Mohamed *et al.* (https://arxiv.org/abs/1906.10652).

satisfying

$$\mathsf{E}[X^{\emptyset}(p)] = \mathsf{E}[X(p)]^{\emptyset} = \frac{\partial \mathsf{E}[X(p)]}{\partial p}. \tag{8}$$

Of course, there are infinitely many such estimators. For example, given any estimator $X^{\emptyset}(p)$ we can add any other random variable that has zero average without changing the expectation value. But in practice there are two additional considerations: (1) we want $X^{\emptyset}(p)$ to be easy to compute/sample (about as easy as $X(p)$), and (2) we want the *variance* (the "spread") of $X^{\emptyset}(p)$ to be small enough that we don't need too many samples to estimate its average accurately (hopefully no worse than estimating $\mathsf{E}[X(p)]$).

13.3 Stochastic di□erentials and the reparameterization trick

Let's begin by answering our first question (Question 37): how does $X(p)$ respond to a change in p? Let us consider a specific p and write down a *stochastic di erential*, taking a small but non-infinitesimal δp to avoid thinking about infinitesimals for now:

$$\delta X(p) = X(p + \delta p) \quad X(p), \tag{9}$$

where δp represents an arbitrary small change in p. What sort of object is $\delta X(p)$?

Since we're subtracting two random variables, it ought to itself be a random variable. However, $\delta X(p)$ is still not fully specified! We have only specified the marginal distributions of $X(p)$ and $X(p+\delta p)$: to be able to subtract the two, we need to know their *joint distribution*.

One possibility is to treat $X(p)$ and $X(p + \delta p)$ as independent. This means that $\delta X(p)$ would be constructed as the difference of independent samples. Let's see how samples from $\delta X(p)$ would look like in this case!

```
julia> sample_X(p) = rand(Exponential(p))
sample_X (generic function with 1 method)

julia> sample_ X(p,  p) = sample_X(p +  p) - sample_X(p)
sample_ X (generic function with 1 method)

julia> p = 10;  p = 1e-5;

julia> sample_ X(p,  p)
-26.000938718875904

julia> sample_ X(p,  p)
-2.6157162001718092

julia> sample_ X(p,  p)
6.352622554495474

julia> sample_ X(p,  p)
-9.53215951927184

julia> sample_ X(p,  p)
1.2232268930932104
```

We can observe something a bit worrying: even for a very tiny δp (we chose $\delta p = 10^{\ -5}$), $\delta X(p)$ is still fairly large:

essentially as large as the original random variables. This is not good news if we want to construct a derivative from $\delta X(p)$: we would rather see its magnitude getting smaller and smaller with δp, like in the non-stochastic case. Computationally, this will make it very difficult to determine $\mathsf{E}[X(p)]'$ by averaging `sample_ X(p, p) / p` over many samples: we'll need a huge number of samples because the ***variance***, the "spread" of random values, is huge for small δp.

Let's try a different approach. It is natural to think of $X(p)$ for all p as forming a ***family*** of random variables, all defined on the same ***probability space***. A probability space, with some simplification, is a sample space Ω, with a probability distribution P defined on the sample space. From this point of view, each $X(p)$ can be expressed as a function Ω ! R. To sample from a particular $X(p)$, we can imagine drawing a random ω from Ω according to P, and then plugging this into $X(p)$, i.e. computing $X(p)(\omega)$. (Computationally, this is how most distributions are actually implemented: you start with a primitive pseudo-random number generator for a very simple distribution,[10] e.g. drawing values ω uniformly from $\Omega = [0, 1)$, and then you build other distributions on top of this by transforming ω somehow.) Intuitively, all of the "randomness" resides in the probability space, and crucially P does not depend on p: as p varies, $X(p)$ just becomes a different ***deterministic*** map on Ω.

The crux here is that all the $X(p)$ functions now depend on a shared source of randomness: the random draw of ω. This means that $X(p)$ and $X(p + \delta p)$ have a nontrivial joint distribution: what does it look like?

For concreteness, let's study our exponential random variable $X(p)$ Exp(p) from above. Using the "inversion sampling" parameterization, it is possible to choose Ω to be $[0, 1)$ and P to be the uniform distribution over Ω; for any distribution, we can construct $X(p)$ to be a corresponding nondecreasing function over Ω (given by the inverse of $X(p)$'s cumulative probability distribution). Applied to $X(p)$ Exp(p), the inversion method gives $X(p)(\omega) = p \log(1 \omega)$. This is implemented below, and is a theoretically equivalent way of sampling $X(p)$ compared with the opaque `rand(Exponential(p))` function we used above:

```
julia> sample_X2(p, !) = -p * log(1 - !)
sample_X2 (generic function with 1 method)

julia> # rand() samples a uniform random number in [0,1)
julia> sample_X2(p) = sample_X2(p, rand())
sample_X2 (generic function with 2 methods)

julia> sample_X2(10.0)
8.380816941818618

julia> sample_X2(10.0)
2.073939134369733

julia> sample_X2(10.0)
29.94586208847568

julia> sample_X2(10.0)
23.91658360124792
```

Okay, so what does our joint distribution look like? As shown in Figure 12, we can plot $X(p)$ and $X(p + \delta p)$ as

[10] Most computer hardware cannot generate numbers that are actually random, only numbers that *seem* random, called "pseudo-random" numbers. The design of these random-seeming numeric sequences is a subtle subject, steeped in number theory, with a long history of mistakes. A famous ironic quotation in this field is (Robert Coveyou, 1970): "Random number generation is too important to be left to chance."

