Mathematical Methods for Computer Vision, Robotics, and Graphics

Course notes for CS 205A, Fall 2013

Justin Solomon
Department of Computer Science
Stanford University

Contents

Part I

Preliminaries

Chapter 0

Mathematics Review

In this chapter we will review relevant notions from linear algebra and multivariable calculus that will figure into our discussion of computational techniques. It is intended as a review of background material with a bias toward ideas and interpretations commonly encountered in practice; the chapter safely can be skipped or used as reference by students with stronger background in mathematics.

0.1 Preliminaries: Numbers and Sets

Rather than considering algebraic (and at times philosophical) discussions like "What is a number?," we will rely on intuition and mathematical common sense to define a few sets:

- The *natural numbers* $\mathbb{N} = \{1, 2, 3, \ldots\}$
- The *integers* $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$
- The *rational numbers* $\mathbb{Q} = \{a/b : a, b \in \mathbb{Z}\}$[1]
- The *real numbers* $\mathbb{R}$ encompassing $\mathbb{Q}$ as well as *irrational* numbers like π and $\sqrt{2}$
- The *complex numbers* $\mathbb{C} = \{a + bi : a, b \in \mathbb{R}\}$, where we think of i as satisfying $i = \sqrt{-1}$.

It is worth acknowledging that our definition of $\mathbb{R}$ is far from rigorous. The construction of the real numbers can be an important topic for practitioners of cryptography techniques that make use of alternative number systems, but these intricacies are irrelevant for the discussion at hand.

As with any other sets, $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{C}$ can be manipulated using generic operations to generate new sets of numbers. In particular, recall that we can define the "Euclidean product" of two sets A and B as

$$A \times B = \{(a, b) : a \in A \text{ and } b \in B\}.$$

We can take *powers* of sets by writing

$$A^n = \underbrace{A \times A \times \cdots \times A}_{n \text{ times}}.$$

[1]This is the first of many times that we will use the notation $\{A : B\}$; the braces should denote a set and the colon can be read as "such that." For instance, the definition of $\mathbb{Q}$ can be read as "the set of fractions a/b *such that* a and b are integers." As a second example, we could write $\mathbb{N} = \{n \in \mathbb{Z} : n > 0\}$.

This construction yields what will become our favorite set of numbers in chapters to come:

$$\mathbb{R}^n = \{(a_1, a_2, \ldots, a_n) : a_i \in \mathbb{R} \text{ for all } i\}$$

0.2 Vector Spaces

Introductory linear algebra courses easily could be titled "Introduction to Finite-Dimensional Vector Spaces." Although the definition of a vector space might appear abstract, we will find many concrete applications that all satisfy the formal aspects and thus can benefit from the machinery we will develop.

0.2.1 Defining Vector Spaces

We begin by defining a vector space and providing a number of examples:

Definition 0.1 (Vector space)**.** *A* vector space *is a set $\mathcal{V}$ that is closed under scalar multiplication and addition.*

For our purposes, a scalar is a number in $\mathbb{R}$, and the addition and multiplication operations satisfy the usual axioms (commutativity, associativity, and so on). It is usually straightforward to spot vector spaces in the wild, including the following examples:

Example 0.1 ($\mathbb{R}^n$ as a vector space)**.** *The most common example of a vector space is $\mathbb{R}^n$. Here, addition and scalar multiplication happen component-by-component:*

$$\begin{aligned}(1,2) + (-3,4) &= (1-3, 2+4) = (-2,6)\\ 10 \cdot (-1,1) &= (10 \cdot -1, 10 \cdot 1) = (-10,10)\end{aligned}$$

Example 0.2 (Polynomials)**.** *A second important example of a vector space is the "ring" of polynomials with real number inputs, denoted $\mathbb{R}[x]$. A polynomial $p \in \mathbb{R}[x]$ is a function $p : \mathbb{R} \to \mathbb{R}$ taking the form*[2]

$$p(x) = \sum_k a_k x^k.$$

Addition and scalar multiplication are carried out in the usual way, e.g. if $p(x) = x^2 + 2x - 1$ and $q(x) = x^3$, then $3p(x) + 5q(x) = 5x^3 + 3x^2 + 6x - 3$, which is another polynomial. As an aside, for future examples note that functions like $p(x) = (x-1)(x+1) + x^2(x^3 - 5)$ are still polynomials even though they are not explicitly written in the form above.

Elements $\vec{v} \in \mathcal{V}$ of a vector space $\mathcal{V}$ are called *vectors*, and a weighted sum of the form $\sum_i a_i \vec{v}_i$, where $a_i \in \mathbb{R}$ and $\vec{v}_i \in \mathcal{V}$, is known as a *linear combination* of the $\vec{v}_i$'s. In our second example, the "vectors" are functions, although we do not normally use this language to discuss $\mathbb{R}[x]$. One way to link these two viewpoints would be to identify the polynomial $\sum_k a_k x^k$ with the sequence $(a_0, a_1, a_2, \cdots)$; remember that polynomials have finite numbers of terms, so the sequence eventually will end in a string of zeros.

[2]The notation $f : A \to B$ means f is a function that takes as input an element of set A and outputs an element of set B. For instance, $f : \mathbb{R} \to \mathbb{Z}$ takes as input a real number in $\mathbb{R}$ and outputs an integer $\mathbb{Z}$, as might be the case for $f(x) = \lfloor x \rfloor$, the "round down" function.

0.2.2 Span, Linear Independence, and Bases

Suppose we start with vectors $\vec{v}_1, \ldots, \vec{v}_k \in \mathcal{V}$ for vector space $\mathcal{V}$. By Definition 0.1, we have two ways to start with these vectors and construct new elements of $\mathcal{V}$: addition and scalar multiplication. The idea of *span* is that it describes all of the vectors you can reach via these two operations:

Definition 0.2 (Span). *The* span *of a set $S \subseteq \mathcal{V}$ of vectors is the set*

$$\textit{span}\, S \equiv \{a_1\vec{v}_1 + \cdots + a_k\vec{v}_k : k \geq 0, v_i \in \mathcal{V} \textit{ for all } i, \textit{ and } a_i \in \mathbb{R} \textit{ for all } i\}.$$

Notice that span S is a *subspace* of $\mathcal{V}$, that is, a subset of $\mathcal{V}$ that is in itself a vector space. We can provide a few examples:

Example 0.3 (Mixology). *The typical "well" at a cocktail bar contains at least four ingredients at the bartender's disposal: vodka, tequila, orange juice, and grenadine. Assuming we have this simple well, we can represent drinks as points in $\mathbb{R}^4$, with one slot for each ingredient. For instance, a typical "tequila sunrise" can be represented using the point $(0, 1.5, 6, 0.75)$, representing amounts of vodka, tequila, orange juice, and grenadine (in ounces), resp.*

The set of drinks that can be made with the typical well is contained in

$$\textit{span}\,\{(1,0,0,0),(0,1,0,0),(0,0,1,0),(0,0,0,1)\},$$

that is, all combinations of the four basic ingredients. A bartender looking to save time, however, might notice that many drinks have the same orange juice to grenadine ratio and mix the bottles. The new simplified well may be easier for pouring but can make fundamentally fewer drinks:

$$\textit{span}\,\{(1,0,0,0),(0,1,0,0),(0,0,6,0.75)\}$$

Example 0.4 (Polynomials). *Define the $p_k(x) \equiv x^k$. Then, it is easy to see that*

$$\mathbb{R}[x] = \textit{span}\,\{p_k : k \geq 0\}.$$

Make sure you understand notation well enough to see why this is the case.

Adding another item to a set of vectors does not always increase the size of its span. For instance, in $\mathbb{R}^2$ it is clearly the case that

$$\text{span}\,\{(1,0),(0,1)\} = \text{span}\,\{(1,0),(0,1),(1,1)\}.$$

In this case, we say that the set $\{(1,0),(0,1),(1,1)\}$ is *linearly dependent*:

Definition 0.3 (Linear dependence). *We provide three equivalent definitions. A set $S \subseteq \mathcal{V}$ of vectors is* linearly dependent *if:*

1. *One of the elements of S can be written as a linear combination of the other elements, or S contains zero.*

2. *There exists a non-empty linear combination of elements $\vec{v}_k \in \mathcal{S}$ yielding $\sum_{k=1}^m c_k\vec{v}_k = 0$ where $c_k \neq 0$ for all k.*

3. *There exists $\vec{v} \in S$ such that span S = span $S\backslash\{\vec{v}\}$. That is, we can remove a vector from S without affecting its span.*

If S is not linearly dependent, then we say it is linearly independent.

Providing proof or informal evidence that each definition is equivalent to its counterparts (in an "if and only if" fashion) is a worthwhile exercise for students less comfortable with notation and abstract mathematics.

The concept of linear dependence leads to an idea of "redundancy" in a set of vectors. In this sense, it is natural to ask how large a set we can choose before adding another vector cannot possibly increase the span. In particular, suppose we have a linearly independent set $S \subseteq \mathcal{V}$, and now we choose an additional vector $\vec{v} \in \mathcal{V}$. Adding $\vec{v}$ to S leads to one of two possible outcomes:

1. The span of $S \cup \{\vec{v}\}$ is *larger* than the span of S.
2. Adding $\vec{v}$ to S has no effect on the span.

The *dimension* of $\mathcal{V}$ is nothing more than the maximal number of times we can get outcome 1, add $\vec{v}$ to S, and repeat.

Definition 0.4 (Dimension and basis)**.** *The* dimension *of $\mathcal{V}$ is the maximal size $|S|$ of a linearly-independent set $S \subset \mathcal{V}$ such that $\operatorname{span} S = \mathcal{V}$. Any set S satisfying this property is called a* basis *for $\mathcal{V}$.*

Example 0.5 ($\mathbb{R}^n$)**.** *The* standard basis *for $\mathbb{R}^n$ is the set of vectors of the form*

$$\vec{e}_k \equiv (\underbrace{0,\ldots,0}_{k-1 \text{ slots}},1,\underbrace{0,\ldots,0}_{n-k \text{ slots}}).$$

That is, $\vec{e}_k$ has all zeros except for a single one in the k-th slot. It is clear that these vectors are linearly independent and form a basis; for example in $\mathbb{R}^3$ any vector (a,b,c) can be written as $a\vec{e}_1 + b\vec{e}_2 + c\vec{e}_3$. Thus, the dimension of $\mathbb{R}^n$ is n, as we would expect.

Example 0.6 (Polynomials)**.** *It is clear that the set $\{1, x, x^2, x^3, \ldots\}$ is a linearly independent set of polynomials spanning $\mathbb{R}[x]$. Notice that this set is infinitely large, and thus the dimension of $\mathbb{R}[x]$ is ∞.*

0.2.3 Our Focus: $\mathbb{R}^n$

Of particular importance for our purposes is the vector space $\mathbb{R}^n$, the so-called *n-dimensional Euclidean space*. This is nothing more than the set of coordinate axes encountered in high school math classes:

- $\mathbb{R}^1 \equiv \mathbb{R}$ is the number line
- $\mathbb{R}^2$ is the two-dimensional plane with coordinates (x, y)
- $\mathbb{R}^3$ represents three-dimensional space with coordinates (x, y, z)

Nearly all methods in this course will deal with transformations and functions on $\mathbb{R}^n$.

For convenience, we usually write vectors in $\mathbb{R}^n$ in "column form," as follows

$$(a_1, \ldots, a_n) \equiv \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix}$$

This notation will include vectors as special cases of *matrices* discussed below.

Unlike some vector spaces, $\mathbb{R}^n$ has not only a vector space structure, but also one additional construction that makes all the difference: the *dot product*.

Definition 0.5 (Dot product). *The dot product of two vectors $\vec{a} = (a_1, \dots, a_n)$ and $\vec{b} = (b_1, \dots, b_n)$ in $\mathbb{R}^n$ is given by*

$$\vec{a} \cdot \vec{b} = \sum_{k=1}^{n} a_k b_k.$$

Example 0.7 ($\mathbb{R}^2$). *The dot product of $(1,2)$ and $(-2,6)$ is $1 \cdot -2 + 2 \cdot 6 = -2 + 12 = 10$.*

The dot product is an example of a *metric*, and its existence gives a notion of geometry to $\mathbb{R}^n$. For instance, we can use the Pythagorean theorem to define the *norm* or *length* of a vector $\vec{a}$ as the square root

$$\|\vec{a}\|_2 \equiv \sqrt{a_1^2 + \cdots + a_n^2} = \sqrt{\vec{a} \cdot \vec{a}}.$$

Then, the distance between two points $\vec{a}, \vec{b} \in \mathbb{R}^n$ is simply $\|\vec{b} - \vec{a}\|_2$.

Dot products yield not only notions of lengths and distances but also of angles. Recall the following identity from trigonometry for $\vec{a}, \vec{b} \in \mathbb{R}^3$:

$$\vec{a} \cdot \vec{b} = \|\vec{a}\| \|\vec{b}\| \cos\theta$$

where θ is the angle between $\vec{a}$ and $\vec{b}$. For $n \geq 4$, however, the notion of "angle" is much harder to visualize for $\mathbb{R}^n$. We might *define* the angle θ between $\vec{a}$ and $\vec{b}$ to be the value θ given by

$$\theta \equiv \arccos \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\| \|\vec{b}\|}.$$

We must do our homework before making such a definition! In particular, recall that cosine outputs values in the interval $[-1, 1]$, so we must check that the input to arc cosine (also notated $\cos^{-1}$) is in this interval; thankfully, the well-known Cauchy-Schwarz inequality $\vec{a} \cdot \vec{b} \leq \|\vec{a}\| \|\vec{b}\|$ guarantees exactly this property.

When $\vec{a} = c\vec{b}$ for some $c \in \mathbb{R}$, we have $\theta = \arccos 1 = 0$, as we would expect: the angle between parallel vectors is zero. What does it mean for vectors to be perpendicular? Let's substitute $\theta = 90°$. Then, we have

$$\begin{aligned} 0 &= \cos 90° \\ &= \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\| \|\vec{b}\|} \end{aligned}$$

Multiplying both sides by $\|\vec{a}\| \|\vec{b}\|$ motivates the definition:

Definition 0.6 (Orthogonality). *Two vectors are perpendicular, or* orthogonal, *when $\vec{a} \cdot \vec{b} = 0$.*

This definition is somewhat surprising from a geometric standpoint. In particular, we have managed to define what it means to be perpendicular without any explicit use of angles. This construction will make it easier to solve certain problems for which the nonlinearity of sine and cosine might have created headache in simpler settings.

Aside 0.1. *There are many theoretical questions to ponder here, some of which we will address in future chapters when they are more motivated:*

- *Do all vector spaces admit dot products or similar structures?*
- *Do all finite-dimensional vector spaces admit dot products?*
- *What might be a reasonable dot product between elements of* $\mathbb{R}[x]$*?*

Intrigued students can consult texts on real and functional analysis.

0.3 Linearity

A function between vector spaces that preserves structure is known as a *linear* function:

Definition 0.7 (Linearity). *Suppose* $\mathcal{V}$ *and* $\mathcal{V}'$ *are vector spaces. Then,* $\mathcal{L} : \mathcal{V} \to \mathcal{V}'$ *is* linear *if it satisfies the following two criteria for all* $\vec{v}, \vec{v}_1, \vec{v}_2 \in \mathcal{V}$ *and* $c \in \mathbb{R}$*:*

- $\mathcal{L}$ *preserves sums:* $\mathcal{L}[\vec{v}_1 + \vec{v}_2] = \mathcal{L}[\vec{v}_1] + \mathcal{L}[\vec{v}_2]$
- $\mathcal{L}$ *preserves scalar products:* $\mathcal{L}[c\vec{v}] = c\mathcal{L}[\vec{v}]$

It is easy to generate linear maps between vector spaces, as we can see in the following examples:

Example 0.8 (Linearity in $\mathbb{R}^n$). *The following map* $f : \mathbb{R}^2 \to \mathbb{R}^3$ *is linear:*

$$f(x, y) = (3x, 2x + y, -y)$$

We can check linearity as follows:

- *Sum preservation:*

$$\begin{aligned} f(x_1 + x_2, y_1 + y_2) &= (3(x_1 + x_2), 2(x_1 + x_2) + (y_1 + y_2), -(y_1 + y_2)) \\ &= (3x_1, 2x_1 + y_1, -y_1) + (3x_2, 2x_2 + y_2, -y_2) \\ &= f(x_1, y_1) + f(x_2, y_2) \end{aligned}$$

- *Scalar product preservation:*

$$\begin{aligned} f(cx, cy) &= (3cx, 2cx + cy, -cy) \\ &= c(3x, 2x + y, -y) \\ &= cf(x, y) \end{aligned}$$

Contrastingly, $g(x, y) \equiv xy^2$ *is not linear. For instance,* $g(1, 1) = 1$ *but* $g(2, 2) = 8 \neq 2 \cdot g(1, 1)$*, so this form does not preserve scalar products.*

Example 0.9 (Integration). *The following "functional"* $\mathcal{L}$ *from* $\mathbb{R}[x]$ *to* $\mathbb{R}$ *is linear:*

$$\mathcal{L}[p(x)] \equiv \int_0^1 p(x)\, dx.$$

This somewhat more abstract example maps polynomials $p(x)$ to real numbers $\mathcal{L}[p(x)]$. For example, we can write

$$\mathcal{L}[3x^2+x-1] = \int_0^1 (3x^2+x-1)\,dx = \frac{1}{2}.$$

Linearity comes from the following well-known facts from calculus:

$$\int_0^1 c\cdot f(x)\,dx = c\int_0^1 f(x)\,dx$$
$$\int_0^1 [f(x)+g(x)]\,dx = \int_0^1 f(x)\,dx + \int_0^1 g(x)\,dx$$

We can write a particularly nice form for linear maps on $\mathbb{R}^n$. Recall that the vector $\vec{a} = (a_1,\ldots,a_n)$ is equal to the sum $\sum_k a_k\vec{e}_k$, where $\vec{e}_k$ is the k-th standard basis vector. Then, if $\mathcal{L}$ is linear we know:

$$\begin{aligned}\mathcal{L}[\vec{a}] &= \mathcal{L}\left[\sum_k a_k\vec{e}_k\right] \text{ for the standard basis } \vec{e}_k\\ &= \sum_k \mathcal{L}\left[a_k\vec{e}_k\right] \text{ by sum preservation}\\ &= \sum_k a_k\mathcal{L}\left[\vec{e}_k\right] \text{ by scalar product preservation}\end{aligned}$$

This derivation shows the following important fact:

$\mathcal{L}$ is completely determined by its action on the standard basis vectors $\vec{e}_k$.

That is, for any vector $\vec{a}\in\mathbb{R}^n$, we can use the sum above to determine $\mathcal{L}[\vec{a}]$ by linearly combining $\mathcal{L}[\vec{e}_1],\ldots,\mathcal{L}[\vec{e}_n]$.

Example 0.10 (Expanding a linear map)**.** *Recall the map in Example 0.8 given by $f(x,y) = (3x, 2x+y, -y)$. We have $f(\vec{e}_1) = f(1,0) = (3,2,0)$ and $f(\vec{e}_2) = f(0,1) = (0,1,-1)$. Thus, the formula above shows:*

$$f(x,y) = xf(\vec{e}_1) + yf(\vec{e}_2) = x\begin{pmatrix}3\\2\\0\end{pmatrix} + y\begin{pmatrix}0\\1\\-1\end{pmatrix}$$

0.3.1 Matrices

The expansion of linear maps above suggests one of many contexts in which it is useful to store multiple vectors in the same structure. More generally, say we have n vectors $\vec{v}_1,\ldots,\vec{v}_n\in\mathbb{R}^m$. We can write each as a column vector:

$$\vec{v}_1 = \begin{pmatrix}v_{11}\\v_{21}\\\vdots\\v_{m1}\end{pmatrix}, \vec{v}_2 = \begin{pmatrix}v_{12}\\v_{22}\\\vdots\\v_{m2}\end{pmatrix}, \cdots, \vec{v}_n = \begin{pmatrix}v_{1n}\\v_{2n}\\\vdots\\v_{mn}\end{pmatrix}$$

Carrying these around separately can be cumbersome notationally, so to simplify matters we simply combine them into a single $m \times n$ matrix:

$$\begin{pmatrix} | & | & & | \\ \vec{v}_1 & \vec{v}_2 & \cdots & \vec{v}_n \\ | & | & & | \end{pmatrix} = \begin{pmatrix} v_{11} & v_{12} & \cdots & v_{1n} \\ v_{21} & v_{22} & \cdots & v_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ v_{m1} & v_{m2} & \cdots & v_{mn} \end{pmatrix}$$

We will call the space of such matrices $\mathbb{R}^{m \times n}$.

Example 0.11 (Identity matrix)**.** *We can store the standard basis for $\mathbb{R}^n$ in the $n \times n$ "identity matrix" $I_{n\times n}$ given by:*

$$I_{n\times n} \equiv \begin{pmatrix} | & | & & | \\ \vec{e}_1 & \vec{e}_2 & \cdots & \vec{e}_n \\ | & | & & | \end{pmatrix} = \begin{pmatrix} 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \\ 0 & 0 & \cdots & 0 & 1 \end{pmatrix}$$

Since we constructed matrices as convenient ways to store sets of vectors, we can use multiplication to express how they can be combined linearly. In particular, a matrix in $\mathbb{R}^{m\times n}$ can be multiplied by a column vector in $\mathbb{R}^n$ as follows:

$$\begin{pmatrix} | & | & & | \\ \vec{v}_1 & \vec{v}_2 & \cdots & \vec{v}_n \\ | & | & & | \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix} \equiv c_1\vec{v}_1 + c_2\vec{v}_2 + \cdots + c_n\vec{v}_n$$

Expanding this sum yields the following explicit formula for matrix-vector products:

$$\begin{pmatrix} v_{11} & v_{12} & \cdots & v_{1n} \\ v_{21} & v_{22} & \cdots & v_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ v_{m1} & v_{m2} & \cdots & v_{mn} \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix} = \begin{pmatrix} c_1v_{11} + c_2v_{12} + \cdots + c_nv_{1n} \\ c_1v_{21} + c_2v_{22} + \cdots + c_nv_{2n} \\ \vdots \\ c_1v_{m1} + c_2v_{m2} + \cdots + c_nv_{mn} \end{pmatrix}$$

Example 0.12 (Identity matrix multiplication)**.** *It is clearly true that for any $\vec{x} \in \mathbb{R}^n$, we can write $\vec{x} = I_{n\times n}\vec{x}$, where $I_{n\times n}$ is the identity matrix from Example 0.11.*

Example 0.13 (Linear map)**.** *We return once again to the expression from Example 0.8 to show one more alternative form:*

$$f(x,y) = \begin{pmatrix} 3 & 0 \\ 2 & 1 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

We similarly define a product between a matrix in $M \in \mathbb{R}^{m\times n}$ and another matrix in $\mathbb{R}^{n\times p}$ by concatenating individual matrix-vector products:

$$M \begin{pmatrix} | & | & & | \\ \vec{c}_1 & \vec{c}_2 & \cdots & \vec{c}_n \\ | & | & & | \end{pmatrix} \equiv \begin{pmatrix} | & | & & | \\ M\vec{c}_1 & M\vec{c}_2 & \cdots & M\vec{c}_n \\ | & | & & | \end{pmatrix}$$

Example 0.14 (Mixology)**.** *Continuing Example 0.3, suppose we make a tequila sunrise and second concoction with equal parts of the two liquors in our simplified well. To find out how much of the basic ingredients are contained in each order, we could combine the recipes for each column-wise and use matrix multiplication:*

$$\begin{array}{r} \\ \textit{Vodka} \\ \textit{Tequila} \\ \textit{OJ} \\ \textit{Grenadine} \end{array}\begin{array}{c} \begin{array}{ccc} \textit{Well 1} & \textit{Well 2} & \textit{Well 3} \end{array} \\ \left(\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 6 \\ 0 & 0 & 0.75 \end{array}\right) \end{array} \cdot \begin{array}{c} \begin{array}{cc} \textit{Drink 1} & \textit{Drink 2} \end{array} \\ \left(\begin{array}{cc} 0 & 0.75 \\ 1.5 & 0.75 \\ 1 & 2 \end{array}\right) \end{array} = \begin{array}{c} \begin{array}{cc} \textit{Drink 1} & \textit{Drink 2} \end{array} \\ \left(\begin{array}{cc} 0 & 0.75 \\ 1.5 & 0.75 \\ 6 & 12 \\ 0.75 & 1.5 \end{array}\right) \end{array}\begin{array}{l} \\ \textit{Vodka} \\ \textit{Tequila} \\ \textit{OJ} \\ \textit{Grenadine} \end{array}$$

In general, we will use capital letters to represent matrices, like $A \in \mathbb{R}^{m\times n}$. We will use the notation $A_{ij} \in \mathbb{R}$ to denote the element of A at row i and column j.

0.3.2 Scalars, Vectors, and Matrices

It comes as no surprise that we can write a scalar as a 1×1 vector $c \in \mathbb{R}^{1\times 1}$. Similar, as we already suggested in §0.2.3, if we write vectors in $\mathbb{R}^n$ in column form, they can be considered $n \times 1$ matrices $\vec{v} \in \mathbb{R}^{n\times 1}$. Notice that matrix-vector products can be interpreted easily in this context; for example, if $A \in \mathbb{R}^{m\times n}$, $\vec{x} \in \mathbb{R}^n$, and $\vec{b} \in \mathbb{R}^m$, then we can write expressions like

$$\underbrace{A}_{m\times n}\underbrace{\vec{x}}_{n\times 1} = \underbrace{\vec{b}}_{m\times 1}$$

We will introduce one additional operator on matrices that is useful in this context:

Definition 0.8 (Transpose)**.** *The* transpose *of a matrix $A \in \mathbb{R}^{m\times n}$ is a matrix $A^\top \in \mathbb{R}^{n\times m}$ with elements $(A^\top)_{ij} = A_{ji}$.*

Example 0.15 (Transposition)**.** *The transpose of the matrix*

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$$

is given by

$$A^\top = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}.$$

Geometrically, we can think of transposition as flipping a matrix on its diagonal.

This unified treatment of scalars, vectors, and matrices combined with operations like transposition and multiplication can lead to slick derivations of well-known identities. For instance, we can compute the dot products of vectors $\vec{a}, \vec{b} \in \mathbb{R}^n$ by making the following series of steps:

$$
\begin{aligned}
\vec{a} \cdot \vec{b} &= \sum_{k=1}^{n} a_k b_k \\
&= \begin{pmatrix} a_1 & a_2 & \cdots & a_n \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} \\
&= \vec{a}^\top \vec{b}
\end{aligned}
$$

Many important identities from linear algebra can be derived by chaining together these operations with a few rules:

$$
\begin{aligned}
(A^\top)^\top &= A \\
(A+B)^\top &= A^\top + B^\top \\
(AB)^\top &= B^\top A^\top
\end{aligned}
$$

Example 0.16 (Residual norm)**.** *Suppose we have a matrix A and two vectors $\vec{x}$ and $\vec{b}$. If we wish to know how well $A\vec{x}$ approximates $\vec{b}$, we might define a* residual $\vec{r} \equiv \vec{b} - A\vec{x}$*; this residual is zero exactly when $A\vec{x} = \vec{b}$. Otherwise, we might use the norm $\|\vec{r}\|_2$ as a proxy for the relationship between $A\vec{x}$ and $\vec{b}$. We can use the identities above to simplify:*

$$
\begin{aligned}
\|\vec{r}\|_2^2 &= \|\vec{b} - A\vec{x}\|_2^2 \\
&= (\vec{b} - A\vec{x}) \cdot (\vec{b} - A\vec{x}) \text{ as explained in §0.2.3} \\
&= (\vec{b} - A\vec{x})^\top (\vec{b} - A\vec{x}) \text{ by our expression for the dot product above} \\
&= (\vec{b}^\top - \vec{x}^\top A^\top)(\vec{b} - A\vec{x}) \text{ by properties of transposition} \\
&= \vec{b}^\top \vec{b} - \vec{b}^\top A\vec{x} - \vec{x}^\top A^\top \vec{b} + \vec{x}^\top A^\top A\vec{x} \text{ after multiplication}
\end{aligned}
$$

All four terms on the right hand side are scalars, or equivalently 1×1 matrices. Scalars thought of as matrices trivially enjoy one additional nice property $c^\top = c$, since there is nothing to transpose! Thus, we can write

$$\vec{x}^\top A^\top \vec{b} = (\vec{x}^\top A^\top \vec{b})^\top = \vec{b}^\top A\vec{x}$$

This allows us to simplify our expression even more:

$$
\begin{aligned}
\|\vec{r}\|_2^2 &= \vec{b}^\top \vec{b} - 2\vec{b}^\top A\vec{x} + \vec{x}^\top A^\top A\vec{x} \\
&= \|A\vec{x}\|_2^2 - 2\vec{b}^\top A\vec{x} + \|\vec{b}\|_2^2
\end{aligned}
$$

We could have derived this expression using dot product identities, but intermediate steps above will prove useful in our later discussion.

0.3.3 Model Problem: $A\vec{x} = \vec{b}$

In introductory algebra class, students spend considerable time solving linear systems such as the following for triplets (x, y, z):

$$\begin{aligned} 3x + 2y + 5z &= 0 \\ -4x + 9y - 3z &= -7 \\ 2x - 3y - 3z &= 1 \end{aligned}$$

Our constructions in §0.3.1 allow us to encode such systems in a cleaner fashion:

$$\begin{pmatrix} 3 & 2 & 5 \\ -4 & 9 & -3 \\ 2 & -3 & -3 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 0 \\ -7 \\ 1 \end{pmatrix}$$

More generally, we can write linear systems of equations in the form $A\vec{x} = \vec{b}$ by following the same pattern above; here, the vector $\vec{x}$ is unknown while A and $\vec{b}$ are known. Such a system of equations is *not* always guaranteed to have a solution. For instance, if A contains only zeros, then clearly no $\vec{x}$ will satisfy $A\vec{x} = \vec{b}$ whenever $\vec{b} \neq \vec{0}$. We will defer a general consideration of when a solution exists to our discussion of linear solvers in future chapters.

A key interpretation of the system $A\vec{x} = \vec{b}$ is that it addresses task:

Write $\vec{b}$ as a linear combination of the columns of A.

Why? Recall from §0.3.1 that the product $A\vec{x}$ is encoding a linear combination of the columns of A with weights contained in elements of $\vec{x}$. So, the equation $A\vec{x} = \vec{b}$ asks that the linear combination $A\vec{x}$ equal the given vector $\vec{b}$. Given this interpretation, we define the *column space* of A to be the space of right hand sides $\vec{b}$ for which the system has a solution:

Definition 0.9 (Column space)**.** *The* column space *of a matrix $A \in \mathbb{R}^{m\times n}$ is the span of the columns of A. We can write as*

$$\text{col } A \equiv \{A\vec{x} : \vec{x} \in \mathbb{R}^n\}.$$

One important case is somewhat easier to consider. Suppose A is square, so we can write $A \in \mathbb{R}^{n\times n}$. Furthermore, suppose that the system $A\vec{x} = \vec{b}$ has a solution *for all* choices of $\vec{b}$. The only condition on $\vec{b}$ is that it is a member of $\mathbb{R}^n$, so by our interpretation above of $A\vec{x} = \vec{b}$ we can conclude that the columns of A span $\mathbb{R}^n$.

In this case, since the linear system is always solvable suppose we plug in the standard basis $\vec{e}_1, \ldots, \vec{e}_n$ to yield vectors $\vec{x}_1, \ldots, \vec{x}_n$ satisfying $A\vec{x}_k = \vec{e}_k$ for each k. Then, we can "stack" these expressions to show:

$$A \begin{pmatrix} | & | & & | \\ \vec{x}_1 & \vec{x}_2 & \cdots & \vec{x}_n \\ | & | & & | \end{pmatrix} = \begin{pmatrix} | & | & & | \\ A\vec{x}_1 & A\vec{x}_2 & \cdots & A\vec{x}_n \\ | & | & & | \end{pmatrix} = \begin{pmatrix} | & | & & | \\ \vec{e}_1 & \vec{e}_2 & \cdots & \vec{e}_n \\ | & | & & | \end{pmatrix} = I_{n\times n},$$

where $I_{n\times n}$ is the identity matrix from Example 0.11. We will call the matrix with columns $\vec{x}_k$ the *inverse* A^{-1}, which satisfies

$$AA^{-1} = A^{-1}A = I_{n\times n}.$$

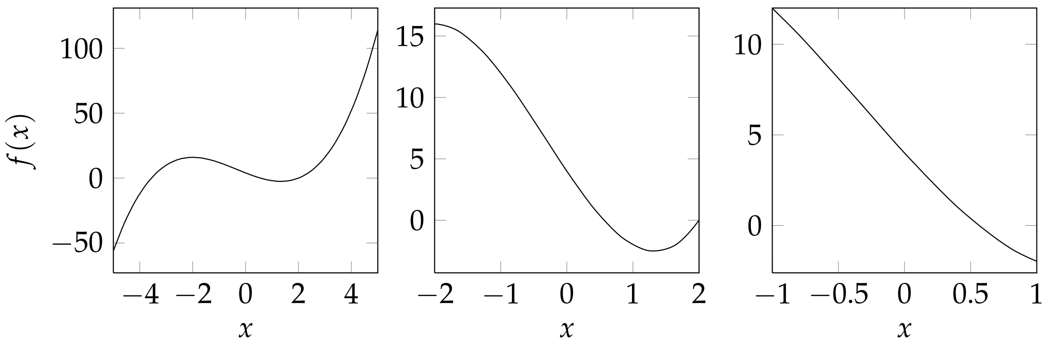

Figure 1: The closer we zoom into $f(x) = x^3 + x^2 - 8x + 4$, the more it looks like a line.

It is also easy to check that $(A^{-1})^{-1} = A$. When such an inverse exists, it is easy to solve the system $A\vec{x} = \vec{b}$. In particular, we find:

$$\vec{x} = I_{n\times n}\vec{x} = (A^{-1}A)\vec{x} = A^{-1}(A\vec{x}) = A^{-1}\vec{b}$$

0.4 Non-Linearity: Differential Calculus

While the beauty and applicability of linear algebra makes it a key target of study, nonlinearities abound in nature and we often must design computational systems that can deal with this fact of life. After all, at the most basic level the square in the famous relationship $E = mc^2$ makes it less than amenable to linear analysis.

0.4.1 Differentiation

While many functions are *globally* nonlinear, *locally* they exhibit linear behavior. This idea of "local linearity" is one of the main motivators behind differential calculus. For instance, Figure 1 shows that if you zoom in close enough to a smooth function eventually it looks like a line. The derivative $f'(x)$ of a function $f(x) : \mathbb{R} \to \mathbb{R}$ is nothing more than the slope of the approximating line, computed by finding the slope of lines through closer and closer points to x:

$$f'(x) = \lim_{y\to x} \frac{f(y) - f(x)}{y - x}$$

We can express local linearity by writing $f(x + \Delta x) = f(x) + \Delta x \cdot f'(x) + O(\Delta x^2)$.

If the function f takes multiple inputs, then it can be written $f(\vec{x}) : \mathbb{R}^n \to \mathbb{R}$ for $\vec{x} \in \mathbb{R}^n$; in other words, to each point $\vec{x} = (x_1, \ldots, x_n)$ in n-dimensional space f assigns a single number $f(x_1, \ldots, x_n)$. Our idea of local linearity breaks down somewhat here, because lines are one-dimensional objects. However, fixing all but one variable reduces back to the case of single-variable calculus. For instance, we could write $g(t) = f(t, x_2, \ldots, x_n)$, where we simply fix constants $x_2, \ldots, x_n$. Then, $g(t)$ is a differentiable function of a single variable. Of course, we could have put t in any of the input slots for f, so in general we make the following definition of the *partial derivative* of f:

Definition 0.10 (Partial derivative)**.** *The k-th* partial derivative *of f, notated $\frac{\partial f}{\partial x_k}$, is given by differentiating f in its k-th input variable:*

$$\frac{\partial f}{\partial x_k}(x_1, \ldots, x_n) \equiv \frac{d}{dt} f(x_1, \ldots, x_{k-1}, t, x_{k+1}, \ldots, x_n)|_{t=x_k}$$

The notation "$|_{t=x_k}$" should be read as "evaluated at $t = x_k$."

Example 0.17 (Relativity)**.** *The relationship $E = mc^2$ can be thought of as a function from m and c to E. Thus, we could write $E(m, c) = mc^2$, yielding the derivatives*

$$\begin{aligned}\frac{\partial E}{\partial m} &= c^2 \\ \frac{\partial E}{\partial c} &= 2mc\end{aligned}$$

Using single-variable calculus, we can write:

$$\begin{aligned} f(\vec{x} + \Delta\vec{x}) &= f(x_1 + \Delta x_1, x_2 + \Delta x_2, \ldots, x_n + \Delta x_n) \\ &= f(x_1, x_2 + \Delta x_2, \ldots, x_n + \Delta x_n) + \frac{\partial f}{\partial x_1}\Delta x_1 + O(\Delta x_1^2) \text{ by single-variable calculus} \\ &= f(x_1, \ldots, x_n) + \sum_{k=1}^{n}\left[\frac{\partial f}{\partial x_k}\Delta x_k + O(\Delta x_k^2)\right] \text{ by repeating this } n \text{ times} \\ &= f(\vec{x}) + \nabla f(\vec{x}) \cdot \Delta\vec{x} + O(\|\vec{x}\|^2) \end{aligned}$$

where we define the *gradient* of f as

$$\nabla f \equiv \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \cdots, \frac{\partial f}{\partial x_n}\right) \in \mathbb{R}^n$$

From this relationship, it is easy to see that f can be differentiated in any direction $\vec{v}$; we can evaluate this derivative $D_{\vec{v}} f$ as follows:

$$\begin{aligned} D_{\vec{v}} f(\vec{x}) &\equiv \frac{d}{dt} f(\vec{x} + t\vec{v})|_{t=0} \\ &= \nabla f(\vec{x}) \cdot \vec{v} \end{aligned}$$

Example 0.18 ($\mathbb{R}^2$)**.** *Take $f(x, y) = x^2 y^3$. Then,*

$$\begin{aligned}\frac{\partial f}{\partial x} &= 2xy^3 \\ \frac{\partial f}{\partial y} &= 3x^2 y^2\end{aligned}$$

Thus, we can write $\nabla f(x, y) = (2xy^3, 3x^2y^2)$. The derivative of f at $(1, 2)$ in the direction $(-1, 4)$ is given by $(-1, 4) \cdot \nabla f(1, 2) = (-1, 4) \cdot (16, 12) = 32$.

Example 0.19 (Linear functions)**.** *It is obvious but worth noting that the gradient of $f(\vec{x}) \equiv \vec{a} \cdot \vec{x} + \vec{c} = (a_1 x_1 + c_1, \ldots, a_n x_n + c_n)$ is $\vec{a}$.*

Example 0.20 (Quadratic forms)**.** *Take any matrix $A \in \mathbb{R}^{n\times n}$, and define $f(\vec{x}) \equiv \vec{x}^\top A\vec{x}$. Expanding this function element-by-element shows*

$$f(\vec{x}) = \sum_{ij} A_{ij}x_ix_j;$$

expanding out f and checking this relationship explicitly is worthwhile. Take some $k \in \{1,\ldots,n\}$. Then, we can can separate out all terms containing x_k:

$$f(\vec{x}) = A_{kk}x_k^2 + x_k\left(\sum_{i\neq k} A_{ik}x_i + \sum_{j\neq k} A_{kj}x_j\right) + \sum_{i,j\neq k} A_{ij}x_ix_j$$

With this factorization, it is easy to see

$$\begin{aligned}\frac{\partial f}{\partial x_k} &= 2A_{kk}x_k + \left(\sum_{i\neq k} A_{ik}x_i + \sum_{j\neq k} A_{kj}x_j\right)\\ &= \sum_{i=1}^{n}(A_{ik}+A_{ki})x_i\end{aligned}$$

This sum is nothing more than the definition of matrix-vector multiplication! Thus, we can write

$$\nabla f(\vec{x}) = A\vec{x} + A^\top\vec{x}.$$

We have generalized from $f : \mathbb{R} \to \mathbb{R}$ to $f : \mathbb{R}^n \to \mathbb{R}$. To reach full generality, we would like to consider $f : \mathbb{R}^n \to \mathbb{R}^m$. In other words, f takes in n numbers and outputs m numbers. Thankfully, this extension is straightforward, because we can think of f as a collection of single-valued functions $f_1,\ldots,f_m : \mathbb{R}^n \to \mathbb{R}$ smashed together into a single vector. That is, we write:

$$f(\vec{x}) = \begin{pmatrix} f_1(\vec{x})\\ f_2(\vec{x})\\ \vdots\\ f_m(\vec{x})\end{pmatrix}$$

Each f_k can be differentiated as before, so in the end we get a matrix of partial derivatives called the *Jacobian* of f:

Definition 0.11 (Jacobian)**.** *The* Jacobian *of $f : \mathbb{R}^n \to \mathbb{R}^m$ is the matrix $Df \in \mathbb{R}^{m\times n}$ with entries*

$$(Df)_{ij} \equiv \frac{\partial f_i}{\partial x_j}.$$

Example 0.21 (Simple function)**.** *Suppose $f(x,y) = (3x, -xy^2, x+y)$. Then,*

$$Df(x,y) = \begin{pmatrix} 3 & 0\\ -y^2 & -2xy\\ 1 & 1\end{pmatrix}.$$

Make sure you can derive this computation by hand.

Example 0.22 (Matrix multiplication)**.** *Unsurprisingly, the Jacobian of $f(\vec{x}) = A\vec{x}$ for matrix A is given by $Df(\vec{x}) = A$.*

Here we encounter a common point of confusion. Suppose a function has vector input and scalar output, that is, $f : \mathbb{R}^n \to \mathbb{R}$. We defined the gradient of f as a column vector, so to align this definition with that of the Jacobian we must write

$$Df = \nabla f^\top.$$

0.4.2 Optimization

Recall from single variable calculus minima and maxima of $f : \mathbb{R} \to \mathbb{R}$ must occur at points x satisfying $f'(x) = 0$. Of course, this condition is *necessary* rather than *sufficient*: there may exist points x with $f'(x) = 0$ that are not maxima or minima. That said, finding such *critical points* of f can be a step of a function minimization algorithm, so long as the next step ensures that the resulting x actually a minimum/maximum.

If $f : \mathbb{R}^n \to \mathbb{R}$ is minimized or maximized at $\vec{x}$, we have to ensure that there does not exist a single direction Δx from $\vec{x}$ in which f decreases or increases, resp. By the discussion in §0.4.1, this means we must find points for which $\nabla f = 0$.

Example 0.23 (Simple function)**.** *Suppose $f(x, y) = x^2 + 2xy + 4y^2$. Then, $\frac{\partial f}{\partial x} = 2x + 2y$ and $\frac{\partial f}{\partial y} = 2x + 8y$. Thus, critical points of f satisfy:*

$$\begin{aligned} 2x + 2y &= 0 \\ 2x + 8y &= 0 \end{aligned}$$

Clearly this system is solved at $(x, y) = (0, 0)$. Indeed, this is the minimum of f, as can be seen more clearly by writing $f(x, y) = (x + y)^2 + 3y^2$.

Example 0.24 (Quadratic functions)**.** *Suppose $f(\vec{x}) = \vec{x}^\top A\vec{x} + \vec{b}^\top\vec{x} + c$. Then, from the examples in the previous section we can write $\nabla f(\vec{x}) = (A^\top + A)\vec{x} + \vec{b}$. Thus, critical points $\vec{x}$ of f satisfy $(A^\top + A)\vec{x} + \vec{b} = 0$.*

Unlike single-variable calculus, when we do calculus on $\mathbb{R}^n$ we can add *constraints* to our optimization. The most general form of such a problem looks like:

$$\begin{aligned} &\text{minimize } f(\vec{x}) \\ &\text{such that } g(\vec{x}) = \vec{0} \end{aligned}$$

Example 0.25 (Rectangle areas)**.** *Suppose a rectangle has width w and height h. A classic geometry problem is to maximize area with a fixed perimeter 1:*

$$\begin{aligned} &\textit{maximize } wh \\ &\textit{such that } 2w + 2h - 1 = 0 \end{aligned}$$

When we add this constraint, we can no longer expect that critical points satisfy $\nabla f(\vec{x}) = 0$, since these points might not satisfy $g(\vec{x}) = 0$.

For now, suppose $g : \mathbb{R}^n \to \mathbb{R}$. Consider the set of points $S_0 \equiv \{\vec{x} : g(\vec{x}) = 0\}$. Obviously, any two $\vec{x}, \vec{y} \in S_0$ satisfy the relationship $g(\vec{y}) - g(\vec{x}) = 0 - 0 = 0$. Suppose $\vec{y} = \vec{x} + \Delta\vec{x}$ for small $\Delta\vec{x}$. Then, $g(\vec{y}) - g(\vec{x}) = \nabla g(\vec{x}) \cdot \Delta\vec{x} + O(\|\Delta\vec{x}\|^2)$. In other words, if we start at $\vec{x}$ satisfying $g(\vec{x}) = 0$, then if we displace in the $\Delta\vec{x}$ direction $\nabla g(\vec{x}) \cdot \Delta\vec{x} \approx 0$ to continue to satisfy this relationship.

Now, recall that the derivative of f in the direction $\vec{v}$ at $\vec{x}$ is given by $\nabla f \cdot \vec{v}$. If $\vec{x}$ is a minimum of the constrained optimization problem above, then any small displacement $\vec{x}$ to $\vec{x} + \vec{v}$ should cause an increase from $f(\vec{x})$ to $f(\vec{x} + \vec{v})$. Since we only care about displacements $\vec{v}$ preserving the $g(\vec{x} + \vec{v}) = c$ constraint, from our argument above we want $\nabla f \cdot \vec{v} = 0$ for all $\vec{v}$ satisfying $\nabla g(\vec{x}) \cdot \vec{v} = 0$. In other words, ∇f and ∇g must be parallel, a condition we can write as $\nabla f = \lambda \nabla g$ for some $\lambda \in \mathbb{R}$.

Define

$$\Lambda(\vec{x}, \lambda) = f(\vec{x}) - \lambda g(\vec{x}).$$

Then, critical points of Λ without constraints satisfy:

$$\begin{aligned} 0 &= \frac{\partial \Lambda}{\partial \lambda} = -g(\vec{x}) \\ 0 &= \nabla_{\vec{x}} \Lambda = \nabla f(\vec{x}) - \lambda \nabla g(\vec{x}) \end{aligned}$$

In other words, critical points of Λ satisfy $g(\vec{x}) = 0$ and $\nabla f(\vec{x}) = \lambda \nabla g(\vec{x})$, exactly the optimality conditions we derived!

Extending to multivariate constraints yields the following:

Theorem 0.1 (Method of Lagrange multipliers)**.** *Critical points of the constrained optimization problem above are unconstrained critical points of the Lagrange multiplier function*

$$\Lambda(\vec{x}, \vec{\lambda}) \equiv f(\vec{x}) - \vec{\lambda} \cdot g(\vec{x}),$$

with respect to both $\vec{x}$ and $\vec{\lambda}$.

Example 0.26 (Maximizing area)**.** *Continuing Example 0.25, we define the Lagrange multiplier function $\Lambda(w, h, \lambda) = wh - \lambda(2w + 2h - 1)$. Differentiating, we find:*

$$\begin{aligned} 0 &= \frac{\partial \Lambda}{\partial w} = h - 2\lambda \\ 0 &= \frac{\partial \Lambda}{\partial h} = w - 2\lambda \\ 0 &= \frac{\partial \Lambda}{\partial \lambda} = 1 - 2w - 2h \end{aligned}$$

So, critical points of the system satisfy

$$\begin{pmatrix} 0 & 1 & -2 \\ 1 & 0 & -2 \\ 2 & 2 & 0 \end{pmatrix} \begin{pmatrix} w \\ h \\ \lambda \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

Solving the system shows $w = h = 1/4$ and $\lambda = 1/8$. In other words, for a fixed amount of perimeter, the rectangle with maximal area is a square.

Example 0.27 (Eigenproblems). *Suppose that A is a symmetric positive definite matrix, meaning $A^\top = A$ (symmetry) and $\vec{x}^\top A\vec{x} > 0$ for all $\vec{x} \in \mathbb{R}^n\backslash\{\vec{0}\}$ (positive definite). Often we wish to minimize $\vec{x}^\top A\vec{x}$ subject to $\|x\|_2^2 = 1$ for a given matrix $A \in \mathbb{R}^{n\times n}$; notice that without the constraint the minimum trivially takes place at $\vec{x} = \vec{0}$. We define the Lagrange multiplier function*

$$\begin{aligned}\Lambda(\vec{x}, \lambda) &= \vec{x}^\top A\vec{x} - \lambda(\|\vec{x}\|_2^2 - 1)\\ &= \vec{x}^\top A\vec{x} - \lambda(\vec{x}^\top\vec{x} - 1).\end{aligned}$$

Differentiating with respect to $\vec{x}$, we find

$$0 = \nabla_{\vec{x}}\Lambda = 2A\vec{x} - 2\lambda\vec{x}$$

In other words, $\vec{x}$ is an eigenvector *of the matrix A:*

$$A\vec{x} = \lambda\vec{x}.$$

0.5 Problems

Problem 0.1. *Take $C^1(\mathbb{R})$ to be the set of functions $f : \mathbb{R} \to \mathbb{R}$ that admit a first derivative $f'(x)$. Why is $C^1(\mathbb{R})$ a vector space? Prove that $C^1(\mathbb{R})$ has dimension ∞.*

Problem 0.2. *Suppose the rows of $A \in \mathbb{R}^{m\times n}$ are given by the transposes of $\vec{r}_1, \ldots, \vec{r}_m \in \mathbb{R}^n$ and the columns of $A \in \mathbb{R}^{m\times n}$ are given by $\vec{c}_1, \ldots, \vec{c}_n \in \mathbb{R}^m$. That is,*

$$A = \begin{pmatrix} - & \vec{r}_1^\top & - \\ - & \vec{r}_2^\top & - \\ & \vdots & \\ - & \vec{r}_m^\top & - \end{pmatrix} = \begin{pmatrix} | & | & & | \\ \vec{c}_1 & \vec{c}_2 & \cdots & \vec{c}_n \\ | & | & & | \end{pmatrix}.$$

Give expressions for the elements of $A^\top A$ and $AA^\top$ in terms of these vectors.

Problem 0.3. *Give a linear system of equations satisfied by minima of the energy $f(\vec{x}) = \|A\vec{x} - \vec{b}\|_2^2$ with respect to $\vec{x}$, for $\vec{x} \in \mathbb{R}^n$, $A \in \mathbb{R}^{m\times n}$, and $\vec{b} \in \mathbb{R}^m$. This system is called the "normal equations" and will appear elsewhere in these notes; even so, it is worth working through and fully understanding the derivation.*

Problem 0.4. *Suppose $A, B \in \mathbb{R}^{n\times n}$. Formulate a condition for vectors $\vec{x} \in \mathbb{R}^n$ to be critical points of $\|A\vec{x}\|_2^2$ subject to $\|Bx\|_2^2 = 1$. Also, give an alternative form for the optimal values of $\|A\vec{x}\|_2^2$.*

Problem 0.5. *Fix some vector $\vec{a} \in \mathbb{R}^n\backslash\{\vec{0}\}$ and define $f(\vec{x}) = \vec{a}\cdot\vec{x}$. Give an expression for the maximum of $f(\vec{x})$ subject to $\|\vec{x}\| = 1$.*

Chapter 1

Numerics and Error Analysis

In studying numerical analysis, we move from dealing with `int`s and `long`s to `float`s and `double`s. This seemingly innocent transition comprises a huge shift in how we must think about algorithmic design and implementation. Unlike the basics of discrete algorithms, we no longer can expect our algorithms to yield exact solutions in all cases. "Big O" and operation counting do not always reign supreme; instead, even in understanding the most basic techniques we are forced to study the trade off between timing, approximation error, and so on.

1.1 Storing Numbers with Fractional Parts

Recall that computers generally store data in *binary* format. In particular, each digit of a positive integer corresponds to a different power of two. For instance, we might convert 463 to binary using the following table:

1	1	1	0	0	1	1	1	1
2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

In other words, this notation encodes the fact that 463 can be decomposed into powers of two uniquely as:

$$\begin{aligned} 463 &= 2^8 + 2^7 + 2^6 + 2^3 + 2^2 + 2^1 + 2^0 \\ &= 256 + 128 + 64 + 8 + 4 + 2 + 1 \end{aligned}$$

Issues of overflow aside, all positive integers can be written in this form using a finite number of digits. Negative numbers also can be represented this way, either by introducing a leading sign bit or by using the "two's complement" trick.

Such a decomposition inspires a simple extension to numbers that include fractions: simply include negative powers of two. For instance, decomposing 463.25 is as simple as adding two slots:

1	1	1	0	0	1	1	1	1.	0	1
2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}

Just as in the decimal system, however, representing fractional parts of numbers this way is not nearly as well-behaved as representing integers. For instance, writing the fraction $1/3$ in binary yields the expression:

$$\frac{1}{3} = 0.0101010101\ldots$$

Such examples show that there exist numbers at all scales that cannot be represented using a finite binary string. In fact, numbers like $\pi = 11.00100100001\ldots_2$ have infinitely-long expansions regardless of which (integer) base you use!

For this reason, when designing computational systems that do math on $\mathbb{R}$ instead of $\mathbb{Z}$, we are forced to make approximations for nearly any reasonably efficient numerical representation. This can lead to many points of confusion while coding. For instance, consider the following C++ snippet:

```
double x = 1.0;
double y = x / 3.0;
if (x == y*3.0) cout << "They are equal!";
else cout << "They are NOT equal.";
```

Contrary to intuition, this program prints `"They are NOT equal."` Why? The definition of `y` makes an approximation to $1/3$ since it cannot be written as a terminating binary string, rounding to a nearby number it can represent. Thus, `y*3.0` no longer is multiplying 3 by $1/3$. One way to fix this issue is below:

```
double x = 1.0;
double y = x / 3.0;
if (fabs(x-y*3.0) < numeric_limits<double>::epsilon) cout << "They are equal!";
else cout << "They are NOT equal.";
```

Here, we check that `x` and `y*3.0` are within some tolerance of one another rather than checking exact equality. This is an example of a very important point:

Rarely if ever should the operator == and its equivalents be used on fractional values. Instead, some *tolerance* should be used to check if numbers are equal.

Of course, there is a tradeoff here: the size of the tolerance defines a line between equality and "close-but-not-the-same," which must be chosen carefully for a given application.

We consider a few options for representing numbers on a computer below.

1.1.1 Fixed Point Representations

The most straightforward option for storing fractions is to add a *fixed* decimal point. That is, as in the example above we represent values by storing 0/1 coefficients in front of powers of two that range from 2^{-k} to 2^{ℓ} for some $k, \ell \in \mathbb{Z}$. For instance, representing all nonnegative values between 0 and 127.75 in increments of $1/4$ is as easy as taking $k = 2$ and $\ell = 7$; in this situation, we represent these values using 9 binary digits, of which two occur after the decimal point.

The primary advantage of this representation is that nearly all arithmetic operations can be carried out using the same algorithms as with integers. For example, it is easy to see that

$$a + b = (a \cdot 2^k + b \cdot 2^k) \cdot 2^{-k}.$$

Multiplying our fixed representation by 2^k guarantees the result is integral, so this observation essentially shows that addition can be carried out using integer addition essentially by "ignoring" the decimal point. Thus, rather than using specialized hardware, the pre-existing integer arithmetic logic unit (ALU) carries out fixed-point mathematics quickly.

Fixed-point arithmetic may be fast, but it can suffer from serious precision issues. In particular, it is often the case that the output of a binary operation like multiplication or division can require more bits than the operands. For instance, suppose we include one decimal point of precision and wish to carry out the product $1/2 \cdot 1/2 = 1/4$. We write $0.1_2 \times 0.1_2 = 0.01_2$, which gets truncated to 0. In this system, it is fairly straightforward to combine fixed point numbers in a reasonable way and get an unreasonable result.

Due to these drawbacks, most major programming languages do not by default include a fixed-point decimal data type. The speed and regularity of fixed-pont arithmetic, however, can be a considerable advantage for systems that favor timing over accuracy. In fact, some lower-end graphics processing units (GPU) implement only these operations since a few decimal points of precision is sufficient for many graphical applications.

1.1.2 Floating Point Representations

One of many numerical challenges in writing scientific applications is the variety of scales that can appear. Chemists alone deal with values anywhere between 9.11×10^{-31} and 6.022×10^{23}. An operation as innocent as a change of units can cause a sudden transition between these regimes: the same observation written in kilograms per lightyear will look considerably different in megatons per second. As numerical analysts, our job is to write software that can transition between these scales gracefully without imposing on the client unnatural restrictions on their techniques.

A few notions and obsevations from the art of scientific measurement are relevant to such a discussion. First, obviously one of the following representations is more compact than the other:

$$6.022 \times 10^{23} = 602,200,000,000,000,000,000,000$$

Furthermore, in the absence of exceptional scientific equipment, the difference between 6.022×10^{23} and $6.022 \times 10^{23} + 9.11 \times 10^{-31}$ is negligible. One way to come to this conclusion is to say that 6.022×10^{23} has only three *digits of precision* and probably represents some range of possible measurements $[6.022 \times 10^{23} - \varepsilon, 6.011 \times 10^{23} + \varepsilon]$ for some $\varepsilon \sim 0.001 \times 10^{23}$.

Our first observation was able to compactify our representation of 6.022×10^{23} by writing it in *scientific notation*. This number system separates the "interesting" digits of a number from its order of magnitude by writing it in the form $a \times 10^b$ for some $a \sim 1$ and $b \in \mathbb{Z}$. We call this format the *floating-point* form of a number, because unlike the fixed-point setup in §1.1.1, here the decimal point "floats" to the top. We can describe floating point systems using a few parameters (CITE):

- The *base* $\beta \in \mathbb{N}$; for scientific notation explained above, the base is 10
- The *precision* $p \in \mathbb{N}$ representing the number of digits in the decimal expansion
- The range of exponents $[L, U]$ representing the possible values of b

Such an expansion looks like:

$$\underbrace{\pm}_{\text{sign}} \underbrace{(d_0 + d_1 \cdot \beta^{-1} + d_2 \cdot \beta^{-2} + \cdots + d_{p-1} \cdot \beta^{1-p})}_{\text{mantissa}} \times \underbrace{\beta^b}_{\text{exponent}}$$

where each digit d_k is in the range $[0, \beta - 1]$ and $b \in [L, U]$.

Floating point representations have a curious property that can affect software in unexpected ways: Their spacing is uneven. For example, the number of values representable between β and β^2 is the same as that between β^2 and β^3 even though usually $\beta^3 - \beta^2 > \beta^2 - \beta$. To understand the precision possible with a given number system, we will define the *machine precision* ε_m as the smallest $\varepsilon_m > 0$ such that $1 + \varepsilon_m$ is representable. Then, numbers like $\beta + \varepsilon_m$ are not expressible in the number system because ε_m is too small!

By far the most common standard for storing floating point numbers is provided by the IEEE 754 standard. This standard specifies several classes of floating point numbers. For instance, a double-precision floating point number is written in base $\beta = 2$ (as are most numbers on the computer), with a single $\pm$ sign bit, 52 digits for d, and a range of exponents between -1022 and 1023. The standard also specifies how to store $\pm\infty$ and values like NaN, or "not-a-number," reserved for the results of computations like $^{10}/_0$. An extra bit of precision can be gained by writing *normalizing* floating point values and assuming the most significant digit d_0 is 1 and not writing it.

The IEEE standard also includes agreed-upon options for dealing with the finite number of values that can be represented given a finite number of bits. For instance, a common unbiased strategy for rounding computations is *round to nearest, ties to even,* which breaks equidistant ties by rounding to the nearest floating point value with an even least-significant (rightmost) bit. Note that there are many equally legitimate strategies for rounding; choosing a single one guarantees that scientific software will work identically on all client machines implementing the same standard.

1.1.3 More Exotic Options

Moving forward, we will assume that decimal values are stored in floating-point format unless otherwise noted. This, however, is not to say that other numerical systems do not exist, and for specific applications an alternative choice might be necessary. We acknowledge some of those situations here.

The headache of adding tolerances to account for rounding errors might be unacceptable for some applications. This situation appears in computational geometry applications, e.g. when the difference between *nearly-* and *completely-*parallel lines may be a difficult distinction to make. One resolution might be to use *arbitrary-precision arithmetic,* that is, to implement arithmetic without rounding or error of any sort.

Arbitrary-precision arithmetic requires a specialized implementation and careful consideration for what types of values you need to represent. For instance, it might be the case that rational numbers $\mathbb{Q}$ are sufficient for a given application, which can be written as ratios $^a/_b$ for $a, b \in \mathbb{Z}$. Basic arithmetic operations can be carried out in $\mathbb{Q}$ without any loss in precision. For instance, it is easy to see

$$\frac{a}{b} \times \frac{c}{d} = \frac{ac}{bd} \qquad\qquad \frac{a}{b} \div \frac{c}{d} = \frac{ad}{bc}.$$

Arithmetic in the rationals precludes the existence of a square root operator, since values like $\sqrt{2}$ are irrational. Also, this representation is nonunique, since e.g. $^a/_b = {}^{5a}/_{5b}$.

Other times it may be useful to bracket error by representing values alongside error estimates as a pair $a, \varepsilon \in \mathbb{R}$; we think of the pair (a, ε) as the range $a \pm \varepsilon$. Then, arithmetic operations also update not only the value but also the error estimate, as in

$$(x \pm \varepsilon_1) + (y \pm \varepsilon_2) = (x + y) \pm (\varepsilon_1 + \varepsilon_2 + \text{error}(x + y)),$$

where the final term represents an estimate of the error induced by adding x and y.

1.2 Understanding Error

With the exception of the arbitrary-precision systems describe in §1.1.3, nearly every computerized representation of real numbers with fractional parts is forced to employ rounding and other approximation schemes. This scheme represents one of many sources of approximations typically encountered in numerical systems:

- *Truncation* error comes from the fact that we can only represent a finite subset of all the possible set of values in $\mathbb{R}$; for example, we must truncate long or infinite sequences past the decimal point to the number of bits we are willing to store.
- *Discretization* error comes from our computerized adaptations of calculus, physics, and other aspects of continuous mathematics. For instance, we make an approximation

  $$\frac{dy}{dx} \approx \frac{y(x+\varepsilon) - y(x)}{\varepsilon}.$$

  We will learn that this approximation is a legitimate and useful one, but depending on the choice of ε it may not be completely correct.
- *Modeling* error comes from incomplete or inaccurate descriptions of the problems we wish to solve. For instance, a simulation for predicting weather in Germany may choose to neglect the collective flapping of butterfly wings in Malaysia, although the displacement of air by these butterflies may be enough to perturb the weather patterns elsewhere somewhat.
- *Empirical constant* error comes from poor representations of physical or mathematical constants. For instance, we may compute π using a Taylor sequence that we terminate early, and even scientists may not even know the speed of light to more than some number of digits.
- *Input* error can come from user-generated approximations of parameters of a given system (and from typos!). Simulation and numerical techniques can be used to answer "what if" type questions, in which exploratory choices of input setups are chosen just to get some idea of how a system behaves.

Example 1.1 (Computational physics)**.** *Suppose we are designing a system for simulating planets as they revolve around the earth. The system essentially solves Newton's equation $F = ma$ by integrating forces forward in time. Examples of error sources in this system might include:*

- *Truncation error: Using IEEE floating point to represent parameters and output of the system and truncating when computing the product ma*
- *Discretization error: Replacing the acceleration a with a divided difference*
- *Modeling error: Neglecting to simulate the moon's effects on the earth's motion within the planetary system*
- *Empirical error: Only entering the mass of Jupiter to four digits*
- *Input error: The user may wish to evaluate the cost of sending garbage into space rather than risking a Wall-E style accumulation on Earth but can only estimate the amount of garbage the government is willing to jettison in this fashion*

1.2.1 Classifying Error

Given our previous discussion, the following two numbers might be regarded as having the same amount of potential error:

$$1 \pm 0.01$$
$$10^5 \pm 0.01$$

Although it has the size as the range $[1 - 0.01, 1 + 0.01]$, the range $[10^5 - 0.01, 10^5 + 0.01]$ appears to encode a more confident measurement because the error 0.01 is much smaller *relative* to 10^5 than to 1.

The distinction between these two classes of error is described by differentiating between *absolute* error and *relative* error:

Definition 1.1 (Absolute error)**.** *The* absolute error *of a measurement is given by the difference between the approximate value and its underlying true value.*

Definition 1.2 (Relative error)**.** *The* relative error *of a measurement is given by the absolute error divided by the true value.*

One way to distinguish between these two species of error is the use of units versus percentages.

Example 1.2 (Absolute and relative error)**.** *Here are two equivalent statements in contrasting forms:*

$$\textit{Absolute: } 2\ \textit{in} \pm 0.02\ \textit{in}$$
$$\textit{Relative: } 2\ \textit{in} \pm 1\%$$

In most applications the *true* value is unknown; after all, if this were not the case the use of an approximation in lieu of the true value may be a dubious proposition. There are two popular ways to resolve this issue. The first simply is to be conservative when carrying out computations: at each step take the largest possible error estimate and propagate these estimates forward as necessary. Such conservative estimates are powerful in that when they are small we can be *very* confident that our solution is useful.

An alternative resolution has to do with *what* you can measure. For instance, suppose we wish to solve the equation $f(x) = 0$ for x given a function $f : \mathbb{R} \to \mathbb{R}$. We know that somewhere there exists a root x_0 satisfying $f(x_0) = 0$ exactly, but if we knew this root our algorithm would not be necessary in the first place. In practice, our computational system may yield some x_{est} satisfying $f(x_{est}) = \varepsilon$ for some ε with $|\varepsilon| \ll 1$. We may not be able to evaluate the difference $x_0 - x_{est}$ since x_0 is unknown. On the other hand, simply by evaluating f we can compute $f(x_{est}) - f(x_0) \equiv f(x_{est})$ since we know $f(x_0) = 0$ by definition. This value gives some notion of error for our calculation.

This example illustrates the distinction between *forward* and *backward* error. The forward error made by an approximation most likely defines our intuition for error analysis as the difference between the approximated and actual solution, but as we have discussed it is not always possible to compute. The backward error, however, has the distinguishably of being calculable but not our exact objective when solving a given problem. We can adjust our definition and interpretation of backward error as we approach different problems, but one suitable if vague definition is as follows:

Definition 1.3 (Backward error)**.** Backward error *is given by the amount a problem statement would have to change to realize a given approximation of its solution.*

This definition is somewhat obtuse, so we illustrate its use in a few examples.

Example 1.3 (Linear systems)**.** *Suppose we wish to solve the $n \times n$ linear system $A\vec{x} = \vec{b}$. Call the true solution $\vec{x}_0 \equiv A^{-1}\vec{b}$. In reality, due to truncation error and other issues, our system yields a near-solution $\vec{x}_{est}$. The forward error of this approximation obviously is measured using the difference $\vec{x}_{est} - \vec{x}_0$; in practice this value is impossible to compute since we do not know $\vec{x}_0$. In reality, $\vec{x}_{est}$ is the* exact *solution to a modified system $A\vec{x} = \vec{b}_{est}$ for $\vec{b}_{est} \equiv A\vec{x}_{est}$; thus, we might measure backward error in terms of the difference $\vec{b} - \vec{b}_{est}$. Unlike the forward error, this error is easily computable without inverting A, and it is easy to see that $\vec{x}_{est}$ is a solution to the problem exactly when backward (or forward) error is zero.*

Example 1.4 (Solving equations, CITE)**.** *Suppose we write a function for finding square roots of positive numbers that outputs $\sqrt{2} \approx 1.4$. The forward error is $|1.4 - 1.41421\cdots| \approx 0.0142$. Notice that $1.4^2 = 1.96$, so the backward error is $|1.96 - 2| = 0.04$.*

The two examples above demonstrate a larger pattern that backward error can be much easier to compute than forward error. For example, evaluating forward error in Example 1.3 required inverting a matrix A while evaluating backward error required only multiplication by A. Similarly, in Example 1.4 transitioning from forward error to backward error replaced square root computation with multiplication.

1.2.2 Conditioning, Stability, and Accuracy

In nearly any numerical problem, zero backward error implies zero forward error and vice versa. Thus, a piece of software designed to solve such a problem surely can terminate if it finds that a candidate solution has zero backward error. But what if backward error is nonzero but small? Does this necessarily imply small forward error? Such questions motivate the analysis of most numerical techniques whose objective is to minimize forward error but in practice only can measure backward error.