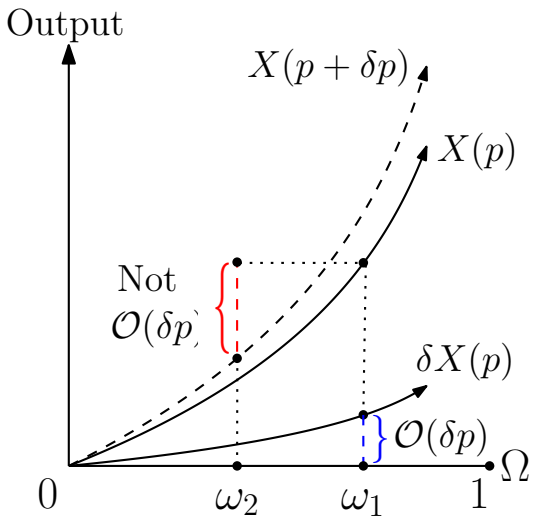


Figure 12: For $X(p)$ Exp(p) parameterized via the inversion method, we can write $X(p)$, $X(p+\delta p)$, and $\delta X(p)$ as functions from $\Omega = [0,1]$! R, defined on a probability space with P $=$ Unif$(0,1)$.

functions over Ω. To sample the two of them jointly, we use the *same* choice of ω: thus, $\delta X(p)$ can be formed by subtracting the two functions *pointwise* at each Ω. Ultimately, $\delta X(p)$ is itself a random variable over the same probability space, sampled in the same way: we pick a random ω according to P, and evaluate $\delta X(p)(\omega)$, using the function $\delta X(p)$ depicted above. Our first approach with independent samples is depicted in red in Figure 12, while our second approach is in blue. We can now see the flaw of the independent-samples approach: the $O(1)$-sized "noise" from the independent samples washes out the $O(\delta p)$-sized "signal".

What about our second question (Question 38): how can actually take the limit of δp ! 0 and compute the derivative? The idea is to differentiate $\delta X(p)$ at each fixed sample ω 2 Ω. In probability theory terms, we take the limit of random variables $\delta X(p)/\delta p$ as δp ! 0:

$$X'(p) = \lim_{\delta p \,!\, 0} \frac{\delta X(p)}{\delta p}. \tag{10}$$

For $X(p)$ Exp(p) parameterized via the inversion method, we get:

$$X'(p)(\omega) = \lim_{\delta p \,!\, 0} \frac{\delta p \log(1 \quad \omega)}{\delta p} = \quad \log(1 \quad \omega). \tag{11}$$

Once again, $X'(p)$ is a random variable over the same probability space. The claim is that $X'(p)$ is the notion of derivative we were looking for! Indeed, $X'(p)$ is itself in fact a valid gradient estimator:

$$\mathsf{E}[X'(p)] = \mathsf{E}\left[\lim_{\delta p \,!\, 0} \frac{\delta X(p)}{\delta p}\right] \stackrel{?}{=} \lim_{\delta p \,!\, 0} \frac{\mathsf{E}[\delta X(p)]}{\delta p} = \frac{\partial \mathsf{E}[X(p)]}{\partial p}. \tag{12}$$

Rigorously, one needs to justify the interchange of limit and expectation in the above. In this chapter, however, we will be content with a crude empirical justification:

```
julia> X'(p, ω) = -log(1 - ω)
X' (generic function with 1 method)

julia> X'(p) = X'(p, rand())
X' (generic function with 2 methods)
```

```
julia> mean(X^0(10.0) for i in 1:10000)
1.011689946421105
```

So $X^0(p)$ does indeed average to 1, which makes sense since the expectation of $\text{Exp}(p)$ is p, which has derivative 1 for any choice of p. However, the crux is that this notion of derivative also works for more complicated random variables that can be formed via *composition* of simple ones such as an exponential random variable. In fact, it turns out to obey the same chain rule as usual!

Let's demonstrate this. Using the dual numbers introduced in Chapter 10, we can differentiate the expectation of the square of a sample from an exponential distribution *without* having an analytic expression for this quantity. (The expression for X^0 we derived is already implemented as a dual-number rule in Julia by the `ForwardDiff.jl` package.) The primal and dual values of the outputted dual number are samples from the joint distribution of $(X(p), X^0(p))$.

```
julia> using Distributions, ForwardDiff: Dual

julia> sample_X(p) = rand(Exponential(p))^2
sample_X (generic function with 1 method)

julia> sample_X(Dual(10.0, 1.0)) # sample a single dual number!
Dual{Nothing}(153.74964559529033,30.749929119058066)

julia> # obtain the derivative!
julia> mean(sample_X(Dual(10.0, 1.0)).partials[1] for i in 1:10000)
40.016569793650525
```

Using the "reparameterization trick" to form a gradient estimator, as we have done here, is a fairly old idea. It is also called the "pathwise" gradient estimator. Recently, it has become very popular in machine learning due to its use in VAEs [e.g. Kingma & Welling (2013): https://arxiv.org/abs/1312.6114], and lots of resources can be found online on it. Since composition simply works by the usual chain rule, it also works in reverse mode, and can differentiate functions far more complicated than the one above!

13.4 Handling discrete randomness

So far we have only considered a continuous random variable. Let's see how the picture changes for a discrete random variable! Let's take a simple Bernoulli variable $X(p)$ $\text{Ber}(p)$, which is 1 with probability p and 0 with probability 1 p.

```
julia> sample_X(p) = rand(Bernoulli(p))
sample_X (generic function with 1 method)

julia> p = 0.5
0.6

julia> sample_X( p) # produces false/true, equivalent to 0/1
true
```

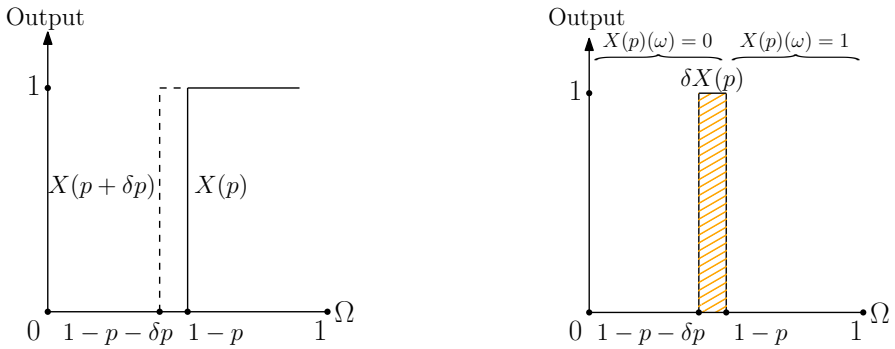


Figure 13: For $X(p)$ Ber(p) parameterized via the inversion method, plots of $X(p)$, $X(p+\delta p)$, and $\delta X(p)$ as functions $\Omega : [0,1]$ *!* R.

```
julia> sample_X( p)
false

julia> sample_X( p)
true
```

The parameterization of a Bernoulli variable is shown in Figure 2. Using the inversion method once again, the parameterization of a Bernoulli variable looks like a step function: for $\omega < 1$ p, $X(p)(\omega) = 0$, while for ω 1 p, $X(p)(\omega) = 1$.

Now, what happens when we perturb p? Let's imagine perturbing p by a positive amount δp. As shown in Figure 2, something qualitatively very different has happened here. At nearly every ω except a small region of probability δp, the output does not change. Thus, the quantity $X^0(p)$ we defined in the previous subsection (which, strictly speaking, was defined by an "almost-sure" limit that neglects regions of probability 0) is 0 at every ω: after all, for every ω, there exists small enough δp such that $\delta X(p)(\omega) = 0$.

However, there is certainly an important derivative contribution to consider here. The expectation of a Bernoulli is p, so we would expect the derivative to be 1: but $\mathsf{E}[X^0(p)] = \mathsf{E}[0] = 0$. What has gone wrong is that, although $\delta X(p)$ is 0 with tiny probability, the value of $\delta X(p)$ on this region of tiny probability is 1, which is *large*. In particular, it does not approach 0 as δp approaches 0. Thus, to develop a notion of derivative of $X(p)$, we need to somehow capture these large jumps with "infinitesimal" probability.

A recent (2022) publication (https://arxiv.org/abs/2210.08572) by the author of this chapter (Gaurav Arya), together with Frank Schäfer, Moritz Schauer, and Chris Rackauckas, worked to extend the above ideas to develop a notion of "stochastic derivative" for discrete randomness, implemented by a software package called `StochasticAD.jl` that performs automatic differentiation of such stochastic processes. It generalizes the idea of dual numbers to stochastic *triples*, which include a third component to capture exactly these large jumps. For example, the stochastic triple of a Bernoulli variable might look like:

```
julia> using StochasticAD, Distributions
julia> f(p) = rand(Bernoulli(p)) # 1 with probability p, 0 otherwise
julia> stochastic_triple(f, 0.5) # Feeds 0.5 +  p into f
StochasticTriple of Int64:
0 + 0  + (1 with probability 2.0 )
```

Here, δp is denoted by , imagined to be an "infinitesimal unit", so that the above triple indicates a flip from 0 to 1 with probability that has derivative 2.

However, many aspects of these problems are still difficult, and there are a lot of improvements awaiting future developments! If you're interested in reading more, you may be interested in the paper and our package linked above, as well as the 2020 review article by Mohamed *et al.* (https://arxiv.org/abs/1906.10652), which is a great survey of the field of gradient estimation in general.

At the end of class, we considered a differentiable random walk example with `StochasticAD.jl`. Here it is!

```
julia> using Distributions, StochasticAD

julia> function X(p)
           n = 0
           for i in 1:100
               n += rand(Bernoulli(p * (1 - (n+i)/200)))
           end
           return n
       end
X (generic function with 1 method)

julia> mean(X(0.5) for _ in 1:10000) # calculate E[X(p)] at p = 0.5
32.6956

julia> st = stochastic_triple(X, 0.5) # sample a single stochastic triple at p = 0.5
StochasticTriple of Int64:
32 + 0 p + (1 with probability 74.17635818221052 p)

julia> derivative_contribution(st) # derivative estimate produced by this triple
74.17635818221052

julia> # compute d/dp of E[X(p)] by taking many samples
julia> mean(derivative_contribution(stochastic_triple(f, 0.5)) for i in 1:10000)
56.65142976168479
```

14 Second Derivatives, Bilinear Forms, and Hessian Matrices

Recall, as we have been using throughout this class so far, that we defined the derivative of a function f by a linearization of its change df for a small change dx in the input:

$$df = f(x + dx) \quad f(x) = f'(x)[dx].$$

If we similarly consider the second derivative, we would obtain the following:

$$df' = f'(x + dx') \quad f'(x) = f''(x)[dx'].$$

(Notation: dx' is not some kind of derivative of dx; the prime simply denotes a *different* arbitrary small change in x.) Here, however, f' is a linear operator, so its change df' must *also* be a linear operator that we can act on an arbitrary dx, in the form:

$$f''(x)[dx'][dx] := f''(x)[dx', dx]\,,$$

where we combine the two brackets for brevity. This implies that $f''(x)$ is a *bilinear form*: acting on **two** vectors linear in both individually.