We desire to analyze changes in backward error relative to forward error so that our algorithms can say with confidence using only backward error that they have produced acceptable solutions. This relationship can be different for each problem we wish to solve, so in the end we make the following rough classification:

- A problem is *insensitive* or *well-conditioned* when small amounts of backward error imply small amounts of forward error. In other words, a small perturbation to the statement of a well-conditioned problem yields only a small perturbation of the true solution.
- A problem is *sensitive* or *poorly-conditioned* when this is not the case.

Example 1.5 ($ax = b$)**.** *Suppose as a toy example that we want to find the solution $x_0 \equiv {}^{b}/_{a}$ to the linear equation $ax = b$ for $a, x, b \in \mathbb{R}$. Forward error of a potential solution x is given by $x - x_0$ while backward error is given by $b - ax = a(x - x_0)$. So, when $|a| \gg 1$, the problem is well-conditioned since small values of backward error $a(x - x_0)$ imply even smaller values of $x - x_0$; contrastingly, when $|a| \ll 1$ the problem is ill-conditioned, since even if $a(x - x_0)$ is small the forward error $x - x_0 \equiv {}^{1}/_{a} \cdot a(x - x_0)$ may be large given the ${}^{1}/_{a}$ factor.*

We define the *condition number* to be a measure of a problem's sensitivity:

Definition 1.4 (Condition number)**.** *The* condition number *of a problem is the ratio of how much its solution changes to the amount its statement changes under small perturbations. Alternatively, it is the ratio of forward to backward error for small changes in the problem statement.*

Example 1.6 ($ax = b$, part two)**.** *Continuing Example 1.5, we can compute the condition number exactly:*

$$c = \frac{\textit{forward error}}{\textit{backward error}} = \frac{x - x_0}{a(x - x_0)} \equiv \frac{1}{a}$$

In general, computing condition numbers is nearly as hard as computing forward error, and thus their exact computation is likely impossible. Even so, many times it is possible to find bounds or approximations for condition numbers to help evaluate how much a solution can be trusted.

Example 1.7 (Root-finding)**.** *Suppose that we are given a smooth function $f : \mathbb{R} \to \mathbb{R}$ and want to find values x with $f(x) = 0$. Notice that $f(x + \Delta) \approx f(x) + \Delta f'(x)$. Thus, an approximation of the condition number for finding x might be*

$$\begin{aligned}\frac{\textit{change in forward error}}{\textit{change in backward error}} &= \frac{(x+\Delta) - x}{f(x+\Delta) - f(x)} \\ &\approx \frac{\Delta}{\Delta f'(x)} \\ &= \frac{1}{f'(x)}\end{aligned}$$

Notice that this approximation aligns with the one in Example 1.6. Of course, if we do not know x we cannot evaluate $f'(x)$, but if we can look at the form of f and bound *$|f'|$ near x, we have an idea of the worst-case situation.*

Forward and backward error are measures of the *accuracy* of a solution. For the sake of scientific repeatability, we also wish to derive *stable* algorithms that produce self-consistent solutions to a class of problems. For instance, an algorithm that generates very accurate solutions only one fifth of the time might not be worth implementing, even if we can go back using the techniques above to check whether the candidate solution is a good one.

1.3 Practical Aspects

The infinitude and density of the real numbers $\mathbb{R}$ can cause pernicious bugs while implementing numerical algorithms. While the theory of error analysis introduced in §1.2 eventually will help us put guarantees on the quality of numerical techniques introduced in future chapters, it is worth noting before we proceed a number of common mistakes and "gotchas" that pervade implementations of numerical methods.

We purposefully introduced the largest offender early in §1.1, which we repeat in a larger font for well-deserved emphasis:

Rarely if ever should the operator == and its equivalents be used on fractional values.

Finding a suitable replacement for == and corresponding conditions for terminating a numerical method depends on the technique under consideration. Example 1.3 shows that a method for solving $A\vec{x} = \vec{b}$ can terminate when the residual $\vec{b} - A\vec{x}$ is zero; since we do not want to check if `A*x==b` explicitly, in practice implementations will check `norm(A*x-b)<epsilon`. Notice that this example demonstrates two techniques:

- The use of *backward* error $\vec{b} - A\vec{x}$ rather than forward error to determine when to terminate, and
- Checking whether backward error is less than `epsilon` to avoid the forbidden `==0` predicate.

The parameter `epsilon` depends on how accurate the desired solution must be as well as the resolution of the numerical system at use.

A programmer making use of these data types and operations must be vigilant when it comes to detecting and preventing poor numerical operations. For example, consider the following code snippet for computing the norm $\|\vec{x}\|_2$ for a vector $\vec{x} \in \mathbb{R}^n$ represented as a 1D array `x[]`:

```
double normSquared = 0;
for (int i = 0; i < n; i++)
        normSquared += x[i]*x[i];
return sqrt(normSquared);
```

It is easy to see that in theory $\min_i |x_i| \leq \|\vec{x}\|_2/\sqrt{n} \leq \max_i |x_i|$, that is, the norm of $\vec{x}$ is on the order of the values of elements contained in $\vec{x}$. Hidden in the computation of $\|\vec{x}\|_2$, however, is the expression `x[i]*x[i]`. If there exists `i` such that `x[i]` is on the order of `DOUBLE_MAX`, the product `x[i]*x[i]` will overflow even though $\|\vec{x}\|_2$ is still within the range of the `double`s. Such overflow is easily preventable by dividing $\vec{x}$ by its maximum value, computing the norm, and multiplying back:

```
double maxElement = epsilon; // don't want to divide by zero!
for (int i = 0; i < n; i++)
        maxElement = max(maxElement, fabs(x[i]));
for (int i = 0; i < n; i++) {
        double scaled = x[i] / maxElement;
        normSquared += scaled*scaled;
}
return sqrt(normSquared) * maxElement;
```

The scaling factor removes the overflow problem by making sure that elements being summed are no larger than 1.

This small example shows one of many circumstances in which a single *character* of code can lead to a non-obvious numerical issue. While our intuition from continuous mathematics is sufficient to generate many numerical methods, we must always double-check that the operations we employ are valid from a discrete standpoint.

1.3.1 Larger-Scale Example: Summation

We now provide an example of a numerical issue caused by finite-precision arithmetic that can be resolved using a less than obvious algorithmic trick.

Suppose that we wish to sum a list of floating-point values, easily a task required by systems in accounting, machine learning, graphics, and nearly any other field. A snippet of code to accomplish this task that no-doubt appears in countless applications looks as follows:

```
double sum = 0;
for (int i = 0; i < n; i++)
        sum += x[i];
```

Before we proceed, it is worth noting that for the vast majority of applications, this is a perfectly stable and certainly mathematically valid technique.

But, what can go wrong? Consider the case where `n` is large and most of the values `x[i]` are small and positive. In this case, when `i` is large enough, the variable `sum` will be large relative to `x[i]`. Eventually, `sum` could be so large that `x[i]` affects only the lowest-order bits of `sum`, and in the extreme case `sum` could be large enough that adding `x[i]` has no effect whatsoever. While a single such mistake might not be a big deal, the accumulated effect of making this mistake repeatedly could overwhelm the amount that we can trust `sum` at all.

To understand this effect mathematically, suppose that computing a sum $a + b$ can be off by as much as $\varepsilon > 0$. Then, the method above clearly can induce error on the order of $n\varepsilon$, which grows linearly with n. In fact, if most elements `x[i]` are on the order of ε, then the sum cannot be trusted *whatsoever*! This is a disappointing result: The error can be as large as the sum itself.

Fortunately, there are many ways to do better. For example, adding the smallest values first might help account for their accumulated effect. Pairwise methods recursively adding pairs of values from `x[]` and building up a sum also are more stable, but they can be difficult to implement as efficiently as the `for` loop above. Thankfully, an algorithm by Kahan (CITE) provides an easily-implemented "compensated summation" method that is nearly as fast.

The useful observation here is that we actually can keep track of an approximation of the error in `sum` during a given iteration. In particular, consider the expression

$$((a + b) - a) - b.$$

Obviously this expression algebraically is zero. Numerically, however, this may not be the case. In particular, the sum $(a + b)$ may round the result to keep it within the realm of floating-point values. Subtracting a and b one-at-a-time then yields an approximation of the error induced by this operation; notice that the subtraction operations likely are better conditioned since moving from large numbers to small ones *adds* digits of precision due to cancellation.

Thus, the Kahan technique proceeds as follows:

```
double sum = 0;
double compensation = 0; // an approximation of the error

for (int i = 0; i < n; i++) {
        // try to add back to both x[i] and the missing part
        double nextTerm = x[i] + compensation;

        // compute the summation result of this iteration
        double nextSum = sum + nextTerm;

        // compute the compensation as the difference between the term you wished
        // to add and the actual result
        compensation = nextTerm - (nextSum - sum);

        sum = nextSum;
}
```

Instead of simply maintaining `sum`, now we keep track of `sum` as well as an approximation `compensation` of the difference between `sum` and the desired value. During each iteration, we attempt to add back this compensation in addition to the current element of `x[]`, and then we recompute `compensation` to account for the latest error.

Analyzing the Kahan algorithm requires more careful bookkeeping than analyzing the simpler incremental technique. You will walk through one derivation of an error expression at the end of this chapter; the final mathematical result will be that error improves from $n\varepsilon$ to $\mathcal{O}(\varepsilon + n\varepsilon^2)$, a considerable improvement when $0 < \varepsilon \ll 1$.

Implementing Kahan summation is straightforward but more than doubles the operation count of the resulting program. In this way, there is an implicit trade-off between speed and accuracy that software engineers must make when deciding which technique is most appropriate.

More broadly, Kahan's algorithm is one of several methods that bypass the accumulation of numerical error during the course of a computation consisting of more than one operation. Other examples include Bresenham's algorithm for rasterizing lines (CITE), which uses only integer arithmetic to draw lines even when they intersect rows and columns of pixels at non-integral locations, and the Fast Fourier Transform (CITE), which effectively uses the binary partition summation trick described above.

1.4 Problems

Problem 1.1. *Here's a problem.*

Part II

Linear Algebra

Chapter 2

Linear Systems and the LU Decomposition

In Chapter 0, we discussed a variety of situations in which linear systems of equations $A\vec{x} = \vec{b}$ appear in mathematical theory and in practice. In this chapter, we tackle the basic problem head-on and explore numerical methods for solving such systems.

2.1 Solvability of Linear Systems

As introduced in §0.3.3, systems of linear equations like

$$\begin{aligned} 3x + 2y &= 6 \\ -4x + y &= 7 \end{aligned}$$

can be written in matrix form as in

$$\begin{pmatrix} 3 & 2 \\ -4 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 6 \\ 7 \end{pmatrix}.$$

More generally, we can write systems of the form $A\vec{x} = \vec{b}$ for $A \in \mathbb{R}^{m\times n}$, $\vec{x} \in \mathbb{R}^n$, and $\vec{b} \in \mathbb{R}^m$.

The solvability of the system must fall into one of three cases:

1. The system may not admit any solutions, as in:

$$\begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} -1 \\ 1 \end{pmatrix}.$$

   This system asks that $x = -1$ and $x = 1$ simultaneously, obviously two incompatible conditions.

2. The system may admit a single solution; for instance, the system at the beginning of this section is solved by $(x, y) = (-8/11, 45/11)$.

3. The system may admit infinitely many solutions, e.g. $0\vec{x} = \vec{0}$. Notice that if a system $A\vec{x} = \vec{b}$ admits two solutions $\vec{x}_0$ and $\vec{x}_1$, then it automatically has infinitely many solutions of the form $c\vec{x}_0 + (1-c)\vec{x}_1$ for $c \in \mathbb{R}$, since

$$A(c\vec{x}_0 + (1-c)\vec{x}_1) = cA\vec{x}_0 + (1-c)A\vec{x}_1 = c\vec{b} + (1-c)\vec{b} = \vec{b}.$$

   This linear system would be labeled *underdetermined*.

In general, the solvability of a system depends both on A and on $\vec{b}$. For instance, if we modify the unsolvable system above to be

$$\begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix},$$

then the system moves from having no solutions to infinitely many of the form $(1, y)$. In fact, every matrix A admits a right hand side $\vec{b}$ such that $A\vec{x} = \vec{b}$ is solvable, since $A\vec{x} = \vec{0}$ always can be solved by $\vec{x} \equiv \vec{0}$ regardless of A. Recall from §0.3.1 that matrix-vector multiplication can be viewed as linearly combining the columns of A with weights from $\vec{x}$. Thus, as mentioned in §0.3.3, we can expect $A\vec{x} = \vec{b}$ to be solvable exactly when $\vec{b}$ is in the column space of A.

In a broad way, the "shape" of the matrix $A \in \mathbb{R}^{m\times n}$ has considerable bearing on the solvability of $A\vec{x} = \vec{b}$. Recall that the columns of A are m-dimensional vectors. First, consider the case when A is "wide," that is, when it has more columns than rows ($n > m$). Each column is a vector in $\mathbb{R}^m$, so at most the column space can have dimension m. Since $n > m$, the n columns of A must then be linearly dependent; this implies that there exists an $\vec{x}_0 \neq \vec{0}$ such that $A\vec{x}_0 = \vec{0}$. Then, if we can solve $A\vec{x} = \vec{b}$ for $\vec{x}$, then $A(\vec{x} + \alpha\vec{x}_0) = A\vec{x} + \alpha A\vec{x}_0 = \vec{b} + \vec{0} = \vec{b}$, showing that there are actually infinitely many solutions $\vec{x}$ to $A\vec{x} = \vec{b}$. In other words, we have shown that *no wide matrix system admits a unique solution.*

When A is "tall," that is, when it has more rows than columns ($m > n$), then the n columns cannot span $\mathbb{R}^m$. Thus, there exists some vector $\vec{b}_0 \in \mathbb{R}^m \backslash \text{col } A$. By definition, this $\vec{b}_0$ cannot satisfy $A\vec{x} = \vec{b}_0$ for *any* $\vec{x}$. In other words, *every tall matrix A admits systems $A\vec{x} = \vec{b}_0$ that are not solvable.*

Both of the situations above are far from favorable for designing numerical algorithms. For example, if a linear system admits many solutions we must first define *which* solution is desired from the user: after all, the solution $\vec{x} + 10^{31}\vec{x}_0$ might not be as meaningful as $\vec{x} - 0.1\vec{x}_0$. On the flip side, in the tall case even if $A\vec{x} = \vec{b}$ is solvable for a particular $\vec{b}$, any small perturbation $A\vec{x} = \vec{b} + \varepsilon\vec{b}_0$ is no longer solvable; this situation can appear simply because rounding procedures discussed in the last chapter can only approximate A and $\vec{b}$ in the first place.

Given these complications, in this chapter we will make some simplifying assumptions:

- We will consider only *square* $A \in \mathbb{R}^{n\times n}$.
- We will assume that A is *nonsingular*, that is, that $A\vec{x} = \vec{b}$ is solvable for any $\vec{b}$.

Recall from §0.3.3 that the nonsingularity condition is equivalent to asking that the columns of A span $\mathbb{R}^n$ and implies the existence of a matrix A^{-1} satisfying $A^{-1}A = AA^{-1} = I_{n\times n}$.

A misleading observation is to think that solving $A\vec{x} = \vec{b}$ is equivalent to computing the matrix A^{-1} explicitly and then multiplying to find $\vec{x} \equiv A^{-1}\vec{b}$. While this solution strategy certainly is valid, it can represent a considerable amount of overkill: after all, we're only interested in the n values in $\vec{x}$ rather than the n^2 values in A^{-1}. Furthermore, even when A is well-behaved it can be the case that writing A^{-1} yields numerical difficulties that can be bypassed.

2.2 Ad-Hoc Solution Strategies

In introductory algebra, we often approach the problem of solving a linear system of equations as an art form. The strategy is to "isolate" variables, iteratively writing alternative forms of the linear system until each line is of the form $x = const$.

In formulating systematic algorithms for solving linear systems, it is instructive to carry out an example of this solution process. Consider the following system:

$$\begin{aligned} y - z &= -1 \\ 3x - y + z &= 4 \\ x + y - 2z &= -3 \end{aligned}$$

In parallel, we can maintain a matrix version of this system. Rather than writing out $A\vec{x} = \vec{b}$ explicitly, we can save a bit of space by writing the "augmented" matrix below:

$$\left(\begin{array}{ccc|c} 0 & 1 & -1 & -1 \\ 3 & -1 & 1 & 4 \\ 1 & 1 & -2 & -3 \end{array}\right)$$

We can always write linear systems this way so long as we agree that the variables remain on the left hand side of the equations and the constants on the right.

Perhaps we wish to deal with the variable x first. For convenience, we may *permute* the rows of the system so that the third equation appears first:

$$\begin{aligned} x + y - 2z &= -3 \\ y - z &= -1 \\ 3x - y + z &= 4 \end{aligned} \qquad \left(\begin{array}{ccc|c} 1 & 1 & -2 & -3 \\ 0 & 1 & -1 & -1 \\ 3 & -1 & 1 & 4 \end{array}\right)$$

We can then *substitute* the first equation into the third to eliminate the $3x$ term. This is the same as scaling the relationship $x + y - 2z = -3$ by -3 and adding the result to the third equation:

$$\begin{aligned} x + y - 2z &= -3 \\ y - z &= -1 \\ -4y + 7z &= 13 \end{aligned} \qquad \left(\begin{array}{ccc|c} 1 & 1 & -2 & -3 \\ 0 & 1 & -1 & -1 \\ 0 & -4 & 7 & 13 \end{array}\right)$$

Similarly, to eliminate y from the third equation we can multiply the second equation by 4 and add the result to the third:

$$\begin{aligned} x + y - 2z &= -3 \\ y - z &= -1 \\ 3z &= 9 \end{aligned} \qquad \left(\begin{array}{ccc|c} 1 & 1 & -2 & -3 \\ 0 & 1 & -1 & -1 \\ 0 & 0 & 3 & 9 \end{array}\right)$$

We have now isolated z! Thus, we can scale the third row by $1/3$ to yield an expression for z:

$$\begin{aligned} x + y - 2z &= -3 \\ y - z &= -1 \\ z &= 3 \end{aligned} \qquad \left(\begin{array}{ccc|c} 1 & 1 & -2 & -3 \\ 0 & 1 & -1 & -1 \\ 0 & 0 & 1 & 3 \end{array}\right)$$

Now, we can substitute $z = 3$ into the other two equations to remove z from all but the final row:

$$\begin{aligned} x + y &= 3 \\ y &= 2 \\ z &= 3 \end{aligned} \qquad \left(\begin{array}{ccc|c} 1 & 1 & 0 & 3 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 3 \end{array}\right)$$

Finally we make a similar substitution for y to complete the solve:

$$\begin{aligned} x &= 1 \\ y &= 2 \\ z &= 3 \end{aligned} \qquad \left(\begin{array}{ccc|c} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 3 \end{array}\right)$$

This example might be somewhat pedantic, but looking back on our strategy yields a few important observations about how we can go about solving linear systems:

- We wrote successive systems $A_i\vec{x} = \vec{b}_i$ that can be viewed as simplifications of the original $A\vec{x} = \vec{b}$.
- We solved the system without ever writing down A^{-1}.
- We repeatedly used a few simple operations: scaling, adding, and permuting rows of the system.
- The same operations were applied to A and $\vec{b}$. If we scaled the k-th row of A, we also scaled the k-th row of $\vec{b}$. If we added rows k and ℓ of A, we added rows k and ℓ of $\vec{b}$.
- Less obviously, the steps of the solve did not depend on $\vec{b}$. That is, all of our decisions for how to solve were motivated by eliminating nonzero values in A rather than examining values in $\vec{b}$; $\vec{b}$ just came along for the ride.
- We terminated when we reduced the system to $I_{n\times n}\vec{x} = \vec{b}$.

We will use all of these general observations about solving linear systems to our advantage.

2.3 Encoding Row Operations

Looking back at the example in §2.2, we see that solving the linear system really only involved applying three operations: permutation, row scaling, and adding the scale of one row to another. In fact, we can solve *any* linear system this way, so it is worth exploring these operations in more detail.

2.3.1 Permutation

Our first step in §2.2 was to swap two of the rows in the system of equations. More generally, we might index the rows of a matrices using the numbers $1, \ldots, m$. Then, a *permutation* of those rows can be written as a function σ such that the list $\sigma(1), \ldots, \sigma(m)$ covers the same set of indices.

If $\vec{e}_k$ is the k-th standard basis function, then it is easy to see that the product $\vec{e}_k^\top A$ yields the k-th row of the matrix A. Thus, we can "stack" or concatenate these row vectors vertically to yield a matrix permuting the rows according to σ:

$$P_\sigma \equiv \begin{pmatrix} - & \vec{e}_{\sigma(1)}^\top & - \\ - & \vec{e}_{\sigma(2)}^\top & - \\ & \cdots & \\ - & \vec{e}_{\sigma(m)}^\top & - \end{pmatrix}$$

That is, the product $P_\sigma A$ is exactly the matrix A with rows permuted according to σ.

Example 2.1 (Permutation matrices)**.** *Suppose we wish to permute rows of a matrix in* $\mathbb{R}^{3\times 3}$ *with* $\sigma(1) = 2$*,* $\sigma(2) = 3$*, and* $\sigma(3) = 1$*. According to our formula we would have*

$$P_\sigma = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$

From Example 2.1, we can see that P_σ has ones in positions of the form $(k, \sigma(k))$ and zeros elsewhere. The pair $(k, \sigma(k))$ represents the statement, "We would like row k of the output matrix to be row $\sigma(k)$ from the input matrix." Based on this description of a permutation matrix, it is easy to see that the inverse of P_σ is the transpose $P_\sigma^\top$, since this simply swaps the roles of the rows and columns – now we take row $\sigma(k)$ of the *input* and put it in row k of the *output*. In other words, $P_\sigma^\top P_\sigma = I_{m\times m}$.

2.3.2 Row Scaling

Suppose we write down a list of constants $a_1, \ldots, a_m$ and seek to scale the k-th row of some matrix A by a_k. This is obviously accomplished by applying the scaling matrix S_a given by:

$$S_a \equiv \begin{pmatrix} a_1 & 0 & 0 & \cdots \\ 0 & a_2 & 0 & \cdots \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_m \end{pmatrix}$$

Assuming that all a_k satisfy $a_k \neq 0$, it is easy to invert S_a by "scaling back:"

$$S_a^{-1} = S_{1/a} \equiv \begin{pmatrix} 1/a_1 & 0 & 0 & \cdots \\ 0 & 1/a_2 & 0 & \cdots \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1/a_m \end{pmatrix}$$

2.3.3 Elimination

Finally, suppose we wish to scale row k by a constant c and add the result to row ℓ. This operation may seem less natural than the previous two but actually is quite practical: It is the only one we need to combine equations from different rows of the linear system! We will realize this operation using an "elimination matrix" M such that the product MA applies this operation to matrix A.

Recall that the product $\vec{e}_k^\top A$ picks out the k-th row of A. Then, premultiplying by $\vec{e}_\ell$ yields a matrix $\vec{e}_\ell \vec{e}_k^\top A$, which is zero except the ℓ-th row is equal to the k-th of A.

Example 2.2 (Elimination matrix construction)**.** *Take*

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

Suppose we wish to isolate the third row of $A \in \mathbb{R}^{3\times 3}$ and move it to row two. As discussed above, this operation is accomplished by writing:

$$\begin{aligned} \vec{e}_2 \vec{e}_3^\top A &= \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \begin{pmatrix} 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \\ &= \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \begin{pmatrix} 7 & 8 & 9 \end{pmatrix} \\ &= \begin{pmatrix} 0 & 0 & 0 \\ 7 & 8 & 9 \\ 0 & 0 & 0 \end{pmatrix} \end{aligned}$$

Of course, we multiplied right-to-left above but just as easily could have grouped the product as $(\vec{e}_2 \vec{e}_3^\top)A$. The structure of this product is easy to see:

$$\vec{e}_2 \vec{e}_3^\top = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \begin{pmatrix} 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

We have succeeded in isolating row k and moving it to row ℓ. Our original elimination operation wanted to add c times row k to row ℓ, which we can now accomplish as the sum $A + c\vec{e}_\ell \vec{e}_k^\top A = (I_{n\times n} + c\vec{e}_\ell \vec{e}_k^\top)A$.

Example 2.3 (Solving a system)**.** *We can now encode each of our operations from Section 2.2 using the matrices we have constructed above:*

1. *Permute the rows to move the third equation to the first row:*

$$P = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

2. *Scale row one by -3 and add the result to row three: $E_1 = I_{3\times 3} - 3\vec{e}_3 \vec{e}_1^\top$*

3. *Scale row two by 4 and add the result to row three: $E_2 = I_{3\times 3} + 4\vec{e}_3 \vec{e}_2^\top$*

4. *Scale row three by $1/3$: $S = diag(1, 1, 1/3)$*

5. *Scale row three by 2 and add it to row one:* $E_3 = I_{3\times 3} + 2\vec{e}_1\vec{e}_3^\top$
6. *Add row three to row two:* $E_4 = I_{3\times 3} + \vec{e}_2\vec{e}_3^\top$
7. *Scale row three by -1 and add the result to row one:* $E_5 = I_{3\times 3} - \vec{e}_1\vec{e}_3^\top$

Thus, the inverse of A in Section 2.2 satisfies

$$A^{-1} = E_5E_4E_3SE_2E_1P.$$

Make sure you understand why these matrices appear in reverse *order!*

2.4 Gaussian Elimination

The sequence of steps chosen in Section 2.2 was by no means unique: There are many different paths that can lead to the solution of $A\vec{x} = \vec{b}$. Our steps, however, followed the strategy of *Gaussian elimination*, a famous algorithm for solving linear systems of equations.

More generally, let's say our system has the following "shape:"

$$\left(\begin{array}{c|c} A & \vec{b} \end{array}\right) = \left(\begin{array}{cccc|c} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{array}\right)$$

The algorithm proceeds in phases described below.

2.4.1 Forward Substitution

Consider the upper-left element of our matrix:

$$\left(\begin{array}{c|c} A & \vec{b} \end{array}\right) = \left(\begin{array}{cccc|c} \textcircled{\times} & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{array}\right)$$

We will call this element our first *pivot* and will assume it is nonzero; if it is zero we can permute rows so that this is not the case. We first apply a scaling matrix so that the pivot equals one:

$$\left(\begin{array}{cccc|c} \textcircled{1} & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{array}\right)$$

Now, we use the row containing the pivot to eliminate all other values underneath in the same column:

$$\left(\begin{array}{cccc|c} \textcircled{1} & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \end{array}\right)$$

We now move our pivot to the next row and repeat a similar series of operations:

$$\left(\begin{array}{cccc|c} 1 & \times & \times & \times & \times \\ 0 & \textcircled{1} & \times & \times & \times \\ 0 & 0 & \times & \times & \times \\ 0 & 0 & \times & \times & \times \end{array}\right)$$

Notice that a nice thing happens here. After the first pivot has been eliminated from all other rows, the first column is zero beneath row 1. This means we can safely add multiples of row two to rows underneath without affecting the zeros in column one.

We repeat this process until the matrix becomes *upper-triangular*:

$$\left(\begin{array}{cccc|c} 1 & \times & \times & \times & \times \\ 0 & 1 & \times & \times & \times \\ 0 & 0 & 1 & \times & \times \\ 0 & 0 & 0 & \textcircled{1} & \times \end{array}\right)$$

2.4.2 Back Substitution

Eliminating the remaining $\times$'s from the system is now a straight forward process. Now, we proceed in *reverse* order of rows and eliminate backward. For instance, after the first series of back substitution steps, we are left with the following shape:

$$\left(\begin{array}{cccc|c} 1 & \times & \times & 0 & \times \\ 0 & 1 & \times & 0 & \times \\ 0 & 0 & 1 & 0 & \times \\ 0 & 0 & 0 & \textcircled{1} & \times \end{array}\right)$$

Similarly, the second iteration yields:

$$\left(\begin{array}{cccc|c} 1 & \times & 0 & 0 & \times \\ 0 & 1 & 0 & 0 & \times \\ 0 & 0 & \textcircled{1} & 0 & \times \\ 0 & 0 & 0 & 1 & \times \end{array}\right)$$

After our final elimination step, we are left with our desired form:

$$\left(\begin{array}{cccc|c} \textcircled{1} & 0 & 0 & 0 & \times \\ 0 & 1 & 0 & 0 & \times \\ 0 & 0 & 1 & 0 & \times \\ 0 & 0 & 0 & 1 & \times \end{array}\right)$$

The right hand side now is the solution to the linear system $A\vec{x} = \vec{b}$.

2.4.3 Analysis of Gaussian Elimination

Each row operation in Gaussian elimination – scaling, elimination, and swapping two rows – obviously takes $O(n)$ time to complete, since you have to iterate over all n elements of a row (or two) of A. Once we choose a pivot, we have to do n forward- or back- substitutions into the rows below or above that pivot, resp.; this means the work for a single pivot in total is $O(n^2)$. In total, we choose one pivot per row, adding a final factor of n. Thus, it is easy enough to see that Gaussian elimination runs in $O(n^3)$ time.

One decision that takes place during Gaussian elimination that we have not discussed is the choice of *pivots*. Recall that we can permute rows of the linear system as we see fit before performing back- or forward- substitution. This operation is *necessary* to be able to deal with all possible matrices A. For example, consider what would happen if we did not use pivoting on the following matrix:

$$A = \begin{pmatrix} \textcircled{0} & 1 \\ 1 & 0 \end{pmatrix}$$

Notice that the circled element is exactly zero, so we cannot expect to divide row one by any number to replace that 0 with a 1. This does *not* mean the system is not solvable, it just means we must do *pivoting*, accomplished by swapping the first and second rows, to put a nonzero in that slot.

More generally, suppose our matrix looks like:

$$A = \begin{pmatrix} \textcircled{\varepsilon} & 1 \\ 1 & 0 \end{pmatrix},$$

where $0 < \varepsilon \ll 1$. If we do not pivot, then the first iteration of Gaussian elimination yields:

$$\tilde{A} = \begin{pmatrix} \textcircled{1} & 1/\varepsilon \\ 0 & -1/\varepsilon \end{pmatrix},$$

We have transformed a matrix A that looks nearly like a permutation matrix (in fact, $A^{-1} \approx A^\top$, a very easy way to solve the system!) into a system with potentially **huge** values $1/\varepsilon$.

This example shows that there are cases when we may wish to pivot even when doing so strictly speaking is not necessary. Since we are scaling by the reciprocal of the pivot value, clearly the most numerically stable options is to have a *large* pivot: Small pivots have large reciprocals, scaling numbers to large values in regimes that are likely to lose precision. There are two well-known pivoting strategies:

1. *Partial* pivoting looks through the current column and permutes rows of the matrix so that the largest absolute value appears on the diagonal.

2. *Full* pivoting iterates over the **entire** matrix and permutes both rows and columns to get the largest possible value on the diagonal. Notice that permuting columns of a matrix is a valid operation: it corresponds to changing the labeling of the variables in the system, or post-multiplying A by a permutation.

Example 2.4 (Pivoting). *Suppose after the first iteration of Gaussian elimination we are left with the following matrix:*

$$\begin{pmatrix} 1 & 10 & -10 \\ 0 & \textcircled{0.1} & 9 \\ 0 & 4 & 6.2 \end{pmatrix}$$

If we implement partial pivoting, then we will look only in the second column and will swap the second and third rows; notice that we leave the 10 *in the first row since that one already has been visited by the algorithm:*

$$\begin{pmatrix} 1 & 10 & -10 \\ 0 & \textcircled{4} & 6.2 \\ 0 & 0.1 & 9 \end{pmatrix}$$

If we implement full pivoting, then we will move the 9*:*

$$\begin{pmatrix} 1 & -10 & 10 \\ 0 & \textcircled{9} & 0.1 \\ 0 & 6.2 & 4 \end{pmatrix}$$

Obviously full pivoting yields the best possible numerics, but the cost is a more expensive search for large elements in the matrix.

2.5 LU Factorization

There are many times when we wish to solve a sequence of problems $A\vec{x}_1 = \vec{b}_1, A\vec{x}_2 = \vec{b}_2, \ldots$. As we already have discussed, the steps of Gaussian elimination for solving $A\vec{x} = \vec{b}_k$ depend mainly on the structure of A rather than the values in a particular $\vec{b}_k$. Since A is kept constant here, we may wish to "remember" the steps we took to solve the system so that each time we are presented with a new $\vec{b}$ we do not have to start from scratch.

Solidifying this suspicion that we can move some of the $O(n^3)$ for Gaussian elimination into precomputation time, recall the *upper triangular* system resulting after the forward substitution stage:

$$\left(\begin{array}{cccc|c} 1 & \times & \times & \times & \times \\ 0 & 1 & \times & \times & \times \\ 0 & 0 & 1 & \times & \times \\ 0 & 0 & 0 & \textcircled{1} & \times \end{array}\right)$$

In fact, solving this system by back-substitution only takes $O(n^2)$ time! Why? Back substitution in this case is far easier thanks to the structure of the zeros in the system. For example, in the first series of back substitutions we obtain the following matrix:

$$\left(\begin{array}{cccc|c} 1 & \times & \times & 0 & \times \\ 0 & 1 & \times & 0 & \times \\ 0 & 0 & 1 & 0 & \times \\ \textcircled{0} & \textcircled{0} & \textcircled{0} & 1 & \times \end{array}\right)$$

Since we know that the (circled) values to the left of the pivot are zero by construction, we don't need to copy them explicitly. Thus, this step only took $O(n)$ time rather than $O(n^2)$ taken by forward substitution.

Now, our next pivot does a similar substitution:

$$\left(\begin{array}{cccc|c} 1 & \times & 0 & 0 & \times \\ 0 & 1 & 0 & 0 & \times \\ \textcircled{0} & \textcircled{0} & 1 & \textcircled{0} & \times \\ 0 & 0 & 0 & 1 & \times \end{array}\right)$$

Once again the zeros on both sides of the 1 do not need to be copied explicitly.