More precisely, we have the following.

Definition 40 (Bilinear Form)

Let U, V, W be a vector spaces, not necessarily the same. Then, a bilinear form is a map $B : U \quad V \; ! \; W$, mapping a u 2 U and v 2 V to $B[u, v]$ 2 W, such that we have linearity in both arguments:

$$\begin{cases} B[u, \alpha v_1 + \beta v_2] = \alpha B[u, v_1] + \beta B[u, v_2] \\ B[\alpha u_1 + \beta u_2, v] = \alpha B[u_1, v] + \beta B[u_2, v] \end{cases}$$

for any scalars α, β,

Note that in general, even if $U = V$ (the two inputs u, v are the "same type" of vector) we may have $B[u, v]$ 6= $B[v, u]$, but in the case of f'' we have something very special that happens. In particular, we can show that $f''(x)$ is a *symmetric bilinear form*, meaning

$$f''(x)[dx', dx] = f''(x)[dx, dx']$$

for any dx and dx'. Why? Because, applying the definition of f'' as giving the change in f' from dx', and then the definition of f' as giving the change in f from dx, we cam re-order terms to obtain:

$$\begin{aligned} f''(x)[dx', dx] &= f'(x + dx')[dx] \quad f'(x)[dx] \\ &= \left(f(x + \underbrace{dx' + dx}_{=dx+dx'}) \quad f(x + dx') \right) \quad (f(x + dx) \quad f(x)) \\ &= (f(x + dx + dx') \quad f(x + dx)) \quad (f(x + dx') \quad f(x)) \\ &= f'(x + dx)[dx'] \quad f'(x)[dx'] \\ &= f''(x)[dx, dx']. \end{aligned}$$

(The basic reason why this works is that the "+" operation is always *commutative* for any vector space.)

Example 41

Let's consider a familiar example from multivariable calculus, $f : \mathbb{R}^n \to \mathbb{R}$. That is, $f(x)$ is a scalar-valued function of a column vector $x \in \mathbb{R}^n$. What is f''?

Recall that

$$f'(x) = (\nabla f)^T \implies f'(x)[dx] = \text{scalar } df = (\nabla f)^T dx.$$

Similarly,

$$\begin{aligned} f''(x)[dx', dx] &= \text{scalar from two vectors, linear in both} \\ &= dx'^T H dx\,, \end{aligned}$$

since $dx'^T H dx$, where H is an $n \times n$ matrix called the **Hessian matrix**, is the most general possible bilinear form mapping two vectors to a scalar. Moreover, since it is a symmetric bilinear form from above, we must have:

$$\begin{aligned} f''(x)[dx', dx] &= dx'^T H dx \\ &= f''(x)[dx, dx'] = dx^T H dx' = (dx'^T H dx)^T \\ &= dx^T H^T dx'. \end{aligned}$$

This implies that $H = H^T$: the Hessian matrix is symmetric.

Explicitly, in terms of coordinates, we have

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} & \cdots & \frac{\partial f}{\partial x_n} \end{bmatrix}^T.$$

and

$$\begin{aligned} f''(x)[dx', dx] &= f'(x + dx')[dx] - f'(x)[dx] \\ &= \left(\nabla f\big|_{x+dx'} - \nabla f\big|_x \right)^T dx \\ &= dx'^T H dx = dx'^T H^T dx\,. \end{aligned}$$

Therefore, by transposing, we have,

$$\begin{aligned} H dx' = d(\nabla f) &= \nabla f\big|_{x+dx'} - \nabla f\big|_x \\ &= \left[\left(\nabla \tfrac{\partial f}{\partial x_1} \right)^T dx' \quad \left(\nabla \tfrac{\partial f}{\partial x_2} \right)^T dx' \quad \cdots \quad \left(\nabla \tfrac{\partial f}{\partial x_n} \right)^T dx' \right]^T \\ &= \left[\left(\nabla \tfrac{\partial f}{\partial x_1} \right)^T \quad \left(\nabla \tfrac{\partial f}{\partial x_2} \right)^T \quad \cdots \quad \left(\nabla \tfrac{\partial f}{\partial x_n} \right)^T \right]^T dx' \\ &= \underbrace{\begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_n} \end{bmatrix}}_{H} dx'\,. \end{aligned}$$

so that the Hessian matrix H has entries

$$H_{i,j} = \frac{\partial^2 f}{\partial x_i \partial x_j} = (H^T)_{i,j} = H_{j,i} = \frac{\partial^2 f}{\partial x_j \partial x_i}\,.$$

The fact that you can take partial derivatives in either order is a familiar fact from multivariable calculus (sometimes called the "symmetry of mixed derivatives" or "equality of mixed partials"), which we obtain here as a general consequence of f'' being a symmetric bilinear form.

Let's consider another example.

Example 42
Let $f(x) = x^T Ax$ for $x \in \mathbb{R}^n$ and A an $n \times n$ matrix. As above, $f(x) \in \mathbb{R}$ (scalar outputs). Compute f''.

The computation is fairly straight forward. Firstly, we have that

$$f' = (\nabla f)^T = x^T(A + A^T).$$

This implies that $\nabla f = (A + A^T)x$. Hence, $f'' = H = A + A^T$. Furthermore, note that this implies

$$\begin{aligned} f(x) &= x^T Ax \\ &= (x^T Ax)^T \\ &= x^T A^T x \\ &= \frac{1}{2} x^T (A + A^T)x \\ &= \frac{1}{2} x^T Hx = \frac{1}{2} f''[x, x]. \end{aligned}$$

Example 43
Let $f(A) = \det A$ for A an $n \times n$ matrix. Compute $f''(A)$ in terms of dA and dA'.

Firstly, from lecture 3, we have that

$$f'(A)[dA] = df = \det(A) \operatorname{tr}(A^{-1} dA).$$

Now, we have that

$$\begin{aligned} f''(A)[dA, dA'] &= d'(\det A \operatorname{tr}(A^{-1} dA)) \\ &= \det A \operatorname{tr}(A^{-1} dA') \operatorname{tr}(A^{-1} dA) \quad \det A \operatorname{tr}(A^{-1} dA' A^{-1} dA) \\ &= f''(A)[dA', dA] \end{aligned}$$

where the last line (symmetry) can be derivated explicitly by the cyclic property of the trace (although of course it must be true for any f''). Although f'' here is a perfectly good bilinear form acting on matrices dA, dA', it is not very natural to express f'' as a "Hessian matrix." (We could do so if we really wanted to by the "vectorization" approach of Sec. 4.)

14.1 Quadratic approximation

So how do we ultimately think about f''? We know that f' is the linearization/linear approximation of $f(x)$, i.e.

$$f(x + \delta x) = f(x) + f'(x)[\delta x] + o(\|\delta x\|).$$

Now, we can use f'' to form a *quadratic approximation* of $f(x)$. In particular, one can show that

$$f(x+\delta x) = f(x) + f'(x)[\delta x] + \frac{1}{2}f''(x)[\delta x, \delta x] + o(\|\delta x\|^2).$$

Note that the $\frac{1}{2}$ factor is just as in the Taylor series. To derive this, simply plug in the quadratic approximation into

$$f''(x)[dx, dx'] = f(x + dx + dx') + f(x) \quad f(x + dx) \quad f(x + dx').$$

and check that the right-hand side reproduces $f''(x)$.

14.2 Hessians and optimization

Many important applications of second derivatives, Hessians, and quadratic approximations arise in optimization: minimization (or maximization) of functions $f(x)$.

14.2.1 Sequential quadratic programming

When searching for a local minimum (or maximum) of a complicated function $f(x)$, a common procedure is to approximate $f(x + \delta x)$ by a simpler "model" function for small δx, and then to optimize this model to obtain a potential optimization step. For example, approximating $f(x + \delta x) \quad f(x) + f'(x)[\delta x]$ (a "linear," or rather affine, model) leads to gradient descent and related algorithms. A better approximation for $f(x + \delta x)$ will often lead to faster-converging algorithms, and so a natural idea is to exploit the *second* derivative f'' to make a quadratic model, as above, and accelerate optimization.

In essence, one proceeds by optimizing a sequence of quadratic approximations for f, and several algorithms of this form are sometimes called "sequential quadratic programming" (SQP). A quadratic program (QP) refers to optimization of a quadratic function subject to affine constraints, for which many efficient algorithms exist. SQP algorithms take a sequence of steps in which one makes a quadratic approximation of the function being minimized (via the Hessian) and an affine approximation of any constraints (via the gradient), solving the resulting QP (or QP-like) problem to obtain a step.

For an unconstrained optimization, minimizing $f(x)$ corresponds to finding a root of the derivative $f' = 0$ (i.e., $\nabla f = 0$), and a *quadratic* approximation for f yields a *linear* (affine) approximation $f'(x+\delta x) \quad f'(x) + f''(x)[\delta x]$ for the derivative f'. (In $\mathbb{R}^n$, $\delta(\nabla f) \quad H\delta x$.) So, minimizing a quadratic model is effectively a *Newton step* $\delta x \quad H^{-1}\nabla f$ to find a root of ∇f via first-order approximation. Thus, optimization via quadratic approximations is often viewed as a form of Newton algorithm.

There are many technical details, beyond the scope of this course, that must be resolved in order to translate such high-level ideas into practical algorithms. For example, a quadratic model is only valid for small enough δx, so there must be some mechanism to limit the step size. A typical idea is a "trust region": optimize the model with the constraint that δx is sufficiently small, e.g. $\|\delta x\| \quad s$ (a spherical trust region), along with some rules to adaptively enlarge or shrink the trust-region size (s) depending on how well the model predicts δf. There are many variants of SQP/Newton-like algorithms depending on the choices made for these and other details.

14.2.2 Computing Hessians

In general, finding f'' or the Hessian is often very expensive computationally in higher dimensions. If $f(x) : \mathbb{R}^n \to \mathbb{R}$, then the Hessian, H, is an $n \quad n$ matrix, which can be huge if n is large—even storing H may be prohibitive, much less computing it. Instead of computing H explicitly, one can approximate the Hessian in various ways; in the context of optimization, approximate Hessians are found in "quasi-Newton" methods such as the famous "BFGS"

algorithm and its variants. One can also derive efficient methods to compute Hessian–vector products Hv without computing H explicitly, e.g. for use in Newton–Krylov methods.