Thus, we have found:

Observation. *While Gaussian elimination takes $O(n^3)$ time, solving triangular systems takes $O(n^2)$ time.*

2.5.1 Constructing the Factorization

Recall from §2.3 that all the operations in Gaussian elimination can be thought of as pre-multiplying $A\vec{x} = \vec{b}$ by different matrices M to obtain an easier system $(MA)\vec{x} = M\vec{b}$. As we demonstrated in Example 2.3, from this standpoint each step of Gaussian elimination represents a system $(M_k \cdots M_2 M_1 A)\vec{x} = M_k \cdots M_2 M_1 \vec{b}$. Of course, explicitly storing these matrices M_k as $n \times n$ objects is overkill, but keeping this interpretation in mind from a theoretical perspective simplifies many of our calculations.

After the forward substitution phase of Gaussian elimination, we are left with an *upper triangular* matrix, which we can call $U \in \mathbb{R}^{n\times n}$. From the matrix multiplication perspective, we can write:

$$M_k \cdots M_1 A = U,$$

or, equivalently,

$$\begin{aligned} A &= (M_k \cdots M_1)^{-1} U \\ &= (M_1^{-1} M_2^{-1} \cdots M_k^{-1}) U \\ &\equiv LU, \text{ if we make the definition } L \equiv M_1^{-1} M_2^{-1} \cdots M_k^{-1}. \end{aligned}$$

We don't know anything about the structure of L yet, but we do know that systems of the form $U\vec{y} = \vec{d}$ are easier to solve since U is upper triangular. If L is equally nice, we could solve $A\vec{x} = \vec{b}$ in two steps, by writing $(LU)\vec{x} = \vec{b}$, or $\vec{x} = U^{-1}L^{-1}\vec{b}$:

1. Solve $L\vec{y} = \vec{b}$ for $\vec{y}$, yielding $\vec{y} = L^{-1}\vec{b}$.
2. Now that we have $\vec{y}$, solve $U\vec{x} = \vec{y}$, yielding $\vec{x} = U^{-1}\vec{y} = U^{-1}(L^{-1}\vec{b}) = (LU)^{-1}\vec{b} = A^{-1}\vec{b}$. We already know that this step only takes $O(n^2)$ time.

Our remaining task is to make sure that L has nice structure that will make solving $L\vec{y} = \vec{b}$ easier than solving $A\vec{x} = \vec{b}$. Thankfully–and unsurprisingly–we will find that L is *lower triangular* and thus can be solved using $O(n^2)$ forward substitution.

To see this, suppose for now that we do not implement pivoting. Then, each of our matrices M_k is either a scaling matrix or has the structure

$$M_k = I_{n\times n} + c\vec{e}_\ell \vec{e}_k^\top,$$

where $\ell > k$ since we only have carried out forward substitution. Remember that this matrix serves a specific purpose: Scale row k by c and add the result to row ℓ. This operation obviously is easy to undo: Scale row k by c and *subtract* the result from row ℓ. We can check this formally:

$$\begin{aligned} (I_{n\times n} + c\vec{e}_\ell \vec{e}_k^\top)(I_{n\times n} - c\vec{e}_\ell \vec{e}_k^\top) &= I_{n\times n} + (-c\vec{e}_\ell \vec{e}_k^\top + c\vec{e}_\ell \vec{e}_k^\top) - c^2 \vec{e}_\ell \vec{e}_k^\top \vec{e}_\ell \vec{e}_k^\top \\ &= I_{n\times n} - c^2 \vec{e}_\ell (\vec{e}_k^\top \vec{e}_\ell) \vec{e}_k^\top \\ &= I_{n\times n} \text{ since } \vec{e}_k^\top \vec{e}_\ell = \vec{e}_k \cdot \vec{e}_\ell, \text{ and } k \neq \ell \end{aligned}$$

So, the L matrix is the product of scaling matrices and matrices of the form $M^{-1} = I_{n\times n} + c\vec{e}_\ell\vec{e}_k^\top$ are lower triangular when $\ell > k$. Scaling matrices are diagonal, and the matrix M is lower triangular. You will show in Exercise 2.1 that the product of lower triangular matrices is lower triangular, showing in turn that L is lower triangular as needed.

We have shown that if it is possible to carry out Gaussian elimination of A without using pivoting, you can factor $A = LU$ into the product of lower- and upper-triangular matrices. Forward and back substitution each take $O(n^2)$ time, so if this factorization can be computed ahead of time the linear solve can be carried out faster than full $O(n^3)$ Gaussian elimination. You will show in Exercise 2.2 what happens when we carry out LU with pivoting; no major changes are necessary.

2.5.2 Implementing LU

A simple implementation of Gaussian elimination to solve $A\vec{x} = \vec{b}$ is straightforward enough to formulate. In particular, as we have discussed earlier, we can form the augmented matrix $(A \mid \vec{b})$ and apply row operations one at a time to this $n \times (n+1)$ block until it looks like $(I_{n\times n}A^{-1} \mid \vec{b})$. This process, however, is *destructive*, that is, in the end we care only about the last column of the augmented matrix and have kept no evidence of our solution path. Such behavior clearly is not acceptable for LU factorization.

Let's examine what happens when we multiply two elimination matrices:

$$(I_{n\times n} - c_\ell\vec{e}_\ell\vec{e}_k^\top)(I_{n\times n} - c_p\vec{e}_p\vec{e}_k^\top) = I_{n\times n} - c_\ell\vec{e}_\ell\vec{e}_k^\top - c_p\vec{e}_p\vec{e}_k^\top$$

As in our construction of the inverse of an elimination matrix, the product of the two $\vec{e}_i$ terms vanishes since the standard basis is orthogonal. This formula shows that after the pivot is scaled to 1, the product of the elimination matrices used to forward substitute for that pivot has the form:

$$M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \textcircled{1} & 0 & 0 \\ 0 & \times & 1 & 0 \\ 0 & \times & 0 & 1 \end{pmatrix},$$

where the values $\times$ are the values used to eliminate the rest of the column. Multiplying matrices of this form together shows that the elements beneath the diagonal of L just come from coefficients used to accomplish substitution.

We can make one final decision to keep the elements along the diagonal of L in the LU factorization equal to 1. This decision is a legitimate one, since we can always post-multiply a L by a scaling matrix S taking these elements to 1 and write $LU = (LS)(S^{-1}U)$ without affecting the triangular pattern of L or U. With this decision in place, we can compress our storage of *both* L and U into a single $n \times n$ matrix whose upper triangle is U and which is equal to L beneath the diagonal; the missing diagonal elements of L are all 1.

We are now ready to write pseudocode for the simplest LU factorization strategy in which we do not permute rows or columns to obtain pivots:

```
// Takes as input an n-by-n matrix A[i,j]
// Edits A in place to obtain the compact LU factorization described above

for pivot from 1 to n {
    pivotValue = A[pivot,pivot]; // Bad assumption that this is nonzero!
    for eliminationRow from (pivot+1) to n { // Eliminate values beneath the pivot
        // How much to scale the pivot row to eliminate the value in the current
        // row; notice we're not scaling the pivot row to 1, so we divide by the
        // pivot value
        scale = A[eliminationRow,pivot]/pivotValue;

        // Since we /subtract/ scale times the pivot row from the current row
        // during Gaussian elimination, we /add/ it in the inverse operation
        // stored in L
        A[eliminationRow,pivot] = scale;

        // Now, as in Gaussian elimination, perform the row operation on the rest
        // of the row:  this will become U
        for eliminationCol from (pivot+1) to n
            A[eliminationRow,eliminationCol] -= A[pivot,eliminationCol]*scale;
    }
}
```

2.6 Problems

Problem 2.1. *Product of lower triangular things is lower triangular; product of pivot matrices looks right*

Problem 2.2. *Implement LU with pivoting*

Problem 2.3. *Non-square LU*

Chapter 3

Designing and Analyzing Linear Systems

Now that we have some methods for solving linear systems of equations, we can use them to solve a variety of problems. In this chapter, we will explore a few such applications and accompanying analytical techniques to characterize the types of solutions we can expect.

3.1 Solution of Square Systems

At the beginning of the previous chapter we made several assumptions on the types of linear systems we were going to solve. While this restriction was a nontrivial one, in fact many if not most applications of linear solvers can be posed in terms of square, invertible linear systems. We explore a few contrasting applications below.

3.1.1 Regression

We will start with a simple application appearing in data analysis known as *regression*. Suppose that we carry out a scientific experiment and wish to understand the structure of our experimental results. One way to model such an relationship might be to write the *independent variables* of the experiment in some vector $\vec{x} \in \mathbb{R}^n$ and to consider the *dependent variable* as a function $f(\vec{x}) : \mathbb{R}^n \to \mathbb{R}$. Our goal is to predict the output of f without carrying out the full experiment.

Example 3.1 (Biological experiment)**.** *Suppose we wish to measure the effect of fertilizer, sunlight, and water on plant growth. We might do a number of experiments applying different amounts of fertilizer (in cm^3), sunlight (in watts), and water (in ml) and measuring the height of the plant after a few days. We might model our observations as a function $f : \mathbb{R}^3 \to \mathbb{R}$ taking in the three parameters we wish to test and outputting the height of the plant.*

In *parametric* regression, we make a simplifying assumption on the structure of f. For example, suppose that f is linear:

$$f(\vec{x}) = a_1x_1 + a_2x_2 + \cdots + a_nx_n.$$

Then, our goal becomes more concrete: to estimate the coefficients a_k.

Suppose we do a series of experiments that show $\vec{x}^{(k)} \mapsto y^{(k)} \equiv f(\vec{x}^{(k)})$. By plugging into our form for f, we obtain a series of statements:

$$\begin{aligned} y^{(1)} &= f(\vec{x}^{(1)}) = a_1 x_1^{(1)} + a_2 x_2^{(1)} + \cdots + a_n x_n^{(1)} \\ y^{(2)} &= f(\vec{x}^{(2)}) = a_1 x_1^{(2)} + a_2 x_2^{(2)} + \cdots + a_n x_n^{(2)} \\ &\vdots \end{aligned}$$

Notice that contrary to our notation $A\vec{x} = \vec{b}$, the unknowns here are the a_k's, *not* the $\vec{x}$ variables. If we make n observations, we can write:

$$\begin{pmatrix} - & \vec{x}^{(1)\top} & - \\ - & \vec{x}^{(2)\top} & - \\ & \vdots & \\ - & \vec{x}^{(n)\top} & - \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(n)} \end{pmatrix}$$

In other words, if we carry out n trials of our experiment and write them in the columns of a matrix $X \in \mathbb{R}^{n\times n}$ and write the dependent variables in a vector $\vec{y} \in \mathbb{R}^n$, the coefficients $\vec{a}$ can be recovered by solving $X^\top \vec{a} = \vec{y}$.

In fact, we can generalize our approach to other more interesting nonlinear forms for the function f. What matters here is that f is a *linear combination* of functions. In particular, suppose $f(\vec{x})$ takes the following form:

$$f(\vec{x}) = a_1 f_1(\vec{x}) + a_2 f_2(\vec{x}) + \cdots + a_m f_m(\vec{x}),$$

where $f_k : \mathbb{R}^n \to \mathbb{R}$ and we wish to estimate the parameters a_k. Then, by a parallel derivation given m observations of the form $\vec{x}^{(k)} \mapsto y^{(k)}$ we can find the parameters by solving:

$$\begin{pmatrix} f_1(\vec{x}^{(1)}) & f_2(\vec{x}^{(1)}) & \cdots & f_m(\vec{x}^{(1)}) \\ f_1(\vec{x}^{(2)}) & f_2(\vec{x}^{(2)}) & \cdots & f_m(\vec{x}^{(2)}) \\ \vdots & \vdots & \cdots & \vdots \\ f_1(\vec{x}^{(m)}) & f_2(\vec{x}^{(m)}) & \cdots & f_m(\vec{x}^{(m)}) \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{pmatrix} = \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{pmatrix}$$

That is, even if the f's are nonlinear, we can learn weights a_k using purely linear techniques.

Example 3.2 (Linear regression)**.** *The system $X^\top \vec{a} = \vec{y}$ can be recovered from the general formulation by taking $f_k(\vec{x}) \equiv x_k$.*

Example 3.3 (Polynomial regression)**.** *Suppose that we observe a function of a single variable $f(x)$ and wish to write it as an n-th degree polynomial*

$$f(x) \equiv a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n.$$

Given n pairs $x^{(k)} \mapsto y^{(k)}$, we can solve for the parameters $\vec{a}$ via the system

$$\begin{pmatrix} 1 & x^{(1)} & (x^{(1)})^2 & \cdots & (x^{(1)})^n \\ 1 & x^{(2)} & (x^{(2)})^2 & \cdots & (x^{(2)})^n \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & x^{(n)} & (x^{(n)})^2 & \cdots & (x^{(n)})^n \end{pmatrix} \begin{pmatrix} a_0 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(n)} \end{pmatrix}.$$

In other words, we take $f_k(x) = x^k$ in our general form above. Incidentally, the matrix on the left hand side of this relationship is known as a Vandermonde matrix, which has many special properties specific to its structure.

Example 3.4 (Oscillation). *Suppose we wish to find a and ϕ for a function $f(x) = a\cos(x+\phi)$. Recall from trigonometry that we can write $\cos(x+\phi) = \cos x \cos\phi - \sin x \sin\phi$. Thus, given two sample points we can use the technique above to find $f(x) = a_1 \cos x + a_2 \sin x$, and applying this identity we can write*

$$\begin{aligned} a &= \sqrt{a_1^2 + a_2^2} \\ \phi &= -\arctan\frac{a_2}{a_1} \end{aligned}$$

This construction can be extended to finding $f(x) = \sum_k a_k \cos(x+\phi_k)$, giving one way to motivate the discrete Fourier transform of f.

3.1.2 Least Squares

The techniques in §3.1.1 provide valuable methods for finding a continuous f matching a set of data pairs $\vec{x}_k \mapsto y_k$ *exactly*. There are two related drawbacks to this approach:

- There might be some error in measuring the values $\vec{x}_k$ and y_k. In this case, an approximate relationship $f(\vec{x}_k) \approx y_k$ may be acceptable or even preferable to an exact $f(\vec{x}_k) = y_k$.
- Notice that if there were m functions f_k total, then we needed exactly m observations $\vec{x}_k \mapsto y_k$. Additional observations would have to be thrown out, or we would have to change the form of f.

Both of these issues related to the larger problem of *over-fitting*: Fitting a function with n degrees of freedom to n data points leaves no room for measurement error.

More generally, suppose we wish to solve the linear system $A\vec{x} = \vec{b}$ for $\vec{x}$. If we denote row k of A as $\vec{r}_k^\top$, then our system looks like

$$\begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} = \begin{pmatrix} - & \vec{r}_1^\top & - \\ - & \vec{r}_2^\top & - \\ \vdots & \cdots & \vdots \\ - & \vec{r}_n^\top & - \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} \vec{r}_1 \cdot \vec{x} \\ \vec{r}_2 \cdot \vec{x} \\ \vdots \\ \vec{r}_n \cdot \vec{x} \end{pmatrix} \text{ by definition of matrix multiplication.}$$

Thus, each row of the system corresponds to an observation of the form $\vec{r}_k \cdot \vec{x} = b_k$. That is, yet another way to interpret the linear system $A\vec{x} = \vec{b}$ is as n statements of the form, "The dot product of $\vec{x}$ with $\vec{r}_k$ is b_k."

From this viewpoint, a tall system $A\vec{x} = \vec{b}$ with $A \in \mathbb{R}^{m\times n}$ and $m > n$ simply encodes more than n of these dot product observations. When we make more than n observations, however, they may be *incompatible*; as explained §2.1, tall systems likely will not admit a solution. In our "experimental" setup explained above, this situation might correspond to errors in the measurement of the pairs $\vec{x}_k \mapsto y_k$.

When we cannot solve $A\vec{x} = \vec{b}$ exactly, we might relax the problem a bit to approximate $A\vec{x} \approx \vec{b}$. In particular, we can ask that the residual $\vec{b} - A\vec{x}$ be as small as possible by minimizing the norm $\|\vec{b} - A\vec{x}\|$. Notice that if there is an exact solution to the linear system, then this norm is minimized at zero, since in this case we have $\|\vec{b} - A\vec{x}\| = \|\vec{b} - \vec{b}\| = 0$. Minimizing $\|\vec{b} - A\vec{x}\|$ is the same as minimizing $\|\vec{b} - A\vec{x}\|^2$, which we expanded in Example 0.16 to:

$$\|\vec{b} - A\vec{x}\|^2 = \vec{x}^\top A^\top A\vec{x} - 2\vec{b}^\top A\vec{x} + \|\vec{b}\|^2.$$[1]

The gradient of this expression with respect to $\vec{x}$ must be zero at its minimum, yielding the following system:

$$\begin{aligned} \vec{0} &= 2A^\top A\vec{x} - 2A^\top\vec{b} \\ \text{Or equivalently:} \qquad A^\top A\vec{x} &= A^\top\vec{b}. \end{aligned}$$

This famous relationship is worthy of a theorem:

Theorem 3.1 (Normal equations)**.** *Minima of the residual $\|\vec{b} - A\vec{x}\|$ for $A \in \mathbb{R}^{m\times n}$ (with no restriction on m or n) satisfy $A^\top A\vec{x} = A^\top\vec{b}$.*

If at least n rows of A are linearly independent, then the matrix $A^\top A \in \mathbb{R}^{n\times n}$ is invertible. In this case, the minimum residual occurs (uniquely) at $(A^\top A)^{-1}A^\top\vec{b}$, or equivalently, solving the least squares problem is as easy as solving the *square* linear system $A^\top A\vec{x} = A^\top\vec{b}$ from Theorem 3.1. Thus, we have expanded our set of solution strategies to $A \in \mathbb{R}^{m\times n}$ with $m \geq n$ by applying only techniques for square matrices.

The underdetermined case $m < n$ is considerably more difficult to deal with. In particular, we lose the possibility of a *unique* solution to $A\vec{x} = \vec{b}$. In this case we must make an additional assumption on $\vec{x}$ to obtain a unique solution, e.g. that it has a small norm or that it contains many zeros. Each such *regularizing* assumption leads to a different solution strategy; we will explore a few in the exercises accompanying this chapter.

3.1.3 Additional Examples

An important skill is to be able to identify linear systems "in the wild." Here we quickly enumerate a few more examples.

Alignment

Suppose we take two photographs of the same scene from different positions. One common task in computer vision and graphics is to stitch them together. To do so, the user (or an automatic system) may mark a number of points $\vec{x}_k, \vec{y}_k \in \mathbb{R}^2$ such that $\vec{x}_k$ in image one corresponds to $\vec{y}_k$ in image two. Of course, likely mistakes were made while matching these points, so we wish to find a stable transformation between the two images by oversampling the number of necessary pairs $(\vec{x}, \vec{y})$.

Assuming our camera has a standard lens, camera projections are linear, so a reasonable assumption is that there exists some $A \in \mathbb{R}^{2\times 2}$ and a translation vector $\vec{b} \in \mathbb{R}^2$ such that

$$\vec{y}_k \approx A\vec{x}_k + \vec{b}.$$

[1]It may be valuable to return to the preliminaries in Chapter 0 at this point for review.

Our unknown variables here are A and $\vec{b}$ rather than $\vec{x}_k$ and $\vec{y}_k$.

In this case, we can find the transformation by solving:

$$\min_{A,\vec{b}} \sum_{k=1}^{p} \|(A\vec{x}_k + \vec{b}) - \vec{y}_k\|^2.$$

This expression is once again a sum of squared linear expressions in our unknowns A and $\vec{b}$, and by a similar derivation to our discussion of the least-squares problem it can be solved linearly.

Deconvolution

Often times we accidentally take photographs that are somewhat out of focus. While a photo that is completely blurred may be a lost cause, if there is localized or small-scale blurring, we may be able to recover a sharper image using computational techniques. One simple strategy is *deconvolution*, explained below.

We can think of a photograph as a point in $\mathbb{R}^p$, where p is the number of pixels; of course, if the photo is in color we may need three values (RGB) per pixel, yielding a similar technique in $\mathbb{R}^{3p}$. Regardless, many simple image blurs are *linear*, e.g. Gaussian convolution or operations averaging pixels with their neighbors on the image. In image processing, these linear operations often have other special properties like shift-invariance, but for our purposes we can think of blurring as some linear operator $\vec{x} \mapsto G * \vec{x}$.

Suppose we take a blurry photo $\vec{x}_0 \in \mathbb{R}^p$. Then, we could try to recover the sharp image $\vec{x} \in \mathbb{R}^p$ by solving the least-squares problem

$$\min_{\vec{x} \in \mathbb{R}^p} \|\vec{x}_0 - G * \vec{x}\|^2.$$

That is, we ask that when you blur $\vec{x}$ with G, you get the observed photo $\vec{x}_0$. Of course, many sharp images might yield the same blurred result under G, so we often add additional terms to the minimization above asking that $\vec{x}_0$ not vary wildly.

3.2 Special Properties of Linear Systems

Our discussion of Gaussian elimination and the LU factorization led to a completely generic method for solving linear systems of equations. While this strategy always works, sometimes we can gain speed or numerical advantages by examining the particular system we are solving. Here we discuss a few common examples where knowing more about the linear system can simplify solution strategies.

3.2.1 Positive Definite Matrices and the Cholesky Factorization

As shown in Theorem 3.1, solving a least-squares problem $A\vec{x} \approx \vec{b}$ yields a solution $\vec{x}$ satisfying the square linear system $(A^\top A)\vec{x} = A^\top \vec{b}$. Regardless of A, the matrix $A^\top A$ has a few special properties that make this system special.

First, it is easy to see that $A^\top A$ is symmetric, since

$$(A^\top A)^\top = A^\top (A^\top)^\top = A^\top A.$$

Here, we simply used the identities $(AB)^\top = B^\top A^\top$ and $(A^\top)^\top = A$. We can express this symmetry index-wise by writing $(A^\top A)_{ij} = (A^\top A)_{ji}$ for all indices i, j. This property implies that it is sufficient to store only the values of $A^\top A$ on or above the diagonal, since the rest of the elements can be obtained by symmetry.

Furthermore, $A^\top A$ is a *positive semidefinite* matrix, as defined below:

Definition 3.1 (Positive (Semi-)Definite)**.** *A matrix $B \in \mathbb{R}^{n\times n}$ is positive semidefinite if for all $\vec{x} \in \mathbb{R}^n$, $\vec{x}^\top B\vec{x} \geq 0$. B is positive definite if $\vec{x}^\top B\vec{x} > 0$ whenever $\vec{x} \neq \vec{0}$.*

It is easy to show that $A^\top A$ is positive semidefinite, since:

$$\vec{x}^\top A^\top A\vec{x} = (A\vec{x})^\top (A\vec{x}) = (A\vec{x}) \cdot (A\vec{x}) = \|A\vec{x}\|_2^2 \geq 0.$$

In fact, if the columns of A are linearly independent, then $A^\top A$ is positive definite.

More generally, suppose we wish to solve a *symmetric positive definite* system $C\vec{x} = \vec{d}$. As we have already explored, we could LU-factorize the matrix C, but in fact we can do somewhat better. We write $C \in \mathbb{R}^{n\times n}$ as a block matrix:

$$C = \begin{pmatrix} c_{11} & \vec{v}^\top \\ \vec{v} & \tilde{C} \end{pmatrix}$$

where $\vec{v} \in \mathbb{R}^{n-1}$ and $\tilde{C} \in \mathbb{R}^{(n-1)\times(n-1)}$. Thanks to the special structure of C, we can make the following observation:

$$\begin{aligned}
\vec{e}_1^\top C\vec{e}_1 &= \begin{pmatrix} 1 & 0 & \cdots & 0 \end{pmatrix} \begin{pmatrix} c_{11} & \vec{v}^\top \\ \vec{v} & \tilde{C} \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 & \cdots & 0 \end{pmatrix} \begin{pmatrix} c_{11} \\ \vec{v} \end{pmatrix} \\
&= c_{11} \\
&> 0 \text{ since } C \text{ is positive definite and } \vec{e}_1 \neq \vec{0}.
\end{aligned}$$

This shows that–ignoring numerical issues– we do not have to use pivoting to ensure that $c_{11} \neq 0$ for the first step Gaussian elimination.

Continuing with Gaussian elimination, we can apply a forward substitution matrix E, which generically has the form

$$E = \begin{pmatrix} 1/\sqrt{c_{11}} & \vec{0}^\top \\ \vec{r} & I_{(n-1)\times(n-1)} \end{pmatrix}.$$

Here, the vector $\vec{r} \in \mathbb{R}^{n-1}$ contains the multiples of row 1 to cancel the rest of the first column of C. We also scale row 1 by $1/\sqrt{c_{11}}$ for reasons that will become apparent shortly!

By design, after forward substitution we know the product

$$EC = \begin{pmatrix} \sqrt{c_{11}} & \vec{v}^\top/\sqrt{c_{11}} \\ \vec{0} & D \end{pmatrix}$$

for some $D \in \mathbb{R}^{(n-1)\times(n-1)}$.

Here's where we diverge from Gaussian elimination: rather than moving on to the second row, we can post-multiply by $E^\top$ to obtain a product $ECE^\top$:

$$\begin{aligned}
ECE^\top &= (EC)E^\top \\
&= \begin{pmatrix} \sqrt{c_{11}} & \vec{v}^\top/\sqrt{c_{11}} \\ \vec{0} & D \end{pmatrix} \begin{pmatrix} 1/\sqrt{c_{11}} & \vec{r}^\top \\ \vec{0} & I_{(n-1)\times(n-1)} \end{pmatrix} \\
&= \begin{pmatrix} 1 & \vec{0}^\top \\ \vec{0} & \tilde{D} \end{pmatrix}
\end{aligned}$$

That is, we have eliminated the first row *and* the first column of C! Furthermore, it is easy to check that the matrix $\tilde{D}$ is also positive definite.

We can repeat this process to eliminate all the rows and columns of C symmetrically. Notice that we used both symmetry and positive definiteness to derive the factorization, since

- symmetry allowed us to apply the same E to both sides, and
- positive definiteness guaranteed that $c_{11} > 0$, thus implying that $\sqrt{c_{11}}$ exists.

In the end, similar to LU factorization we now obtain a factorization $C = LL^\top$ for a lower triangular matrix L. This is known as the *Cholesky factorization* of C. If taking the square roots along the diagonal causes numerical issues, a related $LDL^\top$ factorization, where D is a diagonal matrix, avoids this issue and is straightforward to derive from the discussion above.

The Cholesky factorization is important for a number of reasons. Most prominently, it takes half the memory to store L than the LU factorization of C or even C itself, since the elements above the diagonal are zero, and as in LU solving $C\vec{x} = \vec{d}$ is as easy as forward and back substitution. You will explore other properties of the factorization in the exercises.

In the end, code for Cholesky factorization can be very succinct. To derive a particularly compact form, suppose we choose an arbitrary row k and write L in block form isolating that row:

$$L = \begin{pmatrix} L_{11} & \vec{0} & 0 \\ \vec{\ell}_k^\top & \ell_{kk} & \vec{0}^\top \\ L_{31} & \vec{\ell}_k' & L_{33} \end{pmatrix}$$

Here, L_{11} and L_{33} are both lower triangular square matrices. Then, carrying out a product yields:

$$\begin{aligned}
LL^\top &= \begin{pmatrix} L_{11} & \vec{0} & 0 \\ \vec{\ell}_k^\top & \ell_{kk} & \vec{0}^\top \\ L_{31} & \vec{\ell}_k' & L_{33} \end{pmatrix} \begin{pmatrix} L_{11}^\top & \vec{\ell}_k & L_{31}^\top \\ \vec{0}^\top & \ell_{kk} & (\vec{\ell}_k')^\top \\ 0 & \vec{0} & L_{33}^\top \end{pmatrix} \\
&= \begin{pmatrix} \times & \times & \times \\ \vec{\ell}_k^\top L_{11}^\top & \vec{\ell}_k^\top \vec{\ell}_k + \ell_{kk}^2 & \times \\ \times & \times & \times \end{pmatrix}
\end{aligned}$$

We leave out values of the product that are not necessary for our derivation.

In the end, we know we can write $C = LL^\top$. The middle element of the product shows:

$$\ell_{kk} = \sqrt{c_{kk} - \|\ell_k\|_2^2}$$

where $\vec{\ell}_k \in \mathbb{R}^{k-1}$ contains the elements of the k-th row of L to the left of the diagonal. Furthermore, the middle left element of the product shows

$$L_{11}\vec{\ell}_k = \vec{c}_k$$

where $\vec{c}_k$ contains the elements of C in the same position as $\vec{\ell}_k$. Since L_{11} is lower triangular, this system can be solved by forward substitution!

Notice that our discussion above yields an algorithm for computing the Cholesky factorization from top to bottom, since L_{11} will already be computed by the time we reach row k. We provide pseudocode below, adapted from CITE:

```
// Takes as input an n-by-n matrix A[i,j]
// Edits A in place to obtain the Cholesky factorization in its lower triangle

for k from 1 to n {
    // Back-substitute to find l_k
    for i from 1 to k-1 { // element i of l_k
        sum = 0;
        for j from 1 to i-1
            sum += A[i,j]*A[k,j];
        A[k,i] = (A[k,i]-sum)/A[i,i];
    }

    // Apply the formula for l_kk
    normSquared = 0
    for j from 1 to i-1
        normSquared += A[k,j]^2;
    A[k,k] = sqrt(A[k,k] - normSquared);
}
```

As with LU factorization, this algorithm clearly runs in $O(n^3)$ time.

3.2.2 Sparsity

Many linear systems of equations naturally enjoy *sparsity* properties, meaning that most of the entries of A in the system $A\vec{x} = \vec{b}$ are exactly zero. Sparsity can reflect particular structure in a given problem, including the following use cases:

- In image processing, many systems for photo editing and understanding express relationships between the values of pixels and those of their neighbors on the image grid. An image may be a point in $\mathbb{R}^p$ for p pixels, but when solving $A\vec{x} = \vec{b}$ for a new size-p image, $A \in \mathbb{R}^{p\times p}$ may have only $O(p)$ rather than $O(p^2)$ nonzeros since each row only involves a single pixel and its up/down/left/right neighbors.

- In machine learning, a *graphical model* uses a graph structure $G \equiv (V, E)$ to express probability distributions over several variables. Each variable is represented using a node $v \in V$ of the graph, and edge $e \in E$ represents a probabilistic dependence. Linear systems arising in this context often have one row per vertex $v \in V$ with nonzeros only in columns involving v and its neighbors.

- In computational geometry, shapes are often expressed using collections of triangles linked together into a mesh. Equations for surface smoothing and other tasks once again link positions and other values at a given vertex with those at their neighbors in the mesh.

Example 3.5 (Harmonic parameterization)**.** *Suppose we wish to use an image to texture a triangle mesh. A mesh can be represented as a collection of vertices $V \subset \mathbb{R}^3$ linked together by edges $E \subseteq V \times V$ to form triangles. Since the vertices of the geometry are in $\mathbb{R}^3$, we must find a way to map them to the image plane to store the texture as an image. Thus, we must assign texture coordinates $t(v) \in \mathbb{R}^2$ on the image plane to each $v \in V$. See Figure NUMBER for an illustration.*

One strategy for making this map involves a single linear solve. Suppose the mesh has disk *topology, that is, it can be mapped to the interior of a circle in the plane. For each vertex v_b on the boundary of the mesh, we can specify the position of v_b by placing it on a circle. In the interior, we can ask that the texture map position be the average of its neighboring positions:*

$$t(v) = \frac{1}{|n(v)|} \sum_{w \in n(v)} t(w)$$

Here, $n(v) \subseteq V$ is the set of neighbors of $v \in V$ on the mesh. Thus, each $v \in V$ is associated with a linear equation, either fixing it on the boundary or asking that its position equal the average of its neighboring positions. This $|V| \times |V|$ system of equations leads to a stable parameterization strategy known as harmonic parameterization*; the matrix of the system only has $O(|V|)$ nonzeros in slots corresponding to vertices and their neighbors.*

Of course, if $A \in \mathbb{R}^{n \times n}$ is sparse to the point that it contains $O(n)$ rather than $O(n^2)$ nonzero values, there is no reason to store A as an $n \times n$ matrix. Instead, *sparse matrix* storage techniques only store the $O(n)$ nonzeros in a more reasonable data structure, e.g. a list of row/column/value triplets. The choice of a matrix data structure involves considering the likely operations that will occur on the matrix, possibly including multiplication, iteration over nonzeros, or iterating over individual rows or columns.

Unfortunately, it is easy to see that the LU factorization of a sparse A may not result in sparse L and U matrices; this loss of structure severely limits the applicability of using these methods to solve $A\vec{x} = \vec{b}$ when A is large but sparse. Thankfully, there are many *direct sparse solvers* adapting LU to sparse matrices that can produce an LU-like factorization without inducing much *fill*, or additional nonzeros; discussion of these techniques is outside the scope of this text. Alternatively, *iterative* techniques have been used to obtain approximate solutions to linear systems; we will defer discussion of these methods to future chapters.

Certain matrices are not only sparse but also *structured*. For instance, a *tridiagonal* system of linear equations has the following pattern of nonzero values:

$$\begin{pmatrix} \times & \times & & & \\ \times & \times & \times & & \\ & \times & \times & \times & \\ & & \times & \times & \times \\ & & & \times & \times \end{pmatrix}$$

In the exercises following this chapter, you will derive a special version of Gaussian elimination for dealing with this this simple *banded* structure.

In other cases, matrices may not be sparse but might admit a sparse representation. For example, consider the *cyclic* matrix:

$$\begin{pmatrix} a & b & c & d \\ d & a & b & c \\ c & d & a & b \\ b & c & d & a \end{pmatrix}$$

Obviously, this matrix can be stored using only the values a, b, c, d. Specialized techniques for this and other classes of matrices are well-studied and often more efficient than generic Gaussian elimination.

3.3 Sensitivity Analysis

As we have seen, it is important to examine the matrix of a linear system to find out if it has special properties that can simplify the solution process. Sparsity, positive definiteness, symmetry, and so on all call can provide clues to the proper solver to use for a particular problem.

Even if a given solution strategy might work in theory, however, it is equally important to understand how well we can trust the answer to a linear system given by a particular solver. For instance, due to rounding and other discrete effects, it might be the case that an implementation of Gaussian elimination for solving $A\vec{x} = \vec{b}$ yields a solution $\vec{x}_0$ such that $0 < \|A\vec{x}_0 - \vec{b}\| \ll 1$; in other words, $\vec{x}_0$ only solves the system approximately.