14.2.3 Minima, maxima, and saddle points

Generalizing the rules you may recall from single- and multi-variable calculus, we can use the second derivative to determine whether an extremum is a minimum, maximum, or saddle point. Firstly, an extremum of a scalar function f is a point x_0 such that $f'(x_0) = 0$. That is,

$$f'(x_0)[\delta x] = 0$$

for *any* δx. Equivalently,

$$\nabla f\big|_{x_0} = f'(x_0)^T = 0.$$

Using our quadratic approximation around x_0, we then have that

$$f(x_0 + \delta x) = f(x_0) + \underbrace{f'(x_0)[\delta x]}_{=0} + \frac{1}{2} f''(x_0)[\delta x, \delta x] + o(\|\delta x\|^2).$$

The definition of a local minimum x_0 is that $f(x_0 + \delta x) > f(x_0)$ for any $\delta x \neq 0$ with $\|\delta x\|$ sufficiently small. To achieve this at a point where $f' = 0$, it is enough to have f'' be a positive-definite quadratic form:

$$f''(x_0)[\delta x, \delta x] > 0 \text{ for all } \delta x \neq 0 \quad \Leftarrow \quad \textbf{positive-definite } f''(x_0).$$

For example, for inputs $x \in \mathbb{R}^n$, so that f'' is a real-symmetric $n \times n$ Hessian matrix, $f''(x_0) = H(x_0) = H(x_0)^T$, this corresponds to the usual criteria for a positive-definite matrix:

$$f''(x_0)[\delta x, \delta x] = \delta x^T H(x_0) \delta x > 0 \text{ for all } \delta x \neq 0 \quad \Leftrightarrow \quad H(x_0) \text{ positive-definite} \quad \Leftrightarrow \quad \text{all eigenvalues of } H(x_0) > 0.$$

In first-year calculus, one often focuses in particular on the 2-dimensional case, where H is a 2×2 matrix. In the 2×2 case, there is a simple way to check the signs of the two eigenvalues of H, in order to check whether an extremum is a minimum or maximum: the eigenvalues are both positive if and only if $\det(H) > 0$ and $\mathrm{tr}(H) > 0$, since $\det(H) = \lambda_1 \lambda_2$ and $\mathrm{tr}(H) = \lambda_1 + \lambda_2$. In higher dimensions, however, one needs more complicated techniques to compute eigenvalues and/or check positive-definiteness, e.g. as discussed in MIT courses 18.06 (Linear Algebra) and/or 18.335 (Introduction to Numerical Methods). (In practice, one typically checks positive-definiteness by performing a form of Gaussian elimination, called a Cholesky factorization, and checking that the diagonal "pivot" elements are > 0, rather than by computing eigenvalues which are much more expensive.)

Similarly, we have that x_0 where $\nabla f = 0$ is a local *maximum* if f'' is negative-definite, or equivalantly if the eigenvalues of the Hessian are all negative. Additionally, x_0 is a *saddle* point if f'' is indefinite, i.e. the eigenvalues include both positive and negative values. However, cases where some eigenvalues are zero are more complicated to analyze; e.g. if the eigenvalues are all ≥ 0 but some are $= 0$, then whether the point is a minimum depends upon higher derivatives.

14.3 Further Reading

An important generalization of quadratic operations to arbitrary vector spaces come in the form of Bilinear forms. For example, we saw that the second derivative can be seen as a symmetric bilinear form. This is closely related to a quadratic form, which what we get by plugging in the same vector twice. For example, the $f''(x)[\delta x, \delta x]/2$ that

appears in quadratic approximations of $f(x + \delta x)$ is a quadratic form. The most familiar multivariate version of $f''(x)$ is the Hessian matrix, and Khan Academy has an elementary introduction to quadratic approximation.

Positive-definite Hessian matrices, or more generally definite quadratic forms f'', appear at extrema ($f' = 0$) of scalar-valued functions $f(x)$ that are local minima. There are a lot more formal treatments of the same idea, and conversely Khan Academy has the simple 2-variable version where you can check the sign of the 2 2 eignevalues just by looking at the determinant and a single entry (or the trace). There's a nice stackexchange discussion on why an ill-conditioned Hessian tends to make steepest descent converge slowly. Some Toronto course notes on the topic may also be useful.

Lastly, see for example these Stanford notes on sequential quadratic optimization using trust regions (Section 2.2), as well as the 18.335 notes on BFGS quasi-Newton methods. The fact that a quadratic optimization problem in a sphere has strong duality, and hence is efficiently solvable, is discussed in Section 5.2.4 of the *Convex Optimization* book. There has been a lot of work on automatic Hessian computation, but for large-scale problems you may only be able to compute Hessian–vector products efficiently in general, which are equivalent to a directional derivative of the gradient and can be used (for example) for Newton–Krylov methods.

15 Derivatives of Eigenproblems

15.1 Di□erentiating on the Unit Sphere

Geometrically, we know that velocity vectors (equivalently, tangents) on the sphere are orthogonal to the radii. Out differentials say this algebraically, since given $x \in S^n$ we have $x^T x = 1$, this implies that

$$2x^T dx = d(x^T x) = d(1) = 0.$$

In other words, at the point x on the sphere (a radius, if you will), dx, the linearization of the constraint of moving along the sphere satisfies $dx \perp x$. This is our first example where we have seen the infinitesimal perturbation dx being constrained.

15.1.1 Special Case: A Circle

Let us simply consider the unit circle in the plane where $x = (\cos\theta, \sin\theta)$ for some $\theta \in [0, 2\pi)$. Then,

$$x^T dx = (\cos\theta, \sin\theta) \quad (\quad \sin\theta, \cos\theta) = 0.$$

Here, we can think of x as "extrinsic" coordinates, in that it is a vector in $\mathbb{R}^2$. On the other hand, θ is an "intrinsic" coordinate, as every point on the circle is specified by one θ.