One way to understand the likelihood of these approximation effects is through *sensitivity analysis*. In this approach, we ask what might happen to $\vec{x}$ if instead of solving $A\vec{x} = \vec{b}$ in reality we solve a *perturbed* system of equations $(A + \delta A)\vec{x} = \vec{b} + \delta\vec{b}$. There are two ways of viewing conclusions made by this type of analysis:

1. Likely we make mistakes representing A and $\vec{b}$ thanks to rounding and other effects. This analysis then shows the best possible accuracy we can expect for $\vec{x}$ given the mistakes made representing the problem.

2. If our solver generates an approximation $\vec{x}_0$ to the solution of $A\vec{x} = \vec{b}$, it is an exact solution to the system $A\vec{x}_0 = \vec{b}_0$ if we *define* $\vec{b}_0 \equiv A\vec{x}_0$ (be sure you understand why this sentence is not a tautology!). Understanding how changes in $\vec{x}_0$ affect changes in $\vec{b}_0$ show how sensitive the system is to slightly incorrect answers.

Notice that our discussion here is similar to and indeed motivated by our definitions of forward and backward error in previous chapters.

3.3.1 Matrix and Vector Norms

Before we can discuss the sensitivity of a linear system, we have to be somewhat careful to define what it means for a change $\delta\vec{x}$ to be "small." Generally, we wish to measure the length, or *norm*, of a vector $\vec{x}$. We have already encountered the two-norm of a vector:

$$\|\vec{x}\|_2 \equiv \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}$$

for $\vec{x} \in \mathbb{R}^n$. This norm is popular thanks to its connection to Euclidean geometry, but it is by no means the only norm on $\mathbb{R}^n$. Most generally, we define a *norm* as follows:

Definition 3.2 (Vector norm). *A vector norm is a function $\|\cdot\| : \mathbb{R}^n \to [0, \infty)$ satisfying the following conditions:*

- $\|\vec{x}\| = 0$ *if and only if* $\vec{x} = \vec{0}$.

- $\|c\vec{x}\| = |c|\|\vec{x}\|$ *for all scalars* $c \in \mathbb{R}$ *and vectors* $\vec{x} \in \mathbb{R}^n$.
- $\|\vec{x} + \vec{y}\| \leq \|\vec{x}\| + \|\vec{y}\|$ *for all* $\vec{x}, \vec{y} \in \mathbb{R}^n$.

While we use the two subscript $\|\cdot\|_2$ to denote the two-norm of a vector, unless we note otherwise we will use the notation $\|\vec{x}\|$ to denote the two-norm of $\vec{x}$. Other than this norm, there are many other examples:

- The p-norm $\|\vec{x}\|_p$, for $p \geq 1$, given by:

$$\|\vec{x}\|_p \equiv (|x_1|^p + |x_2|^p + \cdots + |x_n|^p)^{1/p}$$

  Of particular importance is the 1-norm or "taxicab" norm, given by

$$\|\vec{x}\|_1 \equiv \sum_{k=1}^{n} |x_k|.$$

  This norm receives its nickname because it represents the distance a taxicab drives between two points in a city where the roads only run north/south and east/west.

- The ∞-norm $\|\vec{x}\|_\infty$ given by:

$$\|\vec{x}\|_\infty \equiv \max(|x_1|, |x_2|, \cdots, |x_n|).$$

In some sense, many norms on $\mathbb{R}^n$ are the same. In particular, suppose we say two norms are *equivalent* when they satisfy the following property:

Definition 3.3 (Equivalent norms)**.** *Two norms* $\|\cdot\|$ *and* $\|\cdot\|'$ *are* equivalent *if there exist constants* c_{low} *and* c_{high} *such that*

$$c_{low}\|\vec{x}\| \leq \|\vec{x}\|' \leq c_{high}\|\vec{x}\|$$

for all $\vec{x} \in \mathbb{R}^n$.

This condition guarantees that up to some constant factors, all norms agree on which vectors are "small" and "large." In fact, we will state without proof a famous theorem from analysis:

Theorem 3.2 (Equivalence of norms on $\mathbb{R}^n$)**.** *All norms on* $\mathbb{R}^n$ *are equivalent.*

This somewhat surprising result implies that all vector norms have the same *rough* behavior, but the choice of a norm for analyzing or stating a particular problem can make a huge difference. For instance, on $\mathbb{R}^3$ the ∞-norm considers the vector $(1000, 1000, 1000)$ to have the same norm as $(1000, 0, 0)$ whereas the 2-norm certainly is affected by the additional nonzero values.

Since we perturb not only vectors but also matrices, we must also be able to take the norm of a matrix. Of course, the basic definition of a norm does not change on $\mathbb{R}^{n\times m}$. For this reason, we can "unroll" any matrix in $\mathbb{R}^{m\times n}$ to a vector in $\mathbb{R}^{nm}$ to adopt any vector norm to matrices. One such norm is the *Frobenius norm*, given by

$$\|A\|_{\text{Fro}} \equiv \sqrt{\sum_{i,j} a_{ij}^2}.$$

Such adaptations of vector norms, however, are not always very meaningful. In particular, the priority for understanding the structure of a matrix A often is its *action* on vectors, that is, the likely results when A is multiplied by an arbitrary $\vec{x}$. With this motivation, we can define the norm *induced* by a vector norm as follows:

Definition 3.4 (Induced norm). *The norm on* $\mathbb{R}^{m\times n}$ induced *by a norm* $\|\cdot\|$ *on* $\mathbb{R}^n$ *is given by*

$$\|A\| \equiv \max\{\|A\vec{x}\| : \|\vec{x}\| = 1\}.$$

That is, the induced norm is the maximum length of the image of a unit vector multiplied by A.

Since vector norms satisfy $\|c\vec{x}\| = |c|\|\vec{x}\|$, it is easy to see that this definition is equivalent to requiring

$$\|A\| \equiv \max_{\vec{x}\in\mathbb{R}^n\backslash\{0\}} \frac{\|A\vec{x}\|}{\|\vec{x}\|}.$$

From this standpoint, the norm of A induced by $\|\cdot\|$ is the largest achievable ratio of the norm of $A\vec{x}$ relative to that of the input $\vec{x}$.

This general definition makes it somewhat hard to compute the norm $\|A\|$ given a matrix A and a choice of $\|\cdot\|$. Fortunately, the matrix norms induced by many popular vector norms can be simplified. We state some such expressions without proof:

- The induced one-norm of A is the maximum sum of any one column of A:

$$\|A\|_1 = \max_{1\leq j\leq n} \sum_{i=1}^{m} |a_{ij}|$$

- The induced ∞-norm of A is the maximum sum of any one row of A:

$$\|A\|_\infty = \max_{1\leq i\leq m} \sum_{j=1}^{n} |a_{ij}|$$

- The induced two-norm, or *spectral norm*, of $A \in \mathbb{R}^{n\times n}$ is the square root of the largest eigenvalue of $A^\top A$. That is,

$$\|A\|_2^2 = \max\{\lambda : \text{there exists } \vec{x} \in \mathbb{R}^n \text{ with } A^\top A\vec{x} = \lambda\vec{x}\}$$

At least the first two norms are relatively easy to compute; the third we will return to while discussing eigenvalue problems.

3.3.2 Condition Numbers

Now that we have tools for measuring the action of a matrix, we can define the *condition number* of a linear system by adapting our generic definition of condition numbers from Chapter 1. We follow the development presented in CITE.

Suppose we perturbation δA of a matrix A and a corresponding perturbation $\delta\vec{b}$. For each $\varepsilon \geq 0$, ignoring invertibility technicalities we can write a vector $\vec{x}(\varepsilon)$ as the solution to

$$(A + \varepsilon \cdot \delta A)\vec{x}(\varepsilon) = \vec{b} + \varepsilon \cdot \delta\vec{b}.$$

If we differentiate both sides with respect to ε and apply the product rule, we obtain the following result:

$$\delta A \cdot \vec{x} + (A + \varepsilon \cdot \delta A)\frac{d\vec{x}}{d\varepsilon} = \delta\vec{b}$$

In particular, when $\varepsilon = 0$ we find

$$\delta A \cdot \vec{x}(0) + A \frac{d\vec{x}}{d\varepsilon}\Big|_{\varepsilon=0} = \delta \vec{b}$$

or, equivalently,

$$\frac{d\vec{x}}{d\varepsilon}\Big|_{\varepsilon=0} = A^{-1}(\delta\vec{b} - \delta A \cdot \vec{x}(0)).$$

Using the Taylor expansion, we can write

$$\vec{x}(\varepsilon) = \vec{x} + \varepsilon \vec{x}'(0) + O(\varepsilon^2),$$

where we define $\vec{x}'(0) = \frac{d\vec{x}}{d\varepsilon}\big|_{\varepsilon=0}$. Thus, we can expand the relative error made by solving the perturbed system:

$$\begin{aligned}
\frac{\|\vec{x}(\varepsilon) - \vec{x}(0)\|}{\|\vec{x}(0)\|} &= \frac{\|\varepsilon\vec{x}'(0) + O(\varepsilon^2)\|}{\|\vec{x}(0)\|} \text{ by the Taylor expansion} \\
&= \frac{\|\varepsilon A^{-1}(\delta\vec{b} - \delta A \cdot \vec{x}(0)) + O(\varepsilon^2)\|}{\|\vec{x}(0)\|} \text{ by the derivative we computed} \\
&\leq \frac{|\varepsilon|}{\|\vec{x}(0)\|}(\|A^{-1}\delta\vec{b}\| + \|A^{-1}\delta A \cdot \vec{x}(0))\|) + O(\varepsilon^2) \\
&\qquad \text{by the triangle inequality } \|A+B\| \leq \|A\| + \|B\| \\
&\leq |\varepsilon|\|A^{-1}\|\left(\frac{\|\delta\vec{b}\|}{\|\vec{x}(0)\|} + \|\delta A\|\right) + O(\varepsilon^2) \text{ by the identity } \|AB\| \leq \|A\|\|B\| \\
&= |\varepsilon|\|A^{-1}\|\|A\|\left(\frac{\|\delta\vec{b}\|}{\|A\|\|\vec{x}(0)\|} + \frac{\|\delta A\|}{\|A\|}\right) + O(\varepsilon^2) \\
&\leq |\varepsilon|\|A^{-1}\|\|A\|\left(\frac{\|\delta\vec{b}\|}{\|A\vec{x}(0)\|} + \frac{\|\delta A\|}{\|A\|}\right) + O(\varepsilon^2) \text{ since } \|A\vec{x}(0)\| \leq \|A\|\|\vec{x}(0)\| \\
&= |\varepsilon|\|A^{-1}\|\|A\|\left(\frac{\|\delta\vec{b}\|}{\|\vec{b}\|} + \frac{\|\delta A\|}{\|A\|}\right) + O(\varepsilon^2) \text{ since } \|A\vec{x}(0)\| \leq \|A\|\|\vec{x}(0)\| \\
&\qquad \text{since by definition, } A\vec{x}(0) = \vec{b}
\end{aligned}$$

Here we have applied some properties of the matrix norm which follow from corresponding properties for vectors. Notice that the sum $D \equiv \|\delta\vec{b}\|/\|\vec{b}\| + \|\delta A\|/\|A\|$ encodes the relative perturbations of A and $\vec{b}$. From this standpoint, to first order we have bounded the relative error of perturbing the system by ε using a factor $\kappa \equiv \|A\|\|A^{-1}\|$:

$$\frac{\|\vec{x}(\varepsilon) - \vec{x}(0)\|}{\|\vec{x}(0)\|} \leq \varepsilon \cdot D \cdot \kappa + O(\varepsilon^2)$$

In this way, the quantity κ bounds the conditioning of linear systems involving A. For this reason, we make the following definition:

Definition 3.5 (Matrix condition number). *The condition number of $A \in \mathbb{R}^{n\times n}$ for a given matrix norm $\|\cdot\|$ is*

$$\textit{cond } A \equiv \|A\|\|A^{-1}\|.$$

If A is not invertible, we take cond $A \equiv \infty$.

It is easy to see that cond $A \geq 1$ for all A, that scaling A has no effect on its condition number, and that the condition number of the identity matrix is 1. These properties contrast with the determinant, which can scale up and down as you scale A.

If $\|\cdot\|$ is induced by a vector norm and A is invertible, then we have

$$\begin{aligned}\|A^{-1}\| &= \max_{\vec{x}\neq\vec{0}} \frac{\|A^{-1}\vec{x}\|}{\|\vec{x}\|} \text{ by definition}\\ &= \max_{\vec{y}\neq\vec{0}} \frac{\|\vec{y}\|}{\|A\vec{y}\|} \text{ by substituting } \vec{y} = A^{-1}\vec{x}\\ &= \left(\min_{\vec{y}\neq\vec{0}} \frac{\|A\vec{y}\|}{\|\vec{y}\|}\right)^{-1} \text{ by taking the reciprocal}\end{aligned}$$

In this case, the condition number of A is given by:

$$\text{cond } A = \left(\max_{\vec{x}\neq\vec{0}} \frac{\|A\vec{x}\|}{\|\vec{x}\|}\right)\left(\min_{\vec{y}\neq\vec{0}} \frac{\|A\vec{y}\|}{\|\vec{y}\|}\right)^{-1}$$

In other words, cond A measures the maximum to minimum possible stretch of a vector $\vec{x}$ under A.

More generally, a desirable stability property of a system $A\vec{x} = \vec{b}$ is that if A or $\vec{b}$ is perturbed, the solution $\vec{x}$ does not change considerably. Our motivation for cond A shows that when the condition number is small, the change in $\vec{x}$ is small relative to the change in A or $\vec{b}$, as illustrated in Figure NUMBER. Otherwise, a small change in the parameters of the linear system can cause large deviations in $\vec{x}$; this instability can cause linear solvers to make large mistakes in $\vec{x}$ due to rounding and other approximations during the solution process.

The norm $\|A^{-1}\|$ can be as difficult as computing the full inverse A^{-1}. One way to lower-bound the condition number is to apply the identity $\|A^{-1}\vec{x}\| \leq \|A^{-1}\|\|\vec{x}\|$. Thus, for any $\vec{x} \neq \vec{0}$ we can write $\|A^{-1}\| \geq \|A^{-1}\vec{x}\|/\|\vec{x}\|$. Thus,

$$\text{cond } A = \|A\|\|A^{-1}\| \geq \frac{\|A\|\|A^{-1}\vec{x}\|}{\|\vec{x}\|}.$$

So, we can bound the condition number by solving $A^{-1}\vec{x}$ for some vectors $\vec{x}$; of course, the necessity of a linear solver to find $A^{-1}\vec{x}$ creates a circular dependence on the condition number to evaluate the quality of the estimate! When $\|\cdot\|$ is induced by the two-norm, in future chapters we will provide more reliable estimates.

3.4 Problems

Something like:

- Kernel regression as an example of §3.1.1
- Least-norm solution to $A\vec{x} = \vec{b}$, least squares matrix is invertible otherwise
- Variational versions of Tikhonov regularization/"ridge" regression (not the usual approach to this, but whatever); completing the underdetermined story this way
- L^1 approaches to regularization for contrast – draw a picture of why to expect sparsity, draw unit circles, show that p norm isn't a norm for $p < 1$, take limit as $p \to 0$
- Mini-Riesz – derive matrix for inner product, use to show how to rotate space
- tridiagonal solve
- properties of condition number

Chapter 4

Column Spaces and QR

One way to interpret the linear problem $A\vec{x} = \vec{b}$ for $\vec{x}$ is that we wish to write $\vec{b}$ as a linear combination of the columns of A with weights given in $\vec{x}$. This perspective does not change when we allow $A \in \mathbb{R}^{m\times n}$ to be non-square, but the solution may not exist or be unique depending on the structure of the column space. For these reasons, some techniques for factoring matrices and analyzing linear systems seek simpler *representations* of the column space to disambiguate solvability and span more explicitly than row-based factorizations like LU.

4.1 The Structure of the Normal Equations

As we have shown, a necessary and sufficient condition for $\vec{x}$ to be a solution of the least-squares problem $A\vec{x} \approx \vec{b}$ is that $\vec{x}$ satisfy the normal equations $(A^\top A)\vec{x} = A^\top\vec{b}$. This theorem suggests that solving least-squares is a simple enough extension of linear techniques. Methods such as Cholesky factorization also show that the special structure of least-squares problems can be used to the solver's advantage.

There is one large problem limiting the use of this approach, however. For now, suppose A is square; then we can write:

$$\begin{aligned}
\text{cond } A^\top A &= \|A^\top A\| \|(A^\top A)^{-1}\| \\
&\approx \|A^\top\| \|A\| \|A^{-1}\| \|(A^\top)^{-1}\| \text{ depending on the choice of } \|\cdot\| \\
&= \|A\|^2 \|A^{-1}\|^2 \\
&= (\text{cond } A)^2
\end{aligned}$$

That is, the condition number of $A^\top A$ is approximately the **square** of the condition number of A! Thus, while generic linear strategies might work on $A^\top A$ when the least-squares problem is "easy," when the columns of A are nearly linearly dependent these strategies are likely to generate considerable error since they do not deal with A directly.

Intuitively, a primary reason that cond A can be large is that columns of A might look "similar." Think of each column of A as a vector in $\mathbb{R}^m$. If two columns $\vec{a}_i$ and $\vec{a}_j$ satisfy $\vec{a}_i \approx \vec{a}_j$, then the least-squares residual length $\|\vec{b} - A\vec{x}\|$ probably would not suffer much if we replace multiples of $\vec{a}_i$ with multiples of $\vec{a}_j$ or vice versa. This wide range of nearly–but not completely–equivalent solutions yields poor conditioning. While the resulting vector $\vec{x}$ is unstable, however, the product

$A\vec{x}$ remains nearly unchanged, by design of our substitution. Therefore, if we wish to solve $A\vec{x} \approx \vec{b}$ simply to write $\vec{b}$ in the column space of A, either solution would suffice.

To solve such poorly-conditioned problems, we will employ an alternative strategy with closer attention to the column space of A rather than employing row operations as in Gaussian elimination. This way, we can identify such near-dependencies *explicitly* and deal with them in a numerically stable way.

4.2 Orthogonality

We have determined when the least-squares problem is difficult, but we might also ask when it is most straightforward. If we can reduce a system to the straightforward case without inducing conditioning problems along the way, we will have found a more stable way around the issues explained in §4.1.

Obviously the easiest linear system to solve is $I_{n\times n}\vec{x} = \vec{b}$: The solution simply is $\vec{x} \equiv \vec{b}$! We are unlikely to enter this particular linear system into our solver explicitly, but we may do so accidentally while solving least-squares. In particular, even when $A \neq I_{n\times n}$—in fact, A need not be a square matrix—we may in particularly lucky circumstances find that the normal matrix $A^\top A$ satisfies $A^\top A = I_{n\times n}$. To avoid confusion with the general case, we will use the letter Q to represent such a matrix.

Simply praying that $Q^\top Q = I_{n\times n}$ unlikely will yield a desirable solution strategy, but we can examine this case to see how it becomes so favorable. Write the columns of Q as vectors $\vec{q}_1, \cdots, \vec{q}_n \in \mathbb{R}^m$. Then, it is easy to verify that the product $Q^\top Q$ has the following structure:

$$Q^\top Q = \begin{pmatrix} - & \vec{q}_1^\top & - \\ - & \vec{q}_2^\top & - \\ & \vdots & \\ - & \vec{q}_n^\top & \end{pmatrix} \begin{pmatrix} | & | & & | \\ \vec{q}_1 & \vec{q}_2 & \cdots & \vec{q}_n \\ | & | & & | \end{pmatrix} = \begin{pmatrix} \vec{q}_1\cdot\vec{q}_1 & \vec{q}_1\cdot\vec{q}_2 & \cdots & \vec{q}_1\cdot\vec{q}_n \\ \vec{q}_2\cdot\vec{q}_1 & \vec{q}_2\cdot\vec{q}_2 & \cdots & \vec{q}_2\cdot\vec{q}_n \\ \vdots & \vdots & \cdots & \vdots \\ \vec{q}_n\cdot\vec{q}_1 & \vec{q}_n\cdot\vec{q}_2 & \cdots & \vec{q}_n\cdot\vec{q}_n \end{pmatrix}$$

Setting the expression on the right equal to $I_{n\times n}$ yields the following relationship:

$$\vec{q}_i \cdot \vec{q}_j = \begin{cases} 1 & \text{when } i = j \\ 0 & \text{when } i \neq j \end{cases}$$

In other words, the columns of Q are unit-length and orthogonal to one another. We say that they form an *orthonormal basis* for the column space of Q:

Definition 4.1 (Orthonormal; orthogonal matrix). *A set of vectors $\{\vec{v}_1, \cdots, \vec{v}_k\}$ is* orthonormal *if $\|\vec{v}_i\| = 1$ for all i and $\vec{v}_i \cdot \vec{v}_j = 0$ for all $i \neq j$. A square matrix whose columns are orthonormal is called an* orthogonal *matrix.*

We motivated our discussion by asking when we can expect $Q^\top Q = I_{n\times n}$. Now it is easy to see that this occurs when the columns of Q are orthonormal. Furthermore, if Q is square and invertible with $Q^\top Q = I_{n\times n}$, then simply by multiplying both sides of this expression by Q^{-1} we find $Q^{-1} = Q^\top$. Thus, solving $Q\vec{x} = \vec{b}$ in this case is as easy as multiplying both sides by the transpose $Q^\top$.

Orthonormality also has a strong geometric interpretation. Recall from Chapter 0 that we can regard two orthogonal vectors $\vec{a}$ and $\vec{b}$ as being *perpendicular*. So, an orthonormal set of vectors simply is a set of unit-length perpendicular vectors in $\mathbb{R}^n$. If Q is orthogonal, then its action does not affect the length of vectors:

$$\|Q\vec{x}\|^2 = \vec{x}^\top Q^\top Q\vec{x} = \vec{x}^\top I_{n\times n}\vec{x} = \vec{x}\cdot\vec{x} = \|\vec{x}\|^2$$

Similarly, Q cannot affect the angle between two vectors, since:

$$(Q\vec{x})\cdot(Q\vec{y}) = \vec{x}^\top Q^\top Q\vec{y} = \vec{x}^\top I_{n\times n}\vec{y} = \vec{x}\cdot\vec{y}$$

From this standpoint, if Q is orthogonal, then Q represents an *isometry* of $\mathbb{R}^n$, that is, it preserves lengths and angles. In other words, it can rotate or reflect vectors but cannot scale or shear them. From a high level, the linear algebra of orthogonal matrices is easier because their action does not affect the geometry of the underlying space in any nontrivial way.

4.2.1 Strategy for Non-Orthogonal Matrices

Except in special circumstances, most of our matrices A when solving $A\vec{x} = \vec{b}$ or the corresponding least-squares problem will not be orthogonal, so the machinery of §4.2 does not apply directly. For this reason, we must do some additional computations to connect the general case to the orthogonal one.

Take a matrix $A \in \mathbb{R}^{m\times n}$, and denote its column space as col A; recall that col A represents the span of the columns of A. Now, suppose a matrix $B \in \mathbb{R}^{n\times n}$ is invertible. We can make a simple observation about the column space of AB relative to that of A:

Lemma 4.1 (Column space invariance)**.** *For any $A \in \mathbb{R}^{m\times n}$ and invertible $B \in \mathbb{R}^{n\times n}$,*

$$\text{col } A = \text{col } AB.$$

Proof. Suppose $\vec{b} \in$ col A. Then, by definition of multiplication by A there exists $\vec{x}$ with $A\vec{x} = \vec{b}$. Then, $(AB)\cdot(B^{-1}\vec{x}) = A\vec{x} = \vec{b}$, so $\vec{b} \in$ col AB. Conversely, take $\vec{c} \in$ col AB, so there exists $\vec{y}$ with $(AB)\vec{y} = \vec{c}$. Then, $A\cdot(B\vec{y}) = \vec{c}$, showing that $\vec{c}$ is in col A. □

Recall the "elimination matrix" description of Gaussian elimination: We started with a matrix A and applied row operation matrices E_i such that the sequence $A, E_1A, E_2E_1A, \ldots$ represented sequentially easier linear systems. The lemma above suggests an alternative strategy for situations in which we care about the column space: Apply *column* operations to A by *post*-multiplication until the columns are orthonormal. That is, we obtain a product $Q = AE_1E_2\cdots E_k$ such that Q is orthonormal. As long as the E_i's are invertible, the lemma shows that col Q = col A. Inverting these operations yields a factorization $A = QR$ for $R = E_k^{-1}E_{k-1}^{-1}\cdots E_1^{-1}$.

As in the LU factorization, if we design R carefully, the solution of least-squares problems $A\vec{x} \approx \vec{b}$ may simplify. In particular, when $A = QR$, we can write the solution to $A^\top A\vec{x} = A^\top\vec{b}$ as follows:

$$\begin{aligned}\vec{x} &= (A^\top A)^{-1}A^\top\vec{b}\\ &= (R^\top Q^\top QR)^{-1}R^\top Q^\top\vec{b} \text{ since } A = QR\\ &= (R^\top R)^{-1}R^\top Q^\top\vec{b} \text{ since } Q \text{ is orthogonal}\\ &= R^{-1}(R^\top)^{-1}R^\top Q^\top\vec{b} \text{ since } (AB)^{-1} = B^{-1}A^{-1}\\ &= R^{-1}Q^\top\vec{b}\end{aligned}$$

Or equivalently, $R\vec{x} = Q^\top\vec{b}$

Thus, if we design R to be a triangular matrix, then solving the linear system $R\vec{x} = Q^\top \vec{b}$ is as simple as back-substitution.

Our task for the remainder of the chapter is to design strategies for such a factorization.

4.3 Gram-Schmidt Orthogonalization

Our first approach for finding QR factorizations is the simplest to describe and implement but may suffer from numerical issues. We use it here as an initial strategy and then will improve upon it with better operations.

4.3.1 Projections

Suppose we have two vectors $\vec{a}$ and $\vec{b}$. Then, we could easily ask "Which multiple of $\vec{a}$ is closest to $\vec{b}$?" Mathematically, this task is equivalent to minimizing $\|c\vec{a} - \vec{b}\|^2$ over all possible $c \in \mathbb{R}$. If we think of $\vec{a}$ and $\vec{b}$ as $n \times 1$ matrices and c as a 1×1 matrix, then this is nothing more than an unconventional least-squares problem $\vec{a} \cdot c \approx \vec{b}$. In this case, the normal equations show $\vec{a}^\top \vec{a} \cdot c = \vec{a}^\top \vec{b}$, or

$$c = \frac{\vec{a} \cdot \vec{b}}{\vec{a} \cdot \vec{a}} = \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\|^2}.$$

We denote this *projection* of $\vec{b}$ onto $\vec{a}$ as:

$$\text{proj}_{\vec{a}}\, \vec{b} = c\vec{a} = \frac{\vec{a} \cdot \vec{b}}{\vec{a} \cdot \vec{a}}\vec{a} = \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\|^2}\vec{a}$$

Obviously $\text{proj}_{\vec{a}}\, \vec{b}$ is parallel to $\vec{a}$. What about the remainder $\vec{b} - \text{proj}_{\vec{a}}\vec{b}$? We can do a simple computation to find out:

$$\begin{aligned}
\vec{a} \cdot (\vec{b} - \text{proj}_{\vec{a}}\, \vec{b}) &= \vec{a} \cdot \vec{b} - \vec{a} \cdot \left(\frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\|^2}\vec{a}\right) \\
&= \vec{a} \cdot \vec{b} - \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\|^2}(\vec{a} \cdot \vec{a}) \\
&= \vec{a} \cdot \vec{b} - \vec{a} \cdot \vec{b} \\
&= 0
\end{aligned}$$

Thus, we have decomposed $\vec{b}$ into a component parallel to $\vec{a}$ and another component orthogonal to $\vec{a}$.

Now, suppose that $\hat{a}_1, \hat{a}_2, \cdots, \hat{a}_k$ are orthonormal; we will put hats over vectors with unit length. Then, for any single i we can see:

$$\text{proj}_{\hat{a}_i}\, \vec{b} = (\hat{a}_i \cdot \vec{b})\hat{a}_i$$

The norm term does not appear because $\|\hat{a}_i\| = 1$ by definition. We could project $\vec{b}$ onto span $\{\hat{a}_1, \cdots, \hat{a}_k\}$ by minimizing the following energy over $c_1, \ldots, c_k \in \mathbb{R}$:

$$\begin{aligned}
\|c_1\hat{a}_1 + c_2\hat{a}_2 + \cdots + c_k\hat{a}_k - \vec{b}\|^2 &= \left(\sum_{i=1}^{k}\sum_{j=1}^{k} c_i c_j (\hat{a}_i \cdot \hat{a}_j)\right) - 2\vec{b} \cdot \left(\sum_{i=1}^{k} c_i \hat{a}_i\right) + \vec{b} \cdot \vec{b} \\
&\quad \text{by applying } \|\vec{v}\|^2 = \vec{v} \cdot \vec{v} \\
&= \sum_{i=1}^{k} \left(c_i^2 - 2c_i \vec{b} \cdot \hat{a}_i\right) + \|\vec{b}\|^2 \text{ since the } \hat{a}_i\text{'s are orthonormal}
\end{aligned}$$

Notice that the second step here is only valid because of orthonormality. At a minimum, the derivative with respect to c_i is zero for every c_i, yielding:

$$c_i = \hat{a}_i \cdot \vec{b}$$

Thus, we have shown that when $\hat{a}_1, \cdots, \hat{a}_k$ are orthonormal, the following relationship holds:

$$\mathrm{proj}_{\mathrm{span}\,\{\hat{a}_1, \cdots, \hat{a}_k\}} \vec{b} = (\hat{a}_1 \cdot \vec{b})\hat{a}_1 + \cdots + (\hat{a}_k \cdot \vec{b})\hat{a}_k$$

This is simply an extension of our projection formula, and by a similar proof it is easy to see that

$$\hat{a}_i \cdot (\vec{b} - \mathrm{proj}_{\mathrm{span}\,\{\hat{a}_1, \cdots, \hat{a}_k\}} \vec{b}) = 0.$$

That is, we have separated $\vec{b}$ into a component parallel to the span of the $\hat{a}_i$'s and a perpendicular residual.

4.3.2 Gram-Schmidt Orthogonalization

Our observations above lead to a simple algorithm for *orthogonalization*, or finding an orthogonal basis $\{\hat{a}_1, \cdots, \hat{a}_k\}$ whose span is the same as that of a set of linearly independent input vectors $\{\vec{v}_1, \cdots, \vec{v}_k\}$:

1. Set
$$\hat{a}_1 \equiv \frac{\vec{v}_1}{\|\vec{v}_1\|}.$$
That is, we take $\hat{a}_1$ to be a unit vector parallel to $\vec{v}_1$.

2. For i from 2 to k,

   (a) Compute the projection
   $$\vec{p}_i \equiv \mathrm{proj}_{\mathrm{span}\,\{\hat{a}_1, \cdots, \hat{a}_{i-1}\}} \vec{v}_i.$$
   By definition $\hat{a}_1, \cdots, \hat{a}_{i-1}$ are orthonormal, so our formula above applies.

   (b) Define
   $$\hat{a}_i \equiv \frac{\vec{v}_i - \vec{p}_i}{\|\vec{v}_i - \vec{p}_i\|}.$$

This technique, known as "Gram-Schmidt Orthogonalization," is a straightforward application of our discussion above. The key to the proof of this technique is to notice that $\operatorname{span}\{\vec{v}_1, \cdots, \vec{v}_i\} = \operatorname{span}\{\hat{a}_1, \cdots, \hat{a}_i\}$ for each $i \in \{1, \cdots, k\}$. Step 1 clearly makes this the case for $i = 1$, and for $i > 1$ the definition of $\hat{a}_i$ in step 2b simply removes the projection onto the vectors we already have seen.

If we start with a matrix A whose columns are $\vec{v}_1, \cdots, \vec{v}_k$, then we can implement Gram-Schmidt as a series of column operations on A. Dividing column i of A by its norm is equivalent to post-multiplying A by a $k \times k$ diagonal matrix. Similarly, subtracting off the projection of a column onto the orthonormal columns to its left as in step 2 is equivalent to post-multiplying by an upper-triangular matrix: Be sure to understand why this is the case! Thus, our discussion in §4.2.1 applies, and we can use Gram-Schmidt to obtain a factorization $A = QR$.

Unfortunately, the Gram-Schmidt algorithm can introduce serious numerical instabilities due to the subtraction step. For instance, suppose we provide the vectors $\vec{v}_1 = (1, 1)$ and $\vec{v}_2 = (1 + \varepsilon, 1)$ as input to Gram-Schmidt for some $0 < \varepsilon \ll 1$. Notice that an obvious basis for $\operatorname{span}\{\vec{v}_1, \vec{v}_2\}$ is $\{(1, 0), (0, 1)\}$. But, if we apply Gram-Schmidt, we obtain:

$$\begin{aligned}
\hat{a}_1 &= \frac{\vec{v}_1}{\|\vec{v}_1\|} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \\
\vec{p}_2 &= \frac{2+\varepsilon}{2} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \\
\vec{v}_2 - \vec{p}_2 &= \begin{pmatrix} 1+\varepsilon \\ 1 \end{pmatrix} - \frac{2+\varepsilon}{2} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \\
&= \frac{1}{2} \begin{pmatrix} \varepsilon \\ -\varepsilon \end{pmatrix}
\end{aligned}$$

Notice that $\|\vec{v}_2 - \vec{p}_2\| = (\sqrt{2}/2) \cdot \varepsilon$, so computing $\hat{a}_2$ will require division by a scalar on the order of ε. Division by small numbers is an unstable numerical operation that we should avoid.

4.4 Householder Transformations

In §4.2.1, we motivated the construction of QR factorization by post-multiplication and column operations. This construction is reasonable in the context of analyzing column spaces, but as we saw in our derivation of the Gram-Schmidt algorithm, the resulting numerical techniques can be unstable.

Rather than starting with A and post-multiplying by column operations to obtain $Q = AE_1 \cdots E_k$, however, we can preserve our high-level strategy from Gaussian elimination. That is, we can start with A and *pre*-multiply by orthogonal matrices Q_i to obtain $Q_k \cdots Q_1 A = R$; these Q's will act like row operations, eliminating elements of A until the resulting product R is upper-triangular. Then, thanks to orthogonality of the Q's we can write $A = Q_1^\top \cdots Q_k^\top R$, obtaining the QR factorization since the product of orthogonal matrices is orthogonal.