15.1.2 On the Sphere

You may remember that the rank-1 matrix xx^T, for any unit vector $x^T x = 1$, is a **projection matrix** (meaning that it is equal to its square and it is symmetric) which projects vectors onto their components in the direction of x. Correspondingly, $I \quad xx^T$ is also a projection matrix, but onto the directions *perpendicular* to x: geometrically, the matrix removes components in the x direction. In particular, if $x^T dx = 0$, then $(I \quad xx^T)dx = dx$. It follows that if $x^T dx = 0$ and A is a symmetric matrix, we have

$$\begin{aligned} d\left(\frac{1}{2}x^T Ax\right) &= (Ax)^T dx \\ &= x^T A(dx) \\ &= x^T A(I \quad xx^T)dx \\ &= ((I \quad xx^T)Ax)^T dx. \end{aligned}$$

In other words, $(I \quad xx^T)Ax$ is the gradient of $\frac{1}{2}x^T Ax$ *on the sphere.*

So what did we just do? To obtain the gradient on the sphere, we needed (i) a linearization of the function that is correct on tangents, and (ii) a direction that *is* tangent (i.e. satisfies the linearized constraint). Using this, we obtain the gradient of a general scalar function on the sphere:

Theorem 44

Given $f : S^n \to \mathbb{R}$, we have

$$df = g(x)^T dx = ((I \quad xx^T)g(x))^T dx.$$

The proof of this is precisely the same as we did before for $f(x) = \frac{1}{2}x^T Ax$.

15.2 Di□erentiating on Orthogonal Matrices

Let Q be an orthogonal matrix. Then, computationally (as is done in the Julia notebook), one can see that $Q^T dQ$ is an anti-symmetric matrix (sometimes called skew-symmetric).

Definition 45
A matrix M is anti-symmetric if $M = \quad M^T$. Note that all anti-symmetric matrices thus have zeroes on their diagonals.

In fact, we can prove that $Q^T dQ$ is anti-symmetric.

Theorem 46
Given Q is an orthogonal matrix, we have that $Q^T dQ$ is anti-symmetric.

Proof. The constraint of being orthogonal implies that $Q^T Q = I$. Differentiating this equation, we obtain

$$Q^T dQ + dQ^T Q = 0 \ =) \ \ (Q^T dQ) = \quad (Q^T dQ)^T.$$

This is precisely the definition of being anti-symmetric. □

Before we move on, we may ask what the dimension of the "surface" of orthogonal matrices is in R^{n^2}.

When $n = 2$, all orthogonal matrices are rotations and reflections, and rotations have the form

$$Q = \begin{bmatrix} \cos\theta & \sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}.$$

Hence, when $n = 2$ we have one parameter.

When $n = 3$, airplane pilots know about "roll, pitch, and yaw", which are the three parameters for the orthogonal matrices when $n = 3$. In general, in R^{n^2}, the orthogonal group has dimension $n(n \quad 1)/2$.

There are a few ways to see this.

Firstly, orthogonality $Q^T Q = I$ imposes $n(n+1)/2$ constraints, leaving $n(n \quad 1)/2$ free parameters.

When we do QR decomposition, the R "eats" up $n(n+1)/2$ of the parameters, again leaving $n(n \quad 1)/2$ for Q.

Lastly, If we think about the symmetric eigenvalue problem where $S = Q\Lambda Q^T$, S has $n(n+1)/2$ parameters and Λ has n, so Q has $n(n \quad 1)/2$.

15.2.1 Differentiating the Symmetric Eigendecomposition

Let S be a symmetric matrix, Λ be diagonal containing eigenvalues of S, and Q be orthogonal with column vectors as eigenvectors of S such that $S = Q\Lambda Q^T$. [For simplicity, let's assume that the eigenvalues are "simple" (multiplicity 1); repeated eigenvalues turn out to greatly complicate the analysis of perturbations because of the ambiguity in their eigenvector basis.] Then, we have

$$dS = dQ\,\Lambda Q^T + Q\,d\Lambda\,Q^T + Q\Lambda dQ^T,$$

which may be written as

$$Q^T dS\,Q = Q^T dQ\Lambda \quad \Lambda Q^T dQ + d\Lambda.$$

As an exercise, one may check that the left and right hand sides of the above are both symmetric. This may be easier if one looks at the diagonal entries on their own, as there $(Q^T dS\, Q)_{ii} = q_i^T dS\, q_i$. Since q_i is the ith eigenvector, this implies $q_i^T dS\, q_i = d\lambda_i$. (In physics, this is sometimes called the "Hellman–Feynman" theorem, or non-degenerate first-order eigenvalue-perturbation theory.)

Sometimes we think of a curve of matrices $S(t)$ depending on a parameter such as time. If we ask for $\frac{d\lambda_i}{dt}$, this implies it is thus equal to $q_i^T \frac{dS(t)}{dt} q_i$. So how can we get the gradient $\nabla \lambda_i$ for one of the eigenvalues? Well, firstly, note that

$$\mathrm{tr}(q_i q_i^T)^T dS) = d\lambda_i \implies \nabla \lambda_i = q_i q_i^T.$$

What about the eigenvectors? Those come from off diagonal elements, where for $i \neq j$,

$$(Q^T dS\, Q)_{ij} = \left(Q^T \frac{dQ}{dt} \right)_{ij} (\lambda_j - \lambda_i).$$

Therefore, we can form the elements of $Q^T \frac{dQ}{dt}$, and left multiply by Q to obtain $\frac{dQ}{dt}$ (as Q is orthogonal).

It is interesting to get the second derivative of eigenvalues when moving along a line in symmetric matrix space. For simplicity, suppose Λ is diagonal and $S(t) = \Lambda + tE$. Therefore, differentiating

$$\frac{d\Lambda}{dt} = \mathrm{diag}\left(Q^T \frac{dS(t)}{dt} Q \right),$$

we get

$$\frac{d^2\Lambda}{dt^2} = \mathrm{diag}\left(Q^T \frac{d^2 S(t)}{dt^2} Q \right) + 2\,\mathrm{diag}\left(Q^T \frac{dS(t)}{dt} \frac{dQ}{dt} \right).$$

Evaluating this at $Q = I$ and recognizing the first term is zero as we are on a line, we have that

$$\frac{d^2\Lambda}{dt^2} = 2\,\mathrm{diag}\left(E \quad \frac{dQ}{dt} \right),$$

or

$$\frac{d^2\Lambda}{dt^2} = 2 \sum_{k \neq i} E_{ik}^2 / (\lambda_i - \lambda_k).$$

Using this, we can write out the eigenvalues as a Taylor series:

$$\lambda_i(\epsilon) = \lambda_i + \epsilon E_{ii} + \epsilon^2 \sum_{k \neq i} E_{ik}^2 / (\lambda_i - \lambda_k) + \ldots.$$

(In physics, this is known as second-order eigenvalue perturbation theory.)

16 AD on Computational Graphs, ctd.

In this lecture, Professor Edelman reviewed the first part of Section 10.2, and then covered reverse-mode automatic differentiation on graphs. This material is included in Section 10.2.1 of the notes.

17 Where We Go From Here

There are many topics that we did not have time to cover, even in 16 hours of lectures. If you came into this class thinking that taking derivatives is easy and you already learned everything there is to know about it in first-year calculus, hopefully we've convinced you that it is an enormously rich subject that is impossible to exhaust in a single course. Some of the things it might have been nice to include are:

When automatic differentiation (AD) hits something it cannot handle, you may have to write a custom Jacobian–vector product (a "Jvp," "frule," or "pushforward") in forward-mode, and/or a custon row vector–Jacobian product (a "vJp," "rrule," "pullback," or "Jacobian^T-vector product") in reverse-mode. In Julia with Zygote AD, this is done using the ChainRules packages. In Python with JAX, this is done with jax.custon_jvp and/or jax.custon_vjp respectively. In principle, this is straightforward, but the APIs can take some getting used to because the of the generality that they support.

For functions $f(z)$ with complex arguments z (i.e. complex vector spaces), you cannot take "ordinary" complex derivatives whenever the function involves the conjugate $\overline{z}$, for example, $|z|$, $\mathrm{Re}(z)$, and $\mathrm{Im}(z)$. This *must* occur if $f(z)$ is purely real-valued and not constant, as in optimization problems involving complex-number calculations. One option is to write $z = x + iy$ and treat $f(z)$ as a two-argument function $f(x, y)$ with real derivatives, but this can be awkward if your problem is "naturally" expressed in terms of complex variables (for instance, the Fourier frequency domain). A common alternative is the "CR calculus" (or "Wirtinger calculus"), in which you write

$$df = \left(\frac{\partial f}{\partial z}\right) dz + \left(\frac{\partial f}{\partial \overline{z}}\right) d\overline{z},$$

as if z and $\overline{z}$ were independent variables. This can be extended to gradients, Jacobians, steepest-descent, and Newton iterations, for example. A nice review of this concept can be found in these UCSD course notes by K. Kreuz Delgado.

Many, many more derivative results for matrix functions and factorizations can be found in the literature, some of them quite tricky to derive. For example, a number of references are listed in this GitHub issue for the ChainRules package.

Another important generalization of differential calculus is to derivatives on curved manifolds and differential geometry, leading to the exterior derivative.

When differentiating eigenvalues λ of matrices $A(x)$, a complication arises at eigenvalue crossings (where multiplicity $k > 1$). Here, the eigenvalues and eigenvectors usually cease to be differentiable. More generally, this problem arises for any implicit function with a repeated root. In this case, one option is use an expanded definition of sensitivity analysis called a **generalized gradient** (a $k \quad k$ matrix-valued linear operator $G(x)[dx]$ whose *eigenvalues* are the perturbations $d\lambda$. See for example Cox (1995), Seyranian *et al.* (1994), and Stechlinski (2022). Physicistss call a related idea "degenerate perturbation theory." A recent formulation of similar ideas is called the **lexicographic directional derivative**. See for example Nesterov (2005) and Barton *et al.* (2017).

Sometimes, optimization problems involving eigenvalues can be reformulated to avoid this difficulty by using SDP constraints. See for example Men *et al.* (2014).

For a defective matrix the situation is worse: even the generalized derivatives blow up because $d\lambda$ can be proportional to (e.g.) the square root of the perturbation $\|dA\|$ (for an eigenvalue with algebraic multiplicity $= 2$ and geometric multiplicity $= 1$).

Famous generalizations of differentation are the "distributional" and "weak" derivatives. For example, to obtain Dirac delta "functions" by differentiating discontinuities. This requires changing not only the definition of "derivative," but also changing the definition of *function*, as reviewed at an elementary level in these MIT course notes.