The row operation matrices we used in Gaussian elimination and LU will not suffice for QR factorization since they are not orthogonal. A number of alternatives have been suggested; we will introduce one common strategy introduced in 1958 by Alston Scott Householder.

The space of orthogonal $n \times n$ matrices is very large, so we must find a smaller space of Q_i's that is easier to work with. From our geometric discussions in §4.2, we know that orthogonal matrices must preserve angles and lengths, so intuitively they only can rotate and reflect vectors.

Thankfully, the reflections can be easy to write in terms of projections, as illustrated in Figure NUMBER. Suppose we have a vector $\vec{b}$ that we wish to reflect over a vector $\vec{v}$. We have shown that the residual $\vec{r} \equiv \vec{b} - \text{proj}_{\vec{v}}\,\vec{b}$ is perpendicular to $\vec{v}$. As in Figure NUMBER, the difference $2\text{proj}_{\vec{v}}\,\vec{b} - \vec{b}$ reflects $\vec{b}$ *over* $\vec{v}$.

We can expand our reflection formula as follows:

$$\begin{aligned}
2\text{proj}_{\vec{v}}\,\vec{b} - \vec{b} &= 2\frac{\vec{v}\cdot\vec{b}}{\vec{v}\cdot\vec{v}}\vec{v} - \vec{b} \text{ by definition of projection}\\
&= 2\vec{v}\cdot\frac{\vec{v}^\top\vec{b}}{\vec{v}^\top\vec{v}} - \vec{b} \text{ using matrix notation}\\
&= \left(\frac{2\vec{v}\vec{v}^\top}{\vec{v}^\top v} - I_{n\times n}\right)\vec{b}\\
&\equiv -H_{\vec{v}}\vec{b} \text{ where the negative is introduced to align with other treatments}
\end{aligned}$$

Thus, we can think of reflecting $\vec{b}$ over $\vec{v}$ as applying a linear operator $-H_{\vec{v}}$ to $\vec{b}$! Of course, $H_{\vec{v}}$ without the negative is still orthogonal, so we will use it from now on.

Suppose we are doing the first step of forward substitution during Gaussian elimination. Then, we wish to pre-multiply A by a matrix that takes the first column of A, which we will denote $\vec{a}$, to some multiple of the first identity vector $\vec{e}_1$. In other words, we want for some $c \in \mathbb{R}$:

$$\begin{aligned}
c\vec{e}_1 &= H_{\vec{v}}\vec{a}\\
&= \left(I_{n\times n} - \frac{2\vec{v}\vec{v}^\top}{\vec{v}^\top v}\right)\vec{a}\\
&= \vec{a} - 2\vec{v}\frac{\vec{v}^\top\vec{a}}{\vec{v}^\top v}
\end{aligned}$$

Moving terms around shows

$$\vec{v} = (\vec{a} - c\vec{e}_1)\cdot\frac{\vec{v}^\top\vec{v}}{2\vec{v}^\top\vec{a}}$$

In other words, $\vec{v}$ must be parallel to the difference $\vec{a} - c\vec{e}_1$. In fact, scaling $\vec{v}$ does not affect the formula for $H_{\vec{v}}$, so we can *choose* $\vec{v} = \vec{a} - c\vec{e}_1$. Then, for our relationship to hold we must have

$$\begin{aligned}
1 &= \frac{\vec{v}^\top\vec{v}}{2\vec{v}^\top\vec{a}}\\
&= \frac{\|\vec{a}\|^2 - 2c\vec{e}_1\cdot a + c^2}{2(\vec{a}\cdot\vec{a} - c\vec{e}_1\cdot\vec{a})}\\
\text{Or, } 0 &= \|\vec{a}\|^2 - c^2 \implies c = \pm\|\vec{a}\|
\end{aligned}$$

With this choice of c, we have shown:

$$H_{\vec{v}}A = \begin{pmatrix} c & \times & \times & \times\\ 0 & \times & \times & \times\\ \vdots & \vdots & \vdots & \vdots\\ 0 & \times & \times & \times \end{pmatrix}$$

We have just accomplished a step akin to forward elimination using only orthogonal matrices!

Proceeding, in the notation of CITE during the k-th step of triangularization we have a vector $\vec{a}$ that we can split into two components:

$$\vec{a} = \begin{pmatrix} \vec{a}_1 \\ \vec{a}_2 \end{pmatrix}$$

Here, $\vec{a}_1 \in \mathbb{R}^k$ and $\vec{a}_2 \in \mathbb{R}^{m-k}$. We wish to find $\vec{v}$ such that

$$H_{\vec{v}}\vec{a} = \begin{pmatrix} \vec{a}_1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

Following a parallel derivation to the one above, it is easy to show that

$$\vec{v} = \begin{pmatrix} \vec{0} \\ \vec{a}_2 \end{pmatrix} - c\vec{e}_k$$

accomplishes exactly this transformation when $c = \pm\|\vec{a}_2\|$; we usually choose the sign of c to avoid cancellation by making it have sign opposite to that of the k-th value in $\vec{a}$.

The algorithm for Householder QR is thus fairly straightforward. For each column of A, we compute $\vec{v}$ annihilating the bottom elements of the column and apply $H_{\vec{v}}$ to A. The end result is an upper triangular matrix $R = H_{\vec{v}_n} \cdots H_{\vec{v}_1} A$. The orthogonal matrix Q is given by the product $H_{\vec{v}_1}^\top \cdots H_{\vec{v}_n}^\top$, which can be stored implicitly as a list of vectors $\vec{v}$, which fits in the lower triangle as shown above.

4.5 Reduced QR Factorization

We conclude our discussion by returning to the most general case $A\vec{x} \approx \vec{b}$ when $A \in \mathbb{R}^{m\times n}$ is not square. Notice that both algorithms we have discussed in this chapter can factor non-square matrices A into products QR, but the output is somewhat different:

- When applying Gram-Schmidt, we do column operations on A to obtain Q by orthogonalization. For this reason, the dimension of A is that of Q, yielding $Q \in \mathbb{R}^{m\times n}$ and $R \in \mathbb{R}^{n\times n}$.
- When using Householder reflections, we obtain Q as the product of a number of $m \times m$ reflection matrices, leaving $R \in \mathbb{R}^{m\times n}$.

Suppose we are in the typical case for least-squares, for which $m \gg n$. We still prefer to use the Householder method due to its numerical stability, but now the $m \times m$ matrix Q might be too large to store! Thankfully, we know that R is upper triangular. For instance, consider the structure of a 5×3 matrix R:

$$R = \left(\begin{array}{ccc} \times & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \times \\ \hline 0 & 0 & 0 \\ 0 & 0 & 0 \end{array}\right)$$

It is easy to see that anything below the upper $n \times n$ square of R must be zero, yielding a simplification:

$$A = QR = \begin{pmatrix} Q_1 & Q_2 \end{pmatrix} \begin{pmatrix} R_1 \\ 0 \end{pmatrix} = Q_1 R_1$$

Here, $Q_1 \in \mathbb{R}^{m \times n}$ and $R_1 \in \mathbb{R}^{n \times n}$ still contains the upper triangle of R. This is called the "reduced" QR factorization of A, since the columns of Q_1 contain a basis for the column space of A rather than for all of $\mathbb{R}^m$; it takes up far less space. Notice that the discussion in §4.2.1 still applies, so the reduced QR factorization can be used for least-squares in a similar fashion.

4.6 Problems

- tridiagonalization with Householder
- Givens
- Underdetermined QR

Chapter 5

Eigenvectors

We turn our attention now to a *nonlinear* problem about matrices: Finding their eigenvalues and eigenvectors. Eigenvectors $\vec{x}$ and their corresponding eigenvalues λ of a square matrix A are determined by the equation $A\vec{x} = \lambda\vec{x}$. There are many ways to see that this problem is nonlinear. For instance, there is a *product* of unknowns λ and $\vec{x}$, and to avoid the trivial solution $\vec{x} = \vec{0}$ we constrain $\|\vec{x}\| = 1$; this constraint is circular rather than linear. Thanks to this structure, our methods for finding eigenspaces will be considerably different from techniques for solving and analyzing linear systems of equations.

5.1 Motivation

Despite the arbitrary-looking form of the equation $A\vec{x} = \lambda\vec{x}$, the problem of finding eigenvectors and eigenvalues arises naturally in many circumstances. We motivate our discussion with a few examples below.

5.1.1 Statistics

Suppose we have machinery for collecting several statistical observations about a collection of items. For instance, in a medical study we may collect the age, weight, blood pressure, and heart rate of 100 patients. Then, each patient i can be represented by a point $\vec{x}_i$ in $\mathbb{R}^4$ storing these four values.

Of course, such statistics may exhibit strong correlation. For instance, patients with higher blood pressure may be likely to have higher weights or heart rates. For this reason, although we collected our data in $\mathbb{R}^4$, in reality it may—to some approximate degree—live in a lower-dimensional space better capturing the relationships between the different variables.

For now, suppose that in fact there exists a *one*-dimensional space approximating our dataset. Then, we expect all the data points to be nearly parallel to some vector $\vec{v}$, so that each can be written as $\vec{x}_i \approx c_i\vec{v}$ for different $c_i \in \mathbb{R}$. From before, we know that the best approximation of $\vec{x}_i$ parallel to $\vec{v}$ is $\text{proj}_{\vec{v}}\, \vec{x}_i$:

$$\begin{aligned}\text{proj}_{\vec{v}}\, \vec{x}_i &= \frac{\vec{x}_i \cdot \vec{v}}{\vec{v} \cdot \vec{v}}\vec{v} \text{ by definition}\\ &= (\vec{x}_i \cdot \hat{v})\hat{v} \text{ since } \vec{v} \cdot \vec{v} = \|\vec{v}\|^2\end{aligned}$$

Here, we define $\hat{v} \equiv \vec{v}/\|\vec{v}\|$. Of course, the magnitude of $\vec{v}$ does not matter for the problem at hand, so it is reasonable to search over the space of unit vectors $\hat{v}$.

Following the pattern of least squares, we have a new optimization problem:

$$\begin{aligned} \text{minimize } & \sum_i \|\vec{x}_i - \text{proj}_{\hat{v}}\, \vec{x}_i\|^2 \\ \text{such that } & \|\hat{v}\| = 1 \end{aligned}$$

We can simplify our optimization objective a bit:

$$\begin{aligned} \sum_i \|\vec{x}_i - \text{proj}_{\hat{v}}\, \vec{x}_i\|^2 &= \sum_i \|\vec{x}_i - (\vec{x}_i \cdot \hat{v})\hat{v}\|^2 \text{ by definition of projection} \\ &= \sum_i \left(\|\vec{x}_i\|^2 - (\vec{x}_i \cdot \hat{v})^2\right) \text{ since } \|\hat{v}\| = 1 \text{ and } \|\vec{w}\|^2 = \vec{w} \cdot \vec{w} \\ &= \text{const.} - \sum_i (\vec{x}_i \cdot \hat{v})^2 \end{aligned}$$

This derivation shows that we can solve an equivalent optimization problem:

$$\begin{aligned} \text{maximize } & \|X^\top \hat{v}\|^2 \\ \text{such that } & \|\hat{v}\|^2 = 1, \end{aligned}$$

where the columns of X are the vectors $\vec{x}_i$. Notice that $\|X^\top \hat{v}\|^2 = \hat{v}^\top X X^\top \hat{v}$, so by Example 0.27 the vector $\hat{v}$ corresponds to the eigenvector of $XX^\top$ with the highest eigenvalue. The vector $\hat{v}$ is known as the first *principal component* of the dataset.

5.1.2 Differential Equations

Many physical forces can be written as functions of position. For instance, the force between two particles at positions $\vec{x}$ and $\vec{y}$ in $\mathbb{R}^3$ exerted by a spring can be written as $k(\vec{x} - \vec{y})$ by Hooke's Law; such spring forces are used to approximate forces holding cloth together in many simulation systems. Although these forces are not necessarily *linear* in position, we often approximate them in a linear fashion. In particular, in a physical system with n particles encode the positions of all the particles simultaneously in a vector $\vec{X} \in \mathbb{R}^{3n}$. Then, if we assume such an approximation we can write that the forces in the system are approximately $\vec{F} \approx A\vec{X}$ for some matrix A.

Recall Newton's second law of motion $F = ma$, or force equals mass times acceleration. In our context, we can write a diagonal *mass matrix* $M \in \mathbb{R}^{3n \times 3n}$ containing the mass of each particle in the system. Then, we know $\vec{F} = M\vec{X}''$, where prime denotes differentiation in time. Of course, $\vec{X}'' = (\vec{X}')'$, so in the end we have a *first*-order system of equations:

$$\frac{d}{dt}\begin{pmatrix} \vec{X} \\ \vec{V} \end{pmatrix} = \begin{pmatrix} 0 & I_{3n\times 3n} \\ M^{-1}A & 0 \end{pmatrix}\begin{pmatrix} \vec{X} \\ \vec{V} \end{pmatrix}$$

Here, we simultaneously compute both positions in $\vec{X} \in \mathbb{R}^{3n}$ and velocities $\vec{V} \in \mathbb{R}^{3n}$ of all n particles as functions of time.

More generally, differential equations of the form $\vec{x}' = A\vec{x}$ appear in many contexts, including simulation of cloth, springs, heat, waves, and other phenomena. Suppose we know eigenvectors

$\vec{x}_1, \ldots, \vec{x}_k$ of A, such that $A\vec{x}_i = \lambda_i \vec{x}_i$. If we write the initial condition of the differential equation in terms of the eigenvectors, as

$$\vec{x}(0) = c_1\vec{x}_1 + \cdots + c_k\vec{x}_k,$$

then the solution to the equation can be written in closed form:

$$\vec{x}(t) = c_1 e^{\lambda_1 t}\vec{x}_1 + \cdots + c_k e^{\lambda_k t}\vec{x}_k,$$

This solution is easy to check by hand. That is, if we write the initial conditions of this differential equation in terms of the eigenvectors of A, then we know its solution for all times $t \geq 0$ for free. Of course, this formula is not the end of the story for simulation: Finding the complete set of eigenvectors of A is expensive, and A may change over time.

5.2 Spectral Embedding

Suppose we have a collection of n items in a dataset and a measure $w_{ij} \geq 0$ of how similar each pair of elements i and j are; we will assume $w_{ij} = w_{ji}$. For instance, maybe we are given a collection of photographs and use w_{ij} to compare the similarity of their color distributions. We might wish to sort the photographs based on their similarity to simplify viewing and exploring the collection.

One model for ordering the collection might be to assign a number x_i for each item i, asking that similar objects are assigned similar numbers. We can measure how well an assignment groups similar objects by using the energy

$$E(\vec{x}) = \sum_{ij} w_{ij}(x_i - x_j)^2.$$

That is, $E(\vec{x})$ asks that items i and j with high similarity scores w_{ij} get mapped to nearby values.

Of course, minimizing $E(\vec{x})$ with no constraints gives an obvious minimum: $x_i = \text{const.}$ for all i. Adding a constraint $\|\vec{x}\| = 1$ does *not* remove this constant solution! In particular, taking $x_i = 1/\sqrt{n}$ for all i gives $\|\vec{x}\| = 1$ and $E(\vec{x}) = 0$ in an uninteresting way. Thus, we must remove this case as well:

$$\begin{aligned} \text{minimize } & E(\vec{x}) \\ \text{such that } & \|\vec{x}\|^2 = 1 \\ & \vec{1} \cdot \vec{x} = 0 \end{aligned}$$

Notice that our second constraint asks that the sum of $\vec{x}$ is zero.

Once again we can simplify the energy:

$$\begin{aligned} E(\vec{x}) &= \sum_{ij} w_{ij}(x_i - x_j)^2 \\ &= \sum_{ij} w_{ij}(x_i^2 - 2x_ix_j + x_j^2) \\ &= \sum_i a_i x_i^2 - 2\sum_{ij} w_{ij}x_ix_j + \sum_j b_j x_j^2 \\ &\qquad \text{for } a_i \equiv \sum_j w_{ij} \text{ and } b_j \equiv \sum_i w_{ij} \\ &= \vec{x}^\top (A - 2W + B)\vec{x} \text{ where } \operatorname{diag}(A) = \vec{a} \text{ and } \operatorname{diag}(B) = \vec{b} \\ &= \vec{x}^\top (2A - 2W)\vec{x} \text{ by symmetry of } W \end{aligned}$$

It is easy to check that $\vec{1}$ is an eigenvector of $2A - 2W$ with eigenvalue 0. More interestingly, the eigenvector corresponding to the *second*-smallest eigenvalue corresponds to the solution of our minimization objective above! (TODO: Add KKT proof from lecture)

5.3 Properties of Eigenvectors

We have established a variety of applications in need of eigenspace computation. Before we can explore algorithms for this purpose, however, we will more closely examine the structure of the eigenvalue problem.

We can begin with a few definitions that likely are evident at this point:

Definition 5.1 (Eigenvalue and eigenvector)**.** *An* eigenvector $\vec{x} \neq \vec{0}$ *of a matrix* $A \in \mathbb{R}^{n\times n}$ *is any vector satisfying* $A\vec{x} = \lambda\vec{x}$ *for some* $\lambda \in \mathbb{R}$*; the corresponding* λ *is known as an* eigenvalue. Complex *eigenvalues and eigenvectors satisfy the same relationships with* $\lambda \in \mathbb{C}$ *and* $\vec{x} \in \mathbb{C}^n$.

Definition 5.2 (Spectrum and spectral radius)**.** *The* spectrum *of* A *is the set of eigenvalues of* A*. The* spectral radius $\rho(A)$ *is the eigenvalue* λ *maximizing* $|\lambda|$.

The scale of an eigenvector is not important. In particular, scaling an eigenvector $\vec{x}$ by c yields $A(c\vec{x}) = cA\vec{x} = c\lambda\vec{x} = \lambda(c\vec{x})$, so $c\vec{x}$ is an eigenvector with the same eigenvalue. We often restrict our search by adding a constraint $\|\vec{x}\| = 1$. Even this constraint does not completely relieve ambiguity, since now $\pm\vec{x}$ are both eigenvectors with the same eigenvalue.

The algebraic properties of eigenvectors and eigenvalues easily could fill a book. We will limit our discussion to a few important theorems that affect the design of numerical algorithms; we will follow development of CITE AXLER. First, we should check that every matrix has at least one eigenvector so that our search is not in vain. Our usual strategy is to notice that if λ is an eigenvalue such that $A\vec{x} = \lambda\vec{x}$, then $(A - \lambda I_{n\times n})\vec{x} = \vec{0}$; thus, λ is an eigenvalue exactly when the matrix $A - \lambda I_{n\times n}$ is not full-rank.

Lemma 5.1 (CITE Theorem 2.1)**.** *Every matrix* $A \in \mathbb{R}^{n\times n}$ *has at least one (complex) eigenvector.*

Proof. Take any vector $\vec{x} \in \mathbb{R}^n\backslash\{\vec{0}\}$. The set $\{\vec{x}, A\vec{x}, A^2\vec{x}, \cdots, A^n\vec{x}\}$ must be linearly dependent because it contains $n+1$ vectors in n dimensions. So, there exist constants $c_0, \ldots, c_n \in \mathbb{R}$ with $c_n \neq 0$ such that

$$\vec{0} = c_0\vec{x} + c_1A\vec{x} + \cdots + c_nA^n\vec{x}.$$

We can write down a polynomial

$$f(z) = c_0 + c_1z + \cdots + c_nz^n.$$

By the Fundamental Theorem of Algebra, there exist n roots $z_i \in \mathbb{C}$ such that

$$f(z) = c_n(z - z_1)(z - z_2)\cdots(z - z_n).$$

Then, we have:

$$\begin{aligned}\vec{0} &= c_0\vec{x} + c_1A\vec{x} + \cdots + c_nA^n\vec{x}\\ &= (c_0I_{n\times n} + c_1A + \cdots + c_nA^n)\vec{x}\\ &= c_n(A - z_1I_{n\times n})\cdots(A - z_nI_{n\times n})\vec{x} \text{ by our factorization}\end{aligned}$$

Thus, at least one $A - z_iI_{n\times n}$ has a null space, showing that there exists $\vec{v}$ with $A\vec{v} = z_i\vec{v}$, as needed. □

There is one additional fact worth checking to motivate our discussion of eigenvector computation:

Lemma 5.2 (CITE Proposition 2.2)**.** *Eigenvectors corresponding to different eigenvalues must be linearly independent.*

Proof. Suppose this is not the case. Then there exist eigenvectors $\vec{x}_1, \cdots, \vec{x}_k$ with distinct eigenvalues $\lambda_1, \cdots, \lambda_k$ that are linearly dependent. This implies that there are coefficients $c_1, \ldots, c_k$ not all zero with $\vec{0} = c_1\vec{x}_1 + \cdots + c_k\vec{x}_k$. If we premultiply by the matrix $(A - \lambda_2 I_{n\times n})\cdots(A - \lambda_k I_{n\times n})$, we find:

$$\begin{aligned}\vec{0} &= (A - \lambda_2 I_{n\times n})\cdots(A - \lambda_k I_{n\times n})(c_1\vec{x}_1 + \cdots + c_k\vec{x}_k)\\ &= c_1(\lambda_1 - \lambda_2)\cdots(\lambda_1 - \lambda_k)\vec{x}_1 \text{ since } A\vec{x}_i = \lambda_i\vec{x}_i\end{aligned}$$

Since all the λ_i's are distinct, this shows $c_1 = 0$. A similar proof shows that the rest of the c_i's have to be zero, contradicting linear dependence. □

This lemma shows that an $n \times n$ matrix can have at most n distinct eigenvalues, since a set of n eigenvalues yields n linearly independent vectors. The maximum number of linearly independent eigenvectors corresponding to a single eigenvalue λ is known as the *geometric multiplicity* of λ.

It is not true, however, that a matrix has to have *exactly n* linearly independent eigenvectors. This is the case for many matrices, which we will call *nondefective*:

Definition 5.3 (Nondefective)**.** *A matrix $A \in \mathbb{R}^{n\times n}$ is* nondefective *or* diagonalizable *if its eigenvectors span $\mathbb{R}^n$.*

We call such a matrix diagonalizable for the following reason: If a matrix is diagonalizable, then it has n eigenvectors $\vec{x}_1, \ldots, \vec{x}_n \in \mathbb{R}^n$ with corresponding (possibly non-unique) eigenvalues $\lambda_1, \ldots, \lambda_n$. Take the columns of X to be the vectors $\vec{x}_i$, and define D to be the diagonal matrix with eigenvalues $\lambda_1, \ldots, \lambda_n$ along the diagonal. Then, by definition of eigenvalues we have $AX = XD$; this is simply a "stacked" version of $A\vec{x}_i = \lambda_i\vec{x}_i$. In other words,

$$D = X^{-1}AX,$$

meaning A is diagonalized by a *similarity transformation* $A \mapsto X^{-1}AX$:

Definition 5.4 (Similar matrices)**.** *Two matrices A and B are* similar *if there exists T with $B = T^{-1}AT$.*

Similar matrices have the same eigenvalues, since if $B\vec{x} = \lambda x$, then $T^{-1}AT\vec{x} = \lambda\vec{x}$. Equivalently, $A(T\vec{x}) = \lambda(T\vec{x})$, showing $T\vec{x}$ is an eigenvector with eigenvalue λ.

5.3.1 Symmetric and Positive Definite Matrices

Unsurprisingly given our special consideration of normal matrices $A^\top A$, symmetric and/or positive definite matrices enjoy special eigenvector structure. If we can verify either of these properties, specialized algorithms can be used to extract their eigenvectors more quickly.

First, we can prove a property of symmetric matrices that obliterates the need for complex arithmetic. We begin by making a generalization of symmetric matrices to matrices in $\mathbb{C}^{n\times n}$:

Definition 5.5 (Complex conjugate)**.** *The* complex conjugate *of a number $z \equiv a + bi \in \mathbb{C}$ is $\bar{z} \equiv a - bi$.*

Definition 5.6 (Conjugate transpose). *The* conjugate transpose *of $A \in \mathbb{C}^{m\times n}$ is $A^H \equiv \bar{A}^\top$.*

Definition 5.7 (Hermitian matrix). *A matrix $A \in \mathbb{C}^{n\times n}$ is* Hermitian *if $A = A^H$.*

Notice that a symmetric matrix $A \in \mathbb{R}^{n\times n}$ is automatically Hermitian because it has no complex part. With this slight generalization in place, we can prove a symmetry property for eigenvalues. Our proof will make use of the dot product of vectors in $\mathbb{C}^n$, given by

$$\langle \vec{x}, \vec{y} \rangle = \sum_i x_i \bar{y}_i,$$

where $\vec{x}, \vec{y} \in \mathbb{C}^n$. Notice that once again this definition coincides with $\vec{x} \cdot \vec{y}$ when $\vec{x}, \vec{y} \in \mathbb{R}^n$. For the most part, properties of this inner product coincide with those of the dot product on $\mathbb{R}^n$, a notable exception being that $\langle \vec{v}, \vec{w} \rangle = \overline{\langle \vec{w}, \vec{v} \rangle}$.

Lemma 5.3. *All eigenvalues of Hermitian matrices are real.*

Proof. Suppose $A \in \mathbb{C}^{n\times n}$ is Hermitian with $A\vec{x} = \lambda\vec{x}$. By scaling we can assume $\|\vec{x}\|^2 = \langle \vec{x}, \vec{x} \rangle = 1$. Then, we have:

$$\begin{aligned}
\lambda &= \lambda\langle \vec{x}, \vec{x} \rangle \text{ since } \vec{x} \text{ has norm } 1 \\
&= \langle \lambda\vec{x}, \vec{x} \rangle \text{ by linearity of } \langle \cdot, \cdot \rangle \\
&= \langle A\vec{x}, \vec{x} \rangle \text{ since } A\vec{x} = \lambda\vec{x} \\
&= (A\vec{x})^\top \bar{\vec{x}} \text{ by definition of } \langle \cdot, \cdot \rangle \\
&= \vec{x}^\top \overline{(\bar{A}^\top \vec{x})} \text{ by expanding the product and using } \overline{ab} = \bar{a}\bar{b} \\
&= \langle \vec{x}, A^H \vec{x} \rangle \text{ by definition of } A^H \text{ and } \langle \cdot, \cdot \rangle \\
&= \langle \vec{x}, A\vec{x} \rangle \text{ since } A = A^H \\
&= \bar{\lambda}\langle \vec{x}, \vec{x} \rangle \text{ since } A\vec{x} = \lambda\vec{x} \\
&= \bar{\lambda} \text{ since } \vec{x} \text{ has norm } 1
\end{aligned}$$

Thus $\lambda = \bar{\lambda}$, which can happen only if $\lambda \in \mathbb{R}$, as needed. □

Symmetric and Hermitian matrices also enjoy a special orthogonality property for their eigenvectors:

Lemma 5.4. *Eigenvectors corresponding to distinct eigenvalues of Hermitian matrices must be orthogonal.*

Proof. Suppose $A \in \mathbb{C}^{n\times n}$ is Hermitian, and suppose $\lambda \neq \mu$ with $A\vec{x} = \lambda\vec{x}$ and $A\vec{y} = \mu\vec{y}$. By the previous lemma we know $\lambda, \mu \in \mathbb{R}$. Then, $\langle A\vec{x}, \vec{y} \rangle = \lambda\langle \vec{x}, \vec{y} \rangle$. But since A is Hermitian we can also write $\langle A\vec{x}, \vec{y} \rangle = \langle \vec{x}, A^H\vec{y} \rangle = \langle \vec{x}, A\vec{y} \rangle = \mu\langle \vec{x}, \vec{y} \rangle$. Thus, $\lambda\langle \vec{x}, \vec{y} \rangle = \mu\langle \vec{x}, \vec{y} \rangle$. Since $\lambda \neq \mu$, we must have $\langle \vec{x}, \vec{y} \rangle = 0$. □

Finally, we can state without proof a crowning result of linear algebra, the Spectral Theorem. This theorem states that no symmetric or Hermitian matrix can be defective, meaning that an $n \times n$ matrix satisfying this property has exactly n orthogonal eigenvectors.

Theorem 5.1 (Spectral Theorem). *Suppose $A \in \mathbb{C}^{n\times n}$ is Hermitian (if $A \in \mathbb{R}^{n\times n}$, suppose it is symmetric). Then, A has exactly n orthonormal eigenvectors $\vec{x}_1, \cdots, \vec{x}_n$ with (possibly repeated) eigenvalues $\lambda_1, \ldots, \lambda_n$. In other words, there exists an orthonormal matrix X of eigenvectors and diagonal matrix D of eigenvalues such that $D = X^\top AX$.*

This theorem implies that any vector $\vec{y} \in \mathbb{R}^n$ can be decomposed into a linear combination of the eigenvectors of a Hermitian matrix A. Many calculations are easier in this basis, as shown below:

Example 5.1 (Computation using eigenvectors)**.** *Take $\vec{x}_1, \ldots, \vec{x}_n \in \mathbb{R}^n$ to be the unit-length eigenvectors of symmetric matrix $A \in \mathbb{R}^{n\times n}$. Suppose we wish to solve $A\vec{y} = \vec{b}$. We can write*

$$\vec{b} = c_1\vec{x}_1 + \cdots + c_n\vec{x}_n,$$

where $c_i = \vec{b} \cdot \vec{x}_i$ by orthonormality. It is easy to guess the following solution:

$$\vec{y} = \frac{c_1}{\lambda_1}\vec{x}_1 + \cdots + \frac{c_n}{\lambda_n}\vec{x}_n.$$

In particular, we find:

$$\begin{aligned}
A\vec{y} &= A\left(\frac{c_1}{\lambda_1}\vec{x}_1 + \cdots + \frac{c_n}{\lambda_n}\vec{x}_n\right)\\
&= \frac{c_1}{\lambda_1}A\vec{x}_1 + \cdots + \frac{c_n}{\lambda_n}A\vec{x}_n\\
&= c_1\vec{x}_1 + \cdots + c_n\vec{x}_n\\
&= \vec{b}, \text{ as desired.}
\end{aligned}$$

The calculation above is both a positive and negative result. It shows that given the eigenvectors of symmetric A, operations like inversion are straightforward. On the flip side, this means that finding the full set of eigenvectors of a symmetric matrix A is "at least" as difficult as solving $A\vec{x} = \vec{b}$.

Returning from our foray into the complex numbers, we return to real numbers to prove one final useful if straightforward fact about positive definite matrices:

Lemma 5.5. *All eigenvalues of positive definite matrices are nonnegative.*

Proof. Take $A \in \mathbb{R}^{n\times n}$ positive definite, and suppose $A\vec{x} = \lambda\vec{x}$ with $\|\vec{x}\| = 1$. By positive definiteness, we know $\vec{x}^\top A\vec{x} \geq 0$. But, $\vec{x}^\top A\vec{x} = \vec{x}^\top(\lambda\vec{x}) = \lambda\|\vec{x}\|^2 = \lambda$, as needed. $\square$

5.3.2 Specialized Properties[1]

Characteristic Polynomial

Recall that the determinant of a matrix $\det A$ satisfies the relationship that $\det A \neq 0$ if and only if A is invertible. Thus, one way to find eigenvalues of a matrix is to find roots of the *characteristic polynomial*

$$p_A(\lambda) = \det(A - \lambda I_{n\times n}).$$

We will not define determinants in our discussion here, but simplifying p_A reveals that it is an n-th degree polynomial in λ. This provides an alternative reason why there are at most n distinct eigenvalues, since there are at most n roots of this function.

From this construction, we can define the *algebraic multiplicity* of an eigenvalue as its multiplicity as a root of p_A. It is easy to see that the algebraic multiplicity is at least as large as the geometric

[1]This section can be skipped if readers lack sufficient background but is included for completeness.

multiplicity. If the algebraic multiplicity is 1, the root is called *simple*, because it corresponds to a single eigenvector that is linearly dependent with any others. Eigenvalues for which the algebraic and geometric multiplicities are not equal are called *defective*.

In numerical analysis we avoid discussing the determinant of a matrix. While it is a convenient theoretical construction, its practical use is limited. Determinants are difficult to compute. In fact, eigenvalue algorithms do not attempt to find roots of p_A since doing so would require evaluation of a determinant. Furthermore, the determinant $\det A$ has *nothing* to do with the conditioning of A, so near-zero determinant of $\det(A - \lambda I_{n\times n})$ might not show that λ is nearly an eigenvalue of A.

Jordan Normal Form

We can only diagonalize a matrix when it has a full eigenspace. All matrices, however, are similar to a matrix in Jordan normal form, which has the following form:

- Nonzero values are on the diagonal entries a_{ii} and on the "superdiagonal" $a_{i(i+1)}$.
- Diagonal values are eigenvalues repeated as many times as their multiplicity; the matrix is block diagonal about these clusters.
- Off-diagonal values are 1 or 0.

Thus, the shape looks something like the following

$$\begin{pmatrix} \lambda_1 & 1 & & & & & \\ & \lambda_1 & 1 & & & & \\ & & \lambda_1 & & & & \\ & & & \lambda_2 & 1 & & \\ & & & & \lambda_2 & & \\ & & & & & \lambda_3 & \\ & & & & & & \ddots \end{pmatrix}$$

Jordan normal form is attractive theoretically because it always exists, but the 1/0 structure is discrete and unstable under numerical perturbation.

5.4 Computing Eigenvalues

The computation and estimation of the eigenvalues of a matrix is a well-studied problem with many potential solutions. Each solution is tuned for a different situation, and achieving maximum conditioning or speed requires experimentation with several techniques. Here, we cover a few of the most popular and straightforward solutions to the eigenvalue problem frequently encountered in practice.

5.4.1 Power Iteration

For now, suppose that $A \in \mathbb{R}^{n\times n}$ is symmetric. Then, by the spectral theorem we can write eigenvectors $\vec{x}_1, \ldots, \vec{x}_n \in \mathbb{R}^n$; we sort them such that their corresponding eigenvalues satisfy $|\lambda_1| \geq |\lambda_2| \geq \cdots \geq |\lambda_n|$.

Suppose we take an arbitrary vector $\vec{v}$. Since the eigenvectors of A span $\mathbb{R}^n$, we can write:

$$\vec{v} = c_1\vec{x}_1 + \cdots + c_n\vec{x}_n.$$

Then,

$$\begin{aligned}
A\vec{v} &= c_1 A\vec{x}_1 + \cdots + c_n A\vec{x}_n \\
&= c_1\lambda_1\vec{x}_1 + \cdots + c_n\lambda_n\vec{x}_n \text{ since } A\vec{x}_i = \lambda_i\vec{x}_i \\
&= \lambda_1\left(c_1\vec{x}_1 + \frac{\lambda_2}{\lambda_1}c_2\vec{x}_2 + \cdots + \frac{\lambda_n}{\lambda_1}c_n\vec{x}_n\right) \\
A^2\vec{v} &= \lambda_1^2\left(c_1\vec{x}_1 + \left(\frac{\lambda_2}{\lambda_1}\right)^2 c_2\vec{x}_2 + \cdots + \left(\frac{\lambda_n}{\lambda_1}\right)^2 c_n\vec{x}_n\right) \\
&\vdots \\
A^k\vec{v} &= \lambda_1^k\left(c_1\vec{x}_1 + \left(\frac{\lambda_2}{\lambda_1}\right)^k c_2\vec{x}_2 + \cdots + \left(\frac{\lambda_n}{\lambda_1}\right)^k c_n\vec{x}_n\right)
\end{aligned}$$

Notice that as $k \to \infty$, the ratio $({}^{\lambda_i}/{}_{\lambda_1})^k \to 0$ unless $\lambda_i = \lambda_1$, since λ_1 has the largest magnitude of any eigenvalue by definition. Thus, if $\vec{x}$ is the projection of $\vec{v}$ onto the space of eigenvectors with eigenvalues λ_1, then as $k \to \infty$ the following approximation holds more and more exactly:

$$A^k\vec{v} \approx \lambda_1^k\vec{x}.$$

This observation leads to an exceedingly simple algorithm for computing an eigenvector $\vec{x}$ of A corresponding to the largest eigenvalue λ_1:

1. Take $\vec{v}_1 \in \mathbb{R}^n$ to be an arbitrary nonzero vector.
2. Iterate until convergence for increasing k:

$$\vec{v}_k = A\vec{v}_{k-1}$$

This algorithm, known as *power iteration*, will produce vectors $\vec{v}_k$ more and more parallel to the desired $\vec{x}_1$. It is guaranteed to converge, even when A is asymmetric, although the proof of this fact is more involved than the derivation above. The one time that this technique may fail is if we accidentally choose $\vec{v}_1$ such that $c_1 = 0$, but the odds of this occurring are slim to none.

Of course , if $|\lambda_1| > 1$, then $\|\vec{v}_k\| \to \infty$ as $k \to \infty$, an undesirable property for floating point arithmetic. Recall that we only care about the *direction* of the eigenvector rather than its magnitude, so scaling has no effect on the quality of our solution. Thus, to avoid this divergence situation we can simply normalize at each step, producing the *normalized power iteration* algorithm:

1. Take $\vec{v}_1 \in \mathbb{R}^n$ to be an arbitrary nonzero vector.
2. Iterate until convergence for increasing k:

$$\begin{aligned}
\vec{w}_k &= A\vec{v}_{k-1} \\
\vec{v}_k &= \frac{\vec{w}_k}{\|\vec{w}_k\|}
\end{aligned}$$

Notice that we did not decorate the norm $\|\cdot\|$ with a particular subscript. Mathematically, *any* norm will suffice for preventing the divergence issue, since we have shown that all norms on $\mathbb{R}^n$ are equivalent. In practice, we often use the *infinity* norm $\|\cdot\|_\infty$; in this case it is easy to check that $\|\vec{w}_k\| \to |\lambda_1|$.

5.4.2 Inverse Iteration

We now have a strategy for finding the *largest*-magnitude eigenvalue λ_1. Suppose A is invertible, so that we can evaluate $\vec{y} = A^{-1}\vec{v}$ by solving $A\vec{y} = \vec{v}$ using techniques covered in previous chapters.

If $A\vec{x} = \lambda\vec{x}$, then $\vec{x} = \lambda A^{-1}\vec{x}$, or equivalently

$$A^{-1}\vec{x} = \frac{1}{\lambda}\vec{x}.$$

Thus, we have shown that $1/\lambda$ is an eigenvalue of A^{-1} with eigenvector $\vec{x}$. Notice that if $|a| \geq |b|$ then $|b|^{-1} \geq |a|^{-1}$ for any $a, b \in \mathbb{R}$, so the smallest-magnitude eigenvalue of A is the *largest*-magnitude eigenvector of A^{-1}. This observation yields a strategy for finding λ_n rather than λ_1 called *inverse power iteration*:

1. Take $\vec{v}_1 \in \mathbb{R}^n$ to be an arbitrary nonzero vector.
2. Iterate until convergence for increasing k:
   (a) Solve for $\vec{w}_k$: $A\vec{w}_k = \vec{v}_{k-1}$
   (b) Normalize: $\vec{v}_k = \frac{\vec{w}_k}{\|\vec{w}_k\|}$

We repeatedly are solving systems of equations using the same matrix A, which is a perfect application of factorization techniques from previous chapters. For instance, if we write $A = LU$, then we could formulate an equivalent but considerably more efficient version of inverse power iteration:

1. Factor $A = LU$
2. Take $\vec{v}_1 \in \mathbb{R}^n$ to be an arbitrary nonzero vector.
3. Iterate until convergence for increasing k:
   (a) Solve for $\vec{y}_k$ by forward substitution: $L\vec{y}_k = \vec{v}_{k-1}$
   (b) Solve for $\vec{w}_k$ by back substitution: $U\vec{w}_k = \vec{y}_k$
   (c) Normalize: $\vec{v}_k = \frac{\vec{w}_k}{\|\vec{w}_k\|}$

5.4.3 Shifting

Suppose λ_2 is the eigenvalue with second-largest magnitude of A. Given our original derivation of power iteration, it is easy to see that power iteration converges fastest when $|\lambda_2/\lambda_1|$ is small, since in this case the power $(\lambda_2/\lambda_1)^k$ decays quickly. Contrastingly, if this ratio is nearly 1 it may take many iterations of power iteration before a single eigenvector is isolated.

If the eigenvalues of A are $\lambda_1, \ldots, \lambda_n$, then it is easy to see that the eigenvalues of $A - \sigma I_{n\times n}$ are $\lambda_1 - \sigma, \ldots, \lambda_n - \sigma$. Then, one strategy for making power iteration converge quickly is to choose σ such that:

$$\left|\frac{\lambda_2 - \sigma}{\lambda_1 - \sigma}\right| < \left|\frac{\lambda_2}{\lambda_1}\right|.$$

Of course, guessing such a σ can be an art, since the eigenvalues of A obviously are not known initially. Similarly, if we think that σ is near an eigenvalue of A, then $A - \sigma I_{n\times n}$ has an eigenvalue close to 0 that we can reveal by inverse iteration.

One strategy that makes use of this observation is known as *Rayleigh quotient iteration*. If we have a fixed guess at an eigenvector $\vec{x}$ of A, then by NUMBER the least-squares approximation of the corresponding eigenvalue σ is given by

$$\sigma \approx \frac{\vec{x}^\top A \vec{x}}{\|\vec{x}\|_2^2}.$$

This fraction is known as a Rayleigh quotient. Thus, we can attempt to increase convergence by iterating as follows:

1. Take $\vec{v}_1 \in \mathbb{R}^n$ to be an arbitrary nonzero vector or initial guess of an eigenvector.

2. Iterate until convergence for increasing k:

   (a) Write the current estimate of the eigenvalue

   $$\sigma_k = \frac{\vec{v}_{k-1}^\top A \vec{v}_{k-1}}{\|\vec{v}_{k-1}\|_2^2}$$

   (b) Solve for $\vec{w}_k$: $(A - \sigma_k I_{n\times n})\vec{w}_k = \vec{v}_{k-1}$

   (c) Normalize: $\vec{v}_k = \frac{\vec{w}_k}{\|\vec{w}_k\|}$

This strategy converges much faster given a good starting guess, but the matrix $A - \sigma_k I_{n\times n}$ is different each iteration and cannot be prefactored using LU or most other strategies. Thus, fewer iterations are necessary but each iteration takes more time!

5.4.4 Finding Multiple Eigenvalues

So far, we have described techniques for finding a single eigenvalue/eigenvector pair: power iteration to find the largest eigenvalue, inverse iteration to find the smallest, and shifting to target values in between. Of course, for many applications a single eigenvalue will not suffice. Thankfully we can extend our strategies to handle this case as well.

Deflation

Recall our power iteration strategy: Choose an arbitrary $\vec{v}_1$, and iteratively multiply it by A until only the largest eigenvalue λ_1 survives. Take $\vec{x}_1$ to be the corresponding eigenvector.

We were quick to dismiss an unlikely failure mode of this algorithm, however, when $\vec{v}_1 \cdot \vec{x}_1 = 0$. In this case, no matter how many times you premultiply by A you will never recover a vector parallel to $\vec{x}_1$, since you cannot amplify a zero component. The probability of choosing such a $\vec{v}_1$ is exactly zero, so in all but the most pernicious of cases power iteration remains safe.

We can turn this drawback on its head to formulate a strategy for finding more than one eigenvalue when A is symmetric. Suppose we find $\vec{x}_1$ and λ_1 via power iteration as before. Now, we restart power iteration, but before beginning project $\vec{x}_1$ out of $\vec{v}_1$. Then, since the eigenvectors of A are orthogonal, power iteration will recover the *second*-largest eigenvalue!

Due to numerical issues, it may be the case that applying A to a vector introduces a small component parallel to $\vec{x}_1$. In practice we can avoid this effect by projecting in each iteration. In the end, this strategy yields the following algorithm for computing the eigenvalues in order of descending magnitude:

- For each desired eigenvalue $\ell = 1, 2, \ldots$
  1. Take $\vec{v}_1 \in \mathbb{R}^n$ to be an arbitrary nonzero vector.
  2. Iterate until convergence for increasing k:
     (a) Project out the eigenvectors we already have computed:
     $$\vec{u}_k = \vec{v}_{k-1} - \mathrm{proj}_{\mathrm{span}\{\vec{x}_1,\ldots,\vec{x}_{\ell-1}\}}\,\vec{v}_{k-1}$$
     (b) Multiply $A\vec{u}_k = \vec{w}_k$
     (c) Normalize: $\vec{v}_k = \frac{\vec{w}_k}{\|\vec{w}_k\|}$
  3. Add the result of iteration to the set of $\vec{x}_i$'s

The inner loop is equivalent to power iteration on the matrix AP, where P projects out $\vec{x}_1, \ldots, \vec{x}_{\ell-1}$. It is easy to see that AP has the same eigenvectors as A; its eigenvalues are $\lambda_\ell, \ldots, \lambda_n$ with the remaining eigenvalues taken to zero.

More generally, the strategy of *deflation* involves modifying the matrix A so that power iteration reveals an eigenvector you have not yet computed. For instance, AP is a modification of A so that the large eigenvalues we already have computed are zeroed out.

Our projection strategy fails if A is asymmetric, since in that case its eigenvectors may not be orthogonal. Other less obvious deflation strategies can work in this case. For instance, suppose $A\vec{x}_1 = \lambda_1\vec{x}_1$ with $\|\vec{x}_1\| = 1$. Take H to be the Householder matrix such that $H\vec{x}_1 = \vec{e}_1$, the first standard basis vector. Similarity transforms once again do not affect the set of eigenvectors, so we might try conjugating by H. Consider what happens when we multiply $HAH^\top$ by $\vec{e}_1$:

$$\begin{aligned}
HAH^\top\vec{e}_1 &= HAH\vec{e}_1 \text{ since } H \text{ is symmetric}\\
&= HA\vec{x}_1 \text{ since } H\vec{x}_1 = \vec{e}_1 \text{ and } H^2 = I_{n\times n}\\
&= \lambda_1 H\vec{x}_1 \text{ since } A\vec{x}_1 = \lambda_1\vec{x}_1 \qquad = \lambda_1\vec{e}_1 \text{ by definition of } H
\end{aligned}$$

Thus, the first column of $HAH^\top$ is $\lambda_1\vec{e}_1$, showing that $HAH^\top$ has the following structure (CITE HEATH):

$$HAH^\top = \begin{pmatrix} \lambda_1 & \vec{b}^\top \\ \vec{0} & B \end{pmatrix}.$$

The matrix $B \in \mathbb{R}^{(n-1)\times(n-1)}$ has eigenvalues $\lambda_2, \ldots, \lambda_n$. Thus, another strategy for deflation is to construct smaller and smaller B matrices with each eigenvalue computed using power iteration.

QR Iteration

Deflation has the drawback that we must compute each eigenvector separately, which can be slow and can accumulate error if individual eigenvalues are not accurate. Our remaining strategies attempt to find more than one eigenvector at a time.

Recall that similar matrices A and $B = T^{-1}AT$ must have the same eigenvalues. Thus, an algorithm attempting to find the eigenvalues of A can freely apply similarity transformations to A. Of course, applying T^{-1} in general may be a difficult proposition, since it effectively would require inverting T, so we seek T matrices whose inverses are easy to apply.

One of our motivators for deriving QR factorization was that the matrix Q is *orthogonal*, satisfying $Q^{-1} = Q^\top$. Thus, Q and Q^{-1} are equally straightforward to apply, making orthogonal matrices strong choices for similarity transformations.

But which orthogonal matrix Q should we choose? Ideally Q should involve the structure of A while being straightforward to compute. It is unclear how to apply simple transformations like Householder matrices strategically to reveal multiple eigenvalues,[2] but we do know how to generate one such Q simply by factoring $A = QR$. Then, we could conjugate A by Q to find:

$$Q^{-1}AQ = Q^\top AQ = Q^\top(QR)Q = (Q^\top Q)RQ = RQ$$

Amazingly, conjugating $A = QR$ by the orthogonal matrix Q is identical to writing the product RQ!

Based on this reasoning, in the 1950s, multiple groups of European mathematicians hypothesized the same elegant iterative algorithm for finding the eigenvalues of a matrix A:

1. Take $A_1 = A$.
2. For $k = 1, 2, \ldots$
   (a) Factor $A_k = Q_k R_k$.
   (b) Write $A_{k+1} = R_k Q_k$.

By our derivation above, the matrices A_k all have the same eigenvalues as A. Furthermore, suppose the A_k's converge to some A_∞. Then, we can factor $A_\infty = Q_\infty R_\infty$, and by convergence we know $A_\infty = Q_\infty R_\infty = R_\infty Q_\infty$. By NUMBER, the eigenvalues of R_∞ are simply the values along the diagonal of R_∞, and by NUMBER the product $R_\infty Q_\infty = A_\infty$ in turn must have the same eigenvalues. Finally, by construction A_∞ has the same eigenvalues as A. So, we have shown that *if* QR iteration converges, it reveals the eigenvalues of A in a straightforward way.

Of course, the derivation above assumes that there exists A_∞ with $A_k \to A_\infty$ as $k \to \infty$. In fact, QR iteration is a stable method guaranteed to converge in many important situations, and convergence can even be improved by shifting strategies. We will not derive exact conditions here but instead can provide some intuition for why this seemingly arbitrary strategy should converge. We provide some intuition below for the symmetric case $A = A^\top$, which is easier to analyze thanks to the orthogonality of eigenvectors in this case.

Suppose the columns of A are given by $\vec{a}_1, \ldots, \vec{a}_n$, and consider the matrix A^k for large k. We can write:

$$A^k = A^{k-1} \cdot A = \begin{pmatrix} | & | & & | \\ A^{k-1}\vec{a}_1 & A^{k-1}\vec{a}_2 & \cdots & A^{k-1}\vec{a}_n \\ | & | & & | \end{pmatrix}$$

[2]More advanced techniques, however, do exactly this!

By our derivation of power iteration, the first column of A^k in general is parallel to the eigenvector $\vec{x}_1$ with largest magnitude $|\lambda_1|$ since we took a vector $\vec{a}_1$ and multiplied it by A many times. Applying our intuition from deflation, suppose we project $\vec{a}_1$ out of the second column of A^k. This vector must be nearly parallel to $\vec{x}_2$, since it is the *second*-most dominant eigenvalue! Proceeding inductively, factoring $A^k = QR$ would yield a set of near-eigenvectors as the columns of Q, in order of decreasing eigenvalue magnitude, with the corresponding eigenvalues along the diagonal of R.

Of course, computing A^k for large k takes the condition number of A to the k-th power, so QR on the resulting matrix is likely to fail; this is clear to see since *all* the columns of A^k should look like $\vec{x}_1$ for large k. We can make the following observation, however:

$$\begin{aligned}
A &= Q_1 R_1 \\
A^2 &= (Q_1 R_1)(Q_1 R_1) \\
&= Q_1 (R_1 Q_1) R_1 \\
&= Q_1 Q_2 R_2 R_1 \text{ using the notation of QR iteration above, since } A_2 = R_1 Q_1 \\
&\vdots \\
A^k &= Q_1 Q_2 \cdots Q_k R_k R_{k-1} \cdots R_1
\end{aligned}$$

Grouping the Q_i variables and the R_i variables separately provides a QR factorization of A^k. Thus, we expect the columns of $Q_1 \cdots Q_k$ to converge to the eigenvectors of A.

By a similar argument, we can find

$$\begin{aligned}
A &= Q_1 R_1 \\
A_2 &= R_1 Q_1 \text{ by our construction of QR iteration} \\
&= Q_2 R_2 \text{ by definition of the factorization} \\
A_3 &= R_2 Q_2 \text{ from QR iteration} \\
&= Q_2^\top A_2 Q_2 \text{ since } A_2 = Q_2 R_2 \\
&= Q_2^\top R_1 Q_1 Q_2 \text{ since } A_2 = R_1 Q_1 \\
&= Q_2^\top Q_1^\top A Q_1 Q_2 \text{ since } A = Q_1 R_1 \\
&\vdots \\
A_{k+1} &= Q_k^\top \cdots Q_1^\top A Q_1 \cdots Q_k \text{ inductively} \\
&= (Q_1 \cdots Q_k)^\top A (Q_1 \cdots Q_k)
\end{aligned}$$

where A_k is the k-th matrix from QR iteration. Thus, A_{k+1} is simply the matrix A conjugated by the product $\bar{Q}_k \equiv Q_1 \cdots Q_k$. We argued earlier that the columns of $\bar{Q}_k$ converge to the eigenvectors of A. Thus, since conjugating by the matrix of eigenvectors yields a diagonal matrix of eigenvalues, we know $A_{k+1} = \bar{Q}_k^\top A \bar{Q}$ will have approximate eigenvalues of A along its diagonal as $k \to \infty$.

Krylov Subspace Methods

Our justification of QR iteration involved analyzing the columns of A^k as $k \to \infty$ as an extension of power iteration. More generally, for a vector $\vec{b} \in \mathbb{R}^n$, we can examine the so-called *Krylov matrix*

$$K_k = \left(\begin{array}{ccccc} | & | & | & & | \\ \vec{b} & A\vec{b} & A^2\vec{b} & \cdots & A^{k-1}\vec{b} \\ | & | & | & & | \end{array} \right).$$

Methods analyzing K_k to find eigenvectors and eigenvalues generally are known as *Krylov subspace methods*. For instance, the *Arnoldi iteration* algorithm uses Gram-Schmidt orthogonalization to maintain an orthogonal basis $\{\vec{q}_1, \ldots, \vec{q}_k\}$ for the column space of K_k:

1. Begin by taking $\vec{q}_1$ to be an arbitrary unit-norm vector
2. For $k = 2, 3, \ldots$
   (a) Take $\vec{a}_k = A\vec{q}_{k-1}$
   (b) Project out the $\vec{q}$'s you already have computed:
   $$\vec{b}_k = \vec{a}_k - \mathrm{proj}_{\mathrm{span}\,\{\vec{q}_1,\ldots,\vec{q}_{k-1}\}}\vec{a}_k$$
   (c) Renormalize to find the next $\vec{q}_k = \vec{b}_k/\|\vec{b}_k\|$.

The matrix Q_k whose columns are the vectors found above is an orthogonal matrix with the same column space as K_k, and eigenvalue estimates can be recovered from the structure of $Q_k^\top A Q_k$. The use of Gram-Schmidt makes this technique unstable and timing gets progressively worse as k increases, however, so many extensions are needed to make it feasible. For instance, one strategy involves running some iterations of Arnoldi, using the output to generate a better guess for the initial $\vec{q}_1$, and restarting. Methods in this class are suited for problems requiring multiple eigenvectors at one of the ends of the spectrum without computing the complete set.

5.5 Sensitivity and Conditioning

As warned, we have only outlined a few eigenvalue techniques out of a rich and long-standing literature. Almost any algorithmic technique has been experimented with for finding spectra, from iterative methods to root-finding on the characteristic polynomial to methods that divide matrices into blocks for parallel processing.

Just as in linear solvers, we can evaluate the conditioning of an eigenvalue *problem* independently of the solution technique. This analysis can help understand whether a simplistic iterative scheme will be successful for finding the eigenvectors of a given matrix or if more complex methods are necessary; it is important to note that the conditioning of an eigenvalue problem is *not* the same as the condition number of the matrix for solving systems, since these are separate problems.

Suppose a matrix A has an eigenvector $\vec{x}$ with eigenvalue λ. Analyzing the conditioning of the eigenvalue problem involves analyzing the stability of $\vec{x}$ and λ to perturbations in A. To this end, we might perturb A by a small matrix δA, thus changing the set of eigenvectors. In particular, we can write eigenvectors of $A + \delta A$ as perturbations of eigenvectors of A by solving the problem

$$(A + \delta A)(\vec{x} + \delta\vec{x}) = (\lambda + \delta\lambda)(\vec{x} + \delta\vec{x}).$$

Expanding both sides yields:

$$A\vec{x} + A\delta\vec{x} + \delta A \cdot \vec{x} + \delta A \cdot \delta\vec{x} = \lambda\vec{x} + \lambda\delta\vec{x} + \delta\lambda \cdot \vec{x} + \delta\lambda \cdot \delta\vec{x}$$

Assuming δA is small, we will assume[3] that $\delta\vec{x}$ and $\delta\lambda$ also are small. Products between these variables then are negligible, yielding the following approximation:

$$A\vec{x} + A\delta\vec{x} + \delta A \cdot \vec{x} \approx \lambda\vec{x} + \lambda\delta\vec{x} + \delta\lambda \cdot \vec{x}$$

Since $A\vec{x} = \lambda\vec{x}$, we can subtract this value from both sides to find:

$$A\delta\vec{x} + \delta A \cdot \vec{x} \approx \lambda\delta\vec{x} + \delta\lambda \cdot \vec{x}$$

We now apply an analytical trick to complete our derivation. Since $A\vec{x} = \lambda\vec{x}$, we know $(A - \lambda I_{n\times n})\vec{x} = \vec{0}$, so $A - \lambda I_{n\times n}$ is not full rank. The transpose of a matrix is full-rank only if the matrix is full-rank, so we know $(A - \lambda I_{n\times n})^\top = A^\top - \lambda I_{n\times n}$ also has a null space vector $\vec{y}$. Thus $A^\top\vec{y} = \lambda\vec{y}$; we can call $\vec{y}$ the *left* eigenvector corresponding to $\vec{x}$. We can left-multiply our perturbation estimate above by $\vec{y}^\top$:

$$\vec{y}^\top(A\delta\vec{x} + \delta A \cdot \vec{x}) \approx \vec{y}^\top(\lambda\delta\vec{x} + \delta\lambda \cdot \vec{x})$$

Since $A^\top\vec{y} = \lambda\vec{y}$, we can simplify:

$$\vec{y}^\top\delta A \cdot \vec{x} \approx \delta\lambda\vec{y}^\top\vec{x}$$

Rearranging yields:

$$\delta\lambda \approx \frac{\vec{y}^\top(\delta A)\vec{x}}{\vec{y}^\top\vec{x}}$$

Assume $\|\vec{x}\| = 1$ and $\|\vec{y}\| = 1$. Then, if we take norms on both sides we find:

$$|\delta\lambda| \lessapprox \frac{\|\delta A\|_2}{|\vec{y}\cdot\vec{x}|}$$

So in general conditioning of the eigenvalue problem depends on the size of the perturbation δA–as expected–and the angle between the left and right eigenvectors $\vec{x}$ and $\vec{y}$. We can use $^1/_{\vec{x}\cdot\vec{y}}$ as an approximate condition number. Notice that $\vec{x} = \vec{y}$ when A is symmetric, yielding a condition number of 1; this reflects the fact that the eigenvectors of symmetric matrices are orthogonal and thus maximally separated.

5.6 Problems

[3]This assumption should be checked in a more rigorous treatment!

Chapter 6

Singular Value Decomposition

In Chapter 5, we derived a number of algorithms for computing the eigenvalues and eigenvectors of matrices $A \in \mathbb{R}^{n\times n}$. Having developed this machinery, we complete our initial discussion of numerical linear algebra by deriving and making use of one final matrix factorization that exists for *any* matrix $A \in \mathbb{R}^{m\times n}$: the singular value decomposition (SVD).

6.1 Deriving the SVD

For $A \in \mathbb{R}^{m\times n}$, we can think of the function $\vec{x} \mapsto A\vec{x}$ as a map taking points in $\mathbb{R}^n$ to points in $\mathbb{R}^m$. From this perspective, we might ask what happens to the geometry of $\mathbb{R}^n$ in the process, and in particular the effect A has on lengths of and angles between vectors.

Applying our usual starting point for eigenvalue problems, we can ask the effect that A has on the lengths of vectors by examining critical points of the ratio

$$R(\vec{x}) = \frac{\|A\vec{x}\|}{\|\vec{x}\|}$$

over various values of $\vec{x}$. Scaling $\vec{x}$ does not matter, since

$$R(\alpha\vec{x}) = \frac{\|A\cdot\alpha\vec{x}\|}{\|\alpha\vec{x}\|} = \frac{|\alpha|}{|\alpha|}\cdot\frac{\|A\vec{x}\|}{\|\vec{x}\|} = \frac{\|A\vec{x}\|}{\|\vec{x}\|} = R(\vec{x}).$$

Thus, we can restrict our search to $\vec{x}$ with $\|\vec{x}\| = 1$. Furthermore, since $R(\vec{x}) \geq 0$, we can instead consider $[R(\vec{x})]^2 = \|A\vec{x}\|^2 = \vec{x}^\top A^\top A\vec{x}$. As we have shown in previous chapters, however, critical points of $\vec{x}^\top A^\top A\vec{x}$ subject to $\|\vec{x}\| = 1$ are exactly the eigenvectors $\vec{x}_i$ satisfying $A^\top A\vec{x}_i = \lambda_i\vec{x}_i$; notice $\lambda_i \geq 0$ and $\vec{x}_i\cdot\vec{x}_j = 0$ when $i \neq j$ since $A^\top A$ is symmetric and positive semidefinite.

Based on our use of the function R, the $\{\vec{x}_i\}$ basis is a reasonable one for studying the geometric effects of A. Returning to this original goal, define $\vec{y}_i \equiv A\vec{x}_i$. We can make an additional observation about $\vec{y}_i$ revealing even stronger eigenvalue structure:

$$\begin{aligned}
\lambda_i\vec{y}_i &= \lambda_i\cdot A\vec{x}_i \text{ by definition of } \vec{y}_i\\
&= A(\lambda_i\vec{x}_i)\\
&= A(A^\top A\vec{x}_i) \text{ since } \vec{x}_i \text{ is an eigenvector of } A^\top A\\
&= (AA^\top)(A\vec{x}_i) \text{ by associativity}\\
&= (AA^\top)\vec{y}_i
\end{aligned}$$

Thus, we have two cases:

1. When $\lambda_i \neq 0$, then $\vec{y}_i \neq \vec{0}$. In this case, $\vec{x}_i$ is an eigenvector of $A^\top A$ and $\vec{y}_i = A\vec{x}_i$ is a corresponding eigenvector of $AA^\top$ with $\|\vec{y}_i\| = \|A\vec{x}_i\| = \sqrt{\|A\vec{x}_i\|^2} = \sqrt{\vec{x}_i^\top A^\top A\vec{x}_i} = \sqrt{\lambda_i}\|\vec{x}_i\|$.
2. When $\lambda_i = 0$, $\vec{y}_i = \vec{0}$.

An identical proof shows that if $\vec{y}$ is an eigenvector of $AA^\top$, then $\vec{x} \equiv A^\top \vec{y}$ is either zero or an eigenvector of $A^\top A$ with the same eigenvalue.

Take k to be the number of strictly positive eigenvalues $\lambda_i > 0$ discussed above. By our construction above, we can take $\vec{x}_1, \ldots, \vec{x}_k \in \mathbb{R}^n$ to be eigenvectors of $A^\top A$ and corresponding eigenvectors $\vec{y}_1, \ldots, \vec{y}_k \in \mathbb{R}^m$ of $AA^\top$ such that

$$\begin{aligned} A^\top A\vec{x}_i &= \lambda_i \vec{x}_i \\ AA^\top \vec{y}_i &= \lambda_i \vec{y}_i \end{aligned}$$

for eigenvalues $\lambda_i > 0$; here we normalize such that $\|\vec{x}_i\| = \|\vec{y}_i\| = 1$ for all i. Following traditional notation, we can define matrices $\bar{V} \in \mathbb{R}^{n\times k}$ and $\bar{U} \in \mathbb{R}^{m\times k}$ whose columns are $\vec{x}_i$'s and $\vec{y}_i$'s, resp.

We can examine the effect of these new basis matrices on A. Take $\vec{e}_i$ to be the i-th standard basis vector. Then,

$$\begin{aligned} \bar{U}^\top A\bar{V}\vec{e}_i &= \bar{U}^\top A\vec{x}_i \text{ by definition of } \bar{V} \\ &= \frac{1}{\lambda_i}\bar{U}^\top A(\lambda_i\vec{x}_i) \text{ since we assumed } \lambda_i > 0 \\ &= \frac{1}{\lambda_i}\bar{U}^\top A(A^\top A\vec{x}_i) \text{ since } \vec{x}_i \text{ is an eigenvector of } A^\top A \\ &= \frac{1}{\lambda_i}\bar{U}^\top (AA^\top)A\vec{x}_i \text{ by associativity} \\ &= \frac{1}{\sqrt{\lambda_i}}\bar{U}^\top (AA^\top)\vec{y}_i \text{ since we rescaled so that } \|\vec{y}_i\| = 1 \\ &= \sqrt{\lambda_i}\bar{U}^\top \vec{y}_i \text{ since } AA^\top\vec{y}_i = \lambda_i\vec{y}_i \\ &= \sqrt{\lambda_i}\vec{e}_i \end{aligned}$$

Take $\bar{\Sigma} = \operatorname{diag}(\sqrt{\lambda_1}, \ldots, \sqrt{\lambda_k})$. Then, the derivation above shows that $\bar{U}^\top A\bar{V} = \bar{\Sigma}$.

Complete the columns of $\bar{U}$ and $\bar{V}$ to $U \in \mathbb{R}^{m\times m}$ and $V \in \mathbb{R}^{n\times n}$ by adding orthonormal vectors $\vec{x}_i$ and $\vec{y}_i$ with $A^\top A\vec{x}_i = \vec{0}$ and $AA^\top\vec{y}_i = \vec{0}$, resp. In this case it is easy to show $U^\top AV\vec{e}_i = \vec{0}$ and/or $\vec{e}_i^\top U^\top AV = \vec{0}^\top$. Thus, if we take

$$\Sigma_{ij} \equiv \begin{cases} \sqrt{\lambda_i} & i = j \text{ and } i \leq k \\ 0 & \text{otherwise} \end{cases}$$

then we can extend our previous relationship to show $U^\top AV = \Sigma$, or equivalently

$$A = U\Sigma V^\top.$$

This factorization is exactly the *singular value decomposition* (SVD) of A. The columns of U span the column space of A and are called its *left singular vectors*; the columns of V span its row space and are the *right singular vectors*. The diagonal elements σ_i of Σ are the *singular values* of A; usually they are sorted such that $\sigma_1 \geq \sigma_2 \geq \cdots \geq 0$. Both U and V are orthogonal matrices.

The SVD provides a complete geometric characterization of the action of A. Since U and V are orthogonal, they can be thought of as rotation matrices; as a diagonal matrix, Σ simply scales individual coordinates. Thus, *all* matrices $A \in \mathbb{R}^{m\times n}$ are a composition of a rotation, a scale, and a second rotation.

6.1.1 Computing the SVD

Recall that the columns of V simply are the eigenvectors of $A^\top A$, so they can be computed using techniques discussed in the previous chapter. Since $A = U\Sigma V^\top$, we know $AV = U\Sigma$. Thus, the columns of U corresponding to nonzero singular values in Σ simply are normalized columns of AV; the remaining columns satisfy $AA^\top \vec{u}_i = \vec{0}$, which can be solved using LU factorization.

This strategy is by no means the most efficient or stable approach to computing the SVD, but it works reasonably well for many applications. We will omit more specialized approaches to finding the SVD but note that that many are simple extensions of power iteration and other strategies we already have covered that operate without forming $A^\top A$ or $AA^\top$ explicitly.

6.2 Applications of the SVD

We devote the remainder of this chapter introducing many applications of the SVD. The SVD appears countless times in both the theory and practice of numerical linear linear algebra, and its importance hardly can be exaggerated.

6.2.1 Solving Linear Systems and the Pseudoinverse

In the special case where $A \in \mathbb{R}^{n\times n}$ is square and invertible, it is important to note that the SVD can be used to solve the linear problem $A\vec{x} = \vec{b}$. In particular, we have $U\Sigma V^\top \vec{x} = \vec{b}$, or

$$\vec{x} = V\Sigma^{-1}U^\top \vec{b}.$$

In this case Σ is a square diagonal matrix, so Σ^{-1} simply is the matrix whose diagonal entries are $1/\sigma_i$.

Computing the SVD is far more expensive than most of the linear solution techniques we introduced in Chapter 2, so this initial observation mostly is of theoretical interest. More generally, suppose we wish to find a least-squares solution to $A\vec{x} \approx \vec{b}$, where $A \in \mathbb{R}^{m\times n}$ is not necessarily square. From our discussion of the normal equations, we know that $\vec{x}$ must satisfy $A^\top A\vec{x} = A^\top \vec{b}$. Thus far, we mostly have disregarded the case when A is "short" or "underdetermined," that is, when A has more columns than rows. In this case the solution to the normal equations is nonunique.

To cover all three cases, we can solve an optimization problem of the following form:

$$\begin{array}{rl} \text{minimize} & \|\vec{x}\|^2 \\ \text{such that} & A^\top A\vec{x} = A^\top \vec{b} \end{array}$$

In words, this optimization asks that $\vec{x}$ satisfy the normal equations with the least possible norm.

Now, let's write $A = U\Sigma V^\top$. Then,

$$\begin{aligned} A^\top A &= (U\Sigma V^\top)^\top (U\Sigma V^\top) \\ &= V\Sigma^\top U^\top U\Sigma V^\top \text{ since } (AB)^\top = B^\top A^\top \\ &= V\Sigma^\top \Sigma V^\top \text{ since } U \text{ is orthogonal} \end{aligned}$$

Thus, asking that $A^\top A\vec{x} = A^\top \vec{b}$ is the same as asking

$$V\Sigma^\top \Sigma V^\top \vec{x} = V\Sigma U^\top \vec{b}$$
$$\text{Or equivalently, } \Sigma\vec{y} = \vec{d}$$

if we take $\vec{d} \equiv U^\top \vec{b}$ and $\vec{y} \equiv V^\top \vec{x}$. Notice that $\|\vec{y}\| = \|\vec{x}\|$ since U is orthogonal, so our optimization becomes:

$$\begin{aligned} \text{minimize} \quad & \|\vec{y}\|^2 \\ \text{such that} \quad & \Sigma\vec{y} = \vec{d} \end{aligned}$$

Since Σ is diagonal, however, the condition $\Sigma\vec{y} = \vec{d}$ simply states $\sigma_i y_i = d_i$; so, whenever $\sigma_i \neq 0$ we must have $y_i = {}^{d_i}/_{\sigma_i}$. When $\sigma_i = 0$, there is *no* constraint on y_i, so since we are minimizing $\|\vec{y}\|^2$ we might as well take $y_i = 0$. In other words, the solution to this optimization is $\vec{y} = \Sigma^+\vec{d}$, where $\Sigma^+ \in \mathbb{R}^{n\times m}$ has the following form:

$$\Sigma^+_{ij} \equiv \begin{cases} 1/\sigma_i & i = j, \sigma_i \neq 0, \text{ and } i \leq k \\ 0 & \text{otherwise} \end{cases}$$

This form in turn yields $\vec{x} = V\vec{y} = V\Sigma^+\vec{d} = V\Sigma^+U^\top\vec{b}$.

With this motivation, we make the following definition:

Definition 6.1 (Pseudoinverse). *The* pseudoinverse *of* $A = U\Sigma V^\top \in \mathbb{R}^{m\times n}$ *is* $A^+ \equiv V\Sigma^+U^\top \in \mathbb{R}^{n\times m}$.

Our derivation above shows that the pseudoinverse of A enjoys the following properties:

- When A is square and invertible, $A^+ = A^{-1}$.
- When A is overdetermined, $A^+\vec{b}$ gives the least-squares solution to $A\vec{x} \approx \vec{b}$.
- When A is underdetermined, $A^+\vec{b}$ gives the least-squares solution to $A\vec{x} \approx \vec{b}$ with minimal (Euclidean) norm.

In this way, we finally are able to unify the underdetermined, fully-determined, and overdetermined cases of $A\vec{x} \approx \vec{b}$.

6.2.2 Decomposition into Outer Products and Low-Rank Approximations

If we expand out the product $A = U\Sigma V^\top$, it is easy to show that this relationship implies:

$$A = \sum_{i=1}^{\ell} \sigma_i \vec{u}_i \vec{v}_i^\top,$$

where $\ell \equiv \min\{m, n\}$, and $\vec{u}_i$ and $\vec{v}_i$ are the i-th columns of U and V, resp. Our sum only goes to $\min\{m, n\}$ since we know that the remaining columns of U or V will be zeroed out by Σ.

This expression shows that any matrix can be decomposed as the sum of *outer products* of vectors:

Definition 6.2 (Outer product). *The* outer product *of $\vec{u} \in \mathbb{R}^m$ and $\vec{v} \in \mathbb{R}^n$ is the matrix $\vec{u} \otimes \vec{v} \equiv \vec{u}\vec{v}^\top \in \mathbb{R}^{m\times n}$.*

Suppose we wish to write the product $A\vec{x}$. Then, instead we could write:

$$\begin{aligned} A\vec{x} &= \left(\sum_{i=1}^{\ell} \sigma_i \vec{u}_i \vec{v}_i^\top\right)\vec{x} \\ &= \sum_{i=1}^{\ell} \sigma_i \vec{u}_i (\vec{v}_i^\top \vec{x}) \\ &= \sum_{i=1}^{\ell} \sigma_i (\vec{v}_i \cdot \vec{x}) \vec{u}_i \text{ since } \vec{x}\cdot\vec{y} = \vec{x}^\top \vec{y} \end{aligned}$$

So, applying A to $\vec{x}$ is the same as linearly combining the $\vec{u}_i$ vectors with weights $\sigma_i(\vec{v}_i \cdot \vec{x})$. This strategy for computing $A\vec{x}$ can provide considerably savings when the number of nonzero σ_i values is relatively small. Furthermore, we can ignore small values of σ_i, effectively truncating this sum to *approximate* $A\vec{x}$ with less work.

Similarly, from §6.2.1 we can write the pseudoinverse of A as:

$$A^+ = \sum_{\sigma_i \neq 0} \frac{\vec{v}_i \vec{u}_i^\top}{\sigma_i}.$$

Obviously we can apply the same trick to evaluate $A^+\vec{x}$, and in fact we can approximate $A^+\vec{x}$ by only evaluating those terms in the sum for which σ_i is relatively *small*. In practice, we compute the singular values σ_i as square roots of eigenvalues of $A^\top A$ or $AA^\top$, and methods like power iteration can be used to reveal a partial rather than full set of eigenvalues. Thus, if we are going to have to solve a number of least-squares problems $A\vec{x}_i \approx \vec{b}_i$ for different $\vec{b}_i$ and are satisfied with an approximation of $\vec{x}_i$, it can be valuable first to compute the smallest σ_i values first and use the approximation above. This strategy also avoids ever having to compute or store the full A^+ matrix and can be accurate when A has a wide range of singular values.

Returning to our original notation $A = U\Sigma V^\top$, our argument above effectively shows that a potentially useful approximation of A is $\tilde{A} \equiv U\tilde{\Sigma}V^\top$, where $\tilde{\Sigma}$ rounds small values of Σ to zero. It is easy to check that the column space of $\tilde{A}$ has dimension equal to the number of nonzero values on the diagonal of $\tilde{\Sigma}$. In fact, this approximation is not an *ad hoc* estimate but rather solves a difficult optimization problem post by the following famous theorem (stated without proof):

Theorem 6.1 (Eckart-Young, 1936). *Suppose $\tilde{A}$ is obtained from $A = U\Sigma V^\top$ by truncating all but the k largest singular values σ_i of A to zero. Then $\tilde{A}$ minimizes both $\|A - \tilde{A}\|_{Fro}$ and $\|A - \tilde{A}\|_2$ subject to the constraint that the column space of $\tilde{A}$ have at most dimension k.*

6.2.3 Matrix Norms

Constructing the SVD also enables us to return to our discussion of matrix norms from §3.3.1. For example, recall that we defined the *Frobenius* norm of A as

$$\|A\|_{\text{Fro}}^2 \equiv \sum_{ij} a_{ij}^2.$$

If we write $A = U\Sigma V^\top$, we can simplify this expression:

$$\begin{aligned}
\|A\|_{\text{Fro}}^2 &= \sum_j \|A\vec{e}_j\|^2 \text{ since this product is the } j\text{-th column of } A\\
&= \sum_j \|U\Sigma V^\top \vec{e}_j\|^2, \text{ substituting the SVD}\\
&= \sum_j \vec{e}_j^\top V\Sigma^2 V^\top \vec{e}_j \text{ since } \|\vec{x}\|^2 = \vec{x}^\top\vec{x} \text{ and } U \text{ is orthogonal}\\
&= \|\Sigma V^\top\|_{\text{Fro}}^2 \text{ by the same logic}\\
&= \|V\Sigma\|_{\text{Fro}}^2 \text{ since a matrix and its transpose have the same Frobenius norm}\\
&= \sum_j \|V\Sigma\vec{e}_j\|^2 = \sum_j \sigma_j^2 \|V\vec{e}_j\|^2 \text{ by diagonality of } \Sigma\\
&= \sum_j \sigma_j^2 \text{ since } V \text{ is orthogonal}
\end{aligned}$$

Thus,the Frobenius norm of $A \in \mathbb{R}^{m\times n}$ is the sum of the squares of its singular values.

This result is of theoretical interest, but practically speaking the basic definition of the Frobenius norm is already straightforward to evaluate. More interestingly, recall that the induced two-norm of A is given by

$$\|A\|_2^2 = \max\{\lambda : \text{there exists } \vec{x} \in \mathbb{R}^n \text{ with } A^\top A\vec{x} = \lambda\vec{x}\}.$$

Now that we have studied eigenvalue problems, we realize that this value is the square root of the largest eigenvalue of $A^\top A$, or equivalently

$$\|A\|_2 = \max\{\sigma_i\}.$$

In other words, we can read the two-norm of A directly from its eigenvalues.

Similarly, recall that the condition number of A is given by cond $A = \|A\|_2\|A^{-1}\|_2$. By our derivation of A^+, the singular values of A^{-1} must be the reciprocals of the singular values of A. Combining this with our simplification of $\|A\|_2$ yields:

$$\text{cond } A = \frac{\sigma_{\max}}{\sigma_{\min}}.$$

This expression yields a strategy for evaluating the conditioning of A. Of course, computing $\sigma_{\min}$ requires solving systems $A\vec{x} = \vec{b}$, a process which in itself may suffer from poor conditioning of A; if this is an issue, conditioning can be bounded and approximated by using various approximations of the singular values of A.

6.2.4 The Procrustes Problem and Alignment

Many techniques in computer vision involve the alignment of three-dimensional shapes. For instance, suppose we have a three-dimensional scanner that collects two point clouds of the same rigid object from different views. A typical task might be to align these two point clouds into a single coordinate frame.

Since the object is rigid, we expect there to be some rotation matrix R and translation $\vec{t} \in \mathbb{R}^3$ such that that rotating the first point cloud by R and then translating by $\vec{t}$ aligns the two data sets. Our job is to estimate these two objects.

If the two scans overlap, the user or an automated system may mark n corresponding points that correspond between the two scans; we can store these in two matrices $X_1, X_2 \in \mathbb{R}^{3\times n}$. Then, for each column $\vec{x}_{1i}$ of X_1 and $\vec{x}_{2i}$ of X_2, we expect $R\vec{x}_{1i} + \vec{t} = \vec{x}_{2i}$. We can write an energy function measuring how much this relationship holds true:

$$E \equiv \sum_i \|R\vec{x}_{1i} + \vec{t} - \vec{x}_{2i}\|^2.$$

If we fix R and minimize with respect to $\vec{t}$, optimizing E obviously becomes a least-squares problem. Now, suppose we optimize for R with $\vec{t}$ fixed. This is the same as minimizing $\|RX_1 - X_2^t\|_{\text{Fro}}$, where the columns of X_2^t are those of X_2 translated by $\vec{t}$, subject to R being a 3×3 rotation matrix, that is, that $R^\top R = I_{3\times 3}$. This is known as the *orthogonal Procrustes problem.*

To solve this problem, we will introduce the *trace* of a square matrix as follows:

Definition 6.3 (Trace)**.** *The* trace *of $A \in \mathbb{R}^{n\times n}$ is the sum of its diagonal:*

$$tr(A) \equiv \sum_i a_{ii}.$$

It is straightforward to check that $\|A\|_{\text{Fro}}^2 = \text{tr}(A^\top A)$. Thus, we can simplify E as follows:

$$\begin{aligned}\|RX_1 - X_2^t\|_{\text{Fro}}^2 &= \text{tr}((RX_1 - X_2^t)^\top (RX_1 - X_2^t)) \\ &= \text{tr}(X_1^\top X_1 - X_1^\top R^\top X_2^t - X_2^{t\top} RX_1 + X_2^{t\top} X_2) \\ &= \text{const.} - 2\text{tr}(X_2^{t\top} RX_1) \\ &\qquad \text{since } \text{tr}(A+B) = \text{tr}\,A + \text{tr}\,B \text{ and } \text{tr}(A^\top) = \text{tr}(A)\end{aligned}$$

Thus, we wish to maximize $\text{tr}(X_2^{t\top} RX_1)$ with $R^\top R = I_{3\times 3}$. In the exercises, you will prove that $\text{tr}(AB) = \text{tr}(BA)$. Thus our objective can simplify slightly to $\text{tr}(RC)$ with $C \equiv X_1 X_2^{t\top}$. Applying the SVD, if we decompose $C = U\Sigma V^\top$ then we can simplify even more:

$$\begin{aligned}\text{tr}(RC) &= \text{tr}(RU\Sigma V^\top) \text{ by definition} \\ &= \text{tr}((V^\top RU)\Sigma) \text{ since } \text{tr}(AB) = \text{tr}(BA) \\ &= \text{tr}(\tilde{R}\Sigma) \text{ if we define } \tilde{R} = V^\top RU\text{, which is also orthogonal} \\ &= \sum_i \sigma_i \tilde{r}_{ii} \text{ since } \Sigma \text{ is diagonal}\end{aligned}$$

Since $\tilde{R}$ is orthogonal, its columns all have unit length. This implies that $\tilde{r}_{ii} \leq 1$, since otherwise the norm of column i would be too big. Since $\sigma_i \geq 0$ for all i, this argument shows that we can maximize $\text{tr}(RC)$ by taking $\tilde{R} = I_{3\times 3}$. Undoing our substitutions shows $R = V\tilde{R}U^\top = VU^\top$.

More generally, we have shown the following:

Theorem 6.2 (Orthogonal Procrustes). *The orthogonal matrix R minimizing $\|RX - Y\|^2$ is given by $VU^\top$, where SVD is applied to factor $XY^\top = U\Sigma V^\top$.*

Returning to the alignment problem, one typical strategy is an *alternating* approach:

1. Fix R and minimize E with respect to $\vec{t}$.
2. Fix the resulting $\vec{t}$ and minimize E with respect to R subject to $R^\top R = I_{3\times 3}$.
3. Return to step 1.

The energy E decreases with each step and thus converges to a local minimum. Since we never optimize $\vec{t}$ and R simultaneously, we cannot guarantee that the result is the smallest possible value of E, but in practice this method works well.

6.2.5 Principal Components Analysis (PCA)

Recall the setup from §5.1.1: We wish to find a low-dimensional approximation of a set of data points, which we can store in a matrix $X \in \mathbb{R}^{n\times k}$, for k observations in n dimensions. Previously, we showed that if we are allowed only a single dimension, the best possible direction is given by the dominant eigenvector of $XX^\top$.

Suppose instead we are allowed to project onto the span of d vectors with $d \leq \min\{k, n\}$ and wish to choose these vectors optimally. We could write them in an $n \times d$ matrix C; since we can apply Gram-Schmidt to any set of vectors, we can assume that the columns of C are orthonormal, showing $C^\top C = I_{d\times d}$. Since C has orthonormal columns, by the normal equations the projection of X onto the column space of C is given by $CC^\top X$.

In this setup, we wish to minimize $\|X - CC^\top X\|_{\text{Fro}}$ subject to $C^\top C = I_{d\times d}$. We can simplify our problem somewhat:

$$\begin{aligned}
\|X - CC^\top X\|_{\text{Fro}}^2 &= \text{tr}((X - CC^\top X)^\top (X - CC^\top X)) \text{ since } \|A\|_{\text{Fro}}^2 = \text{tr}(A^\top A) \\
&= \text{tr}(X^\top X - 2X^\top CC^\top X + X^\top CC^\top CC^\top X) \\
&= \text{const.} - \text{tr}(X^\top CC^\top X) \text{ since } C^\top C = I_{d\times d} \\
&= -\|C^\top X\|_{\text{Fro}}^2 + \text{const.}
\end{aligned}$$

So, equivalently we can maximize $\|C^\top X\|_{\text{Fro}}^2$; for statisticians, this shows when the rows of X have mean zero that we wish to maximize the variance of the projection $C^\top X$.

Now, suppose we factor $X = U\Sigma V^\top$. Then, we wish to maximize $\|C^\top U\Sigma V^\top\|_{\text{Fro}} = \|\tilde{C}^\top \Sigma\|_{\text{Fro}} = \|\Sigma^\top C\|_{\text{Fro}}$ by orthogonality of V if we take $\tilde{C} = CU^\top$. If the elements of $\tilde{C}$ are $\tilde{c}_{ij}$, then expanding this norm yields

$$\|\Sigma^\top \tilde{C}\|_{\text{Fro}}^2 = \sum_i \sigma_i^2 \sum_j \tilde{c}_{ij}^2.$$

By orthogonality of the columns of $\tilde{C}$, we know $\sum_i \tilde{c}_{ij}^2 = 1$ for all j and, since $\tilde{C}$ may have fewer than n columns, $\sum_j \tilde{c}_{ij}^2 \leq 1$. Thus, the coefficient next to σ_i^2 is at most 1 in the sum above, so if we sort such that $\sigma_1 \geq \sigma_2 \geq \cdots$, then clearly the maximum is achieved by taking the columns of $\tilde{C}$ to be $\vec{e}_1, \ldots, \vec{e}_d$. Undoing our change of coordinates, we see that our choice of C should be the first d columns of U.

We have shown that the SVD of X can be used to solve such a *principal components analysis* (PCA) problem. In practice the rows of X usually are shifted to have mean zero before carrying out the SVD; as shown in Figure NUMBER, this centers the dataset about the origin, providing more meaningful PCA vectors $\vec{u}_i$.

6.3 Problems

Part III

Nonlinear Techniques

Chapter 7

Nonlinear Systems

Try as we might, it simply is not possible to express all systems of equations in the linear framework we have developed over the last several chapters. It is hardly necessary to motivate the usage of logarithms, exponentials, trigonometric functions, absolute values, polynomials, and so on in practical problems, but except in a few special cases none of these functions is linear. When these functions appear, we must employ a more general if less efficient set of machinery.

7.1 Single-Variable Problems

We begin our discussion by considering problems of a single scalar variable. In particular, given a function $f(x) : \mathbb{R} \to \mathbb{R}$, we wish to develop strategies for finding points $x^* \in \mathbb{R}$ such that $f(x^*) = 0$; we call x^* a *root* of f. Single-variable problems in linear algebra are not particularly interesting; after all we can solve the equation $ax - b = 0$ in closed form as $x^* = {}^b/_a$. Solving a nonlinear equation like $y^2 + e^{\cos y} - 3 = 0$, however, is far less obvious (incidentally, the solution is $y^* = \pm 1.30246\ldots$).

7.1.1 Characterizing Problems

We no longer can assume f is linear, but without any assumption on its structure we are unlikely to make headway on solving single-variable systems. For instance, a solver is guaranteed to fail finding zeros of $f(x)$ given by

$$f(x) = \begin{cases} -1 & x \leq 0 \\ 1 & x > 0 \end{cases}$$

Or worse:

$$f(x) = \begin{cases} -1 & x \in \mathbb{Q} \\ 1 & \text{otherwise} \end{cases}$$

These examples are trivial in the sense that a rational client of root-finding software would be unlikely to expect it to succeed in this case, but far less obvious cases are not much more difficult to construct.

For this reason, we must add some "regularizing" assumptions about f providing a toehold into the possibility of designing root-finding techniques. Typical such assumptions are below, listed in increasing order of strength:

- *Continuity:* A function f is *continuous* if it can be drawn without lifting up a pen; more formally, f is continuous if the difference $f(x) - f(y)$ vanishes as $x \to y$.
- *Lipschitz:* A function f is *Lipschitz continuous* if there exists a constant C such that $|f(x) - f(y)| \leq C|x - y|$; Lipschitz functions need not be differentiable but are limited in their rates of change.
- *Differentiability:* A function f is *differentiable* if its derivative f' exists for all x.
- C^k*:* A function is C^k if it is differentiable k times and each of those k derivatives is continuous; C^∞ indicates that all derivatives of f exist and are continuous.

As we add stronger and stronger assumptions about f, we can design more effective algorithms to solve $f(x^*) = 0$. We will illustrate this effect by considering a few algorithms below.

7.1.2 Continuity and Bisection

Suppose all we know about f is that it is continuous. In this case, we can state an intuitive theorem from standard single variable calculus:

Theorem 7.1 (Intermediate Value Theorem)**.** *Suppose $f : [a, b] \to \mathbb{R}$ is continuous. Suppose $f(x) < u < f(y)$. Then, there exists z between x and y such that $f(z) = u$.*

In other words, the function f must achieve every value in between $f(x)$ and $f(y)$.

Suppose we are given as input the function f as well as two values ℓ and r such that $f(\ell) \cdot f(r) < 0$; notice this means that $f(\ell)$ and $f(r)$ have opposite sign. Then, by the Intermediate Value Theorem we know that somewhere between ℓ and r there is a root of f! This provides an obvious bisection strategy for finding x^*:

1. Compute $c = {}^{\ell+r}/_2$.
2. If $f(c) = 0$, return $x^* = c$.
3. If $f(\ell) \cdot f(c) < 0$, take $r \leftarrow c$. Otherwise take $\ell \leftarrow c$.
4. If $|r - \ell| < \varepsilon$, return $x^* \approx c$.
5. Go back to step 1

This strategy simply divides the interval $[\ell, r]$ in half iteratively, each time keeping the side in which a root is known to exist. Clearly by the Intermediate Value Theorem it converges *unconditionally*, in the sense that so long as $f(\ell) \cdot f(r) < 0$ eventually both ℓ and r are guaranteed converge to a valid root x^*.

7.1.3 Analysis of Root-Finding

Bisection is the simplest but not necessarily the most effective technique for root-finding. As with most eigenvalue methods, bisection inherently is iterative and may never provide an *exact* solution x^*. We can ask, however, how close the value c_k of c in the k-th iteration is to the root x^* that we hope to compute. This analysis will provide a baseline for comparison to other methods.

In general, suppose we can establish an error bound E_k such that the estimate x_k of the root x^* during the k-th iteration of a root-finding method satisfies $|x_k - x^*| < E_k$. Obviously any algorithm with $E_k \to 0$ represents a *convergent* scheme; the speed of convergence, however, can be characterized by the rate at which E_k approaches 0.

For example, in bisection since both c_k and x^* are in the interval $[\ell_k, r_k]$, an upper bound of error is given by $E_k \equiv |r_k - \ell_k|$. Since we divide the interval in half each iteration, we know $E_{k+1} = {}^1\!/_2 E_k$. Since E_{k+1} is linear in E_k, we say that bisection exhibits *linear* convergence.

7.1.4 Fixed Point Iteration

Bisection is guaranteed to converge to a root for any continuous function f, but if we know more about f we can formulate algorithms that can converge more quickly.

As an example, suppose we wish to find x^* satisfying $g(x^*) = x^*$; of course, this setup is equivalent to the root-finding problem since solving $f(x) = 0$ is the same as solving $f(x) + x = x$. As additional piece of information, however, we also might know that g is Lipschitz with constant $C < 1$.

The system $g(x) = x$ suggests a potential strategy we might hypothesize:

1. Take x_0 to be an initial guess of a root.
2. Iterate $x_k = g(x_{k-1})$.

If this strategy converges, clearly the result is a *fixed point* of g satisfying the criteria above.

Thankfully, the Lipschitz property ensures that this strategy converges to a root if one exists. If we take $E_k = |x_k - x^*|$, then we have the following property:

$$\begin{aligned}
E_k &= |x_k - x^*| \\
&= |g(x_{k-1}) - g(x^*)| \text{ by design of the iterative scheme and definition of } x^* \\
&\leq C|x_{k-1} - x^*| \text{ since } g \text{ is Lipschitz} \\
&= CE_{k-1}
\end{aligned}$$

Applying this statement inductively shows $E_k \leq C^k|E_0| \to 0$ as $k \to \infty$. Thus, fixed point iteration converges to the desired x^*!

In fact, if g is Lipschitz with constant $C < 1$ in a *neighborhood* $[x^* - \delta, x^* + \delta]$, then so long as x_0 is chosen in this interval fixed point iteration will converge. This is true since our expression for E_k above shows that it shrinks each iteration.

One important case occurs when g is C^1 and $|g'(x^*)| < 1$. By continuity of g' in this case, we know that there is some neighborhood $N = [x^* - \delta, x^* + \delta]$ in which $|g'(x)| < 1 - \varepsilon$ for any $x \in N$, for some choice of a sufficiently small $\varepsilon > 0$.[1] Take any $x, y \in N$. Then, we have

$$\begin{aligned}
|g(x) - g(y)| &= |g'(\theta)| \cdot |x - y| \text{ by the Mean Value Theorem of basic calculus, for some } \theta \in [x, y] \\
&< (1 - \varepsilon)|x - y|
\end{aligned}$$

This shows that g is Lipschitz with constant $1 - \varepsilon < 1$ in N. Thus, when g is continuously differentiable and $g'(x^*) < 1$, fixed point iteration will converge to x^* when the initial guess x_0 is close by.

[1]This statement is hard to parse: Make sure you understand it!

So far we have little reason to use fixed point iteration: We have shown it is guaranteed to converge only when g is Lipschitz, and our argument about the E_k's shows linear convergence like bisection. There is one case, however, in which fixed point iteration provides an advantage.

Suppose g is differentiable with $g'(x^*) = 0$. Then, the first-order term vanishes in the Taylor series for g, leaving behind:

$$g(x_k) = g(x^*) + \frac{1}{2}g''(x^*)(x_k - x^*)^2 + O\left((x_k - x^*)^3\right).$$

Thus, in this case we have:

$$\begin{aligned}
E_k &= |x_k - x^*| \\
&= |g(x_{k-1}) - g(x^*)| \text{ as before} \\
&= \frac{1}{2}|g''(x^*)|(x_{k-1} - x^*)^2 + O((x_{k-1} - x^*)^3) \text{ from the Taylor argument} \\
&\leq \frac{1}{2}(|g''(x^*)| + \varepsilon)|(x_{k-1} - x^*)^2 \text{ for some } \varepsilon \text{ so long as } x_{k-1} \text{ is close to } x^* \\
&= \frac{1}{2}(|g''(x^*)| + \varepsilon)E_{k-1}^2
\end{aligned}$$

Thus, in this case E_k is *quadratic* in E_{k-1}, so we say fixed point iteration can have *quadratic convergence*; notice this proof of quadratic convergence only holds because we already know $E_k \to 0$ from our more general convergence proof. This implies that $E_k \to 0$ much faster, so we will need fewer iterations to reach a reasonable root.

Example 7.1 (Convergence of fixed-point iteration)**.**

7.1.5 Newton's Method

We tighten our class of functions once more to derive a method that has more consistent quadratic convergence. Now, suppose again we wish to solve $f(x^*) = 0$, but now we assume that f is C^1, a slightly tighter condition than Lipschitz.

At a point $x_k \in \mathbb{R}$, since f now is differentiable we can approximate it using a tangent line:

$$f(x) \approx f(x_k) + f'(x_k)(x - x_k)$$

Solving this approximation for $f(x) \approx 0$ yields a root

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}.$$

Iterating this formula is known as *Newton's method* for root-finding, and it amounts to iteratively solving a linear approximation of the nonlinear problem.

Notice that if we define

$$g(x) = x - \frac{f(x)}{f'(x)},$$

then Newton's method amounts to fixed point iteration on g. Differentiating, we find:

$$\begin{aligned}
g'(x) &= 1 - \frac{f'(x)^2 - f(x)f''(x)}{f'(x)^2} \text{ by the quotient rule} \\
&= \frac{f(x)f''(x)}{f'(x)^2}
\end{aligned}$$

Suppose x^* is a *simple* root, meaning $f'(x^*) \neq 0$. Then, $g'(x^*) = 0$, and by our derivation of fixed-point iteration above we know that Newton's method converges quadratically to x^* for a sufficiently close initial guess. Thus, when f is differentiable with a simple root Newton's method provides a fixed-point iteration formula that is guaranteed to converge quadratically; when x^* is not simple, however, convergence can be linear or worse.

The derivation of Newton's method suggests other methods derived by using more terms in the Taylor series. For instance, "Halley's method" adds terms involving f'' to the iterations, and a class of "Householder methods" takes an arbitrary number of derivatives. These techniques offer even higher-order convergence at the cost of having to evaluate more complex iterations and the possibility of more exotic failure modes. Other methods replace Taylor series with other basic forms; for example, linear fractional interpolation uses rational functions to better approximate functions with asymptote structure.

7.1.6 Secant Method

One efficiency concern we have not addressed yet is the cost of evaluating f and its derivatives. If f is a very complicated function, we may wish to minimize the number of times we have to compute f or worse f'. Higher orders of convergence help with this problem, but we also can design numerical methods that avoid evaluating costly derivatives.

Example 7.2 (Design). *Suppose we are designing a rocket and wish to know how much fuel to add to the engine. For a given number of gallons x, we can write a function $f(x)$ giving the maximum height of the rocket; our engineers have specified that we wish to the rocket to reach a height h, so we need to solve $f(x) = h$. Evaluating $f(x)$ involves simulating a rocket as it takes off and monitoring its fuel consumption, which is an expensive proposition, and although we might suspect that f is differentiable we might not be able to evaluate f' in a practical amount of time.*

One strategy for designing lower-impact methods is to reuse data as much as possible. For instance, we easily could approximate:

$$f'(x_k) \approx \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}}.$$

That is, since we had to compute $f(x_{k-1})$ in the previous iteration, we simply use the slope to $f(x_k)$ to approximate the derivative. Certainly this approximation works well, especially when x_k's are near convergence.

Plugging our approximation into Newton's method reveals a new iterative scheme:

$$x_{k+1} = x_k - \frac{f(x_k)(x_k - x_{k-1})}{f(x_k) - f(x_{k-1})}$$

Notice that the user will have to provide two initial guesses x_0 and x_{-1} to start this scheme, or can run a single iteration of Newton to get it started.

Analyzing the secant method is somewhat more complicated than the other methods we consider because it uses both $f(x_k)$ and $f(x_{k-1})$; proof of its convergence is outside the scope of our discussion. Interestingly, error analysis reveals that error decreases at a rate of $1+\sqrt{5}/2$ (the "Golden Ratio"), between linear and quadratic; since convergence is *close* to that of Newton's method without the need for evaluating f', the secant method can provide a strong alternative.

7.1.7 Hybrid Techniques

Additional engineering can be carried out to attempt to combine the advantages of different root-finding algorithms. For instance, we might make the following observations about two methods we have discussed:

- *Bisection* is guaranteed to converge unconditionally but only does so at a linear rate.
- The *secant method* converges faster when it does reach a root, but in some cases it may not converge.

Suppose we have bracketed a root of $f(x)$ in an interval $[\ell_k, r_k]$ as in bisection. We can say that our current estimate of x^* is given by $x_k = \ell_k$ when $|f(\ell_k)| < |f(r_k)|$ and $x_k = r_k$ otherwise. If we keep track of x_k and x_{k-1}, then we could take x_{k+1} to be the next estimate of the root given by the secant method. If x_k is *outside* the interval $[\ell_k, r_k]$, however, we can replace it with ${}^{\ell_k+r_k}/_2$. This correction guarantees that $x_{k+1} \in [\ell_k, r_k]$, and regardless of the choice we can update to a valid bracket $[\ell_{k+1}, r_{k+1}]$ as in bisection by examining the sign of $f(x_{k+1})$. This algorithm is known as "Dekker's method."

The strategy above strategy attempts to combine the unconditional convergence of bisection with the stronger root estimates of the secant method. In many cases it is successful, but its convergence rate is somewhat difficult to analyze; specialized failure modes can reduce this method to linear convergence or worse–in fact, in some cases bisection surprisingly can converge more quickly! Other techniques, e.g. "Brent's method," make bisection steps more often to avoid this case and can exhibit guaranteed behavior at the cost of a somewhat more complex implementation.

7.1.8 Single-Variable Case: Summary

We now have presented and analyzed a number of methods for solving $f(x^*) = 0$ in the single-variable case. It is probably obvious at this point that we only have scraped the surface of such techniques; many iterative schemes for root-finding exist, all with different guarantees, convergence rates, and caveats. Regardless, through our experiences we can make a number of observations:

- Due to the possible generic form of f, we are unlikely to be able to find roots x^* exactly and instead settle for iterative schemes.
- We wish for the sequence x_k of root estimates to reach x^* as quickly as possible. If E_k is an error bound, then we can characterize a number of convergence situations assuming $E_k \to 0$ as $k \to \infty$. A complete list of conditions that must hold when k is large enough is below:
    1. Linear convergence: $E_{k+1} \leq CE_k$ for some $C < 1$
    2. Superlinear convergence: $E_{k+1} \leq CE_k^r$ for $r > 1$ (now we do not require $C < 1$ since if E_k is small enough, the r power can cancel the effects of C)
    3. Quadratic convergence: $E_{k+1} \leq CE_k^2$
    4. Cubic convergence: $E_{k+1} \leq CE_k^3$ (and so on)
- A method might converge more quickly but during each individual iteration require additional computation; for this reason, it may be preferable to do more iterations of a simpler method than fewer iterations of a more complex one.

7.2 Multivariable Problems

Some applications may require solving a more general problem $f(\vec{x}) = \vec{0}$ for a function $f : \mathbb{R}^n \to \mathbb{R}^m$. We have already seen one instance of this problem when solving $A\vec{x} = \vec{b}$, which is equivalent to finding roots of $f(\vec{x}) \equiv A\vec{x} - \vec{b}$, but the general case is considerably more difficult. In particular, strategies like bisection are difficult to extend since we now much guarantee that m different values are all zero *simultaneously*.

7.2.1 Newton's Method

Thankfully, one of our strategies extends in a straightforward way. Recall that for $f : \mathbb{R}^n \to \mathbb{R}^m$ we can write the *Jacobian* matrix, which gives the derivative of each component of f in each of the coordinate directions:

$$(Df)_{ij} \equiv \frac{df_i}{dx_j}$$

We can use the Jacobian of f to extend our derivation of Newton's method to multiple dimensions. In particular, the first-order approximation of f is given by:

$$f(\vec{x}) \approx f(\vec{x}_k) + Df(\vec{x}_k) \cdot (\vec{x} - \vec{x}_k).$$

Substituting the desired $f(\vec{x}) = \vec{0}$ yields the following linear system for the next iterate $\vec{x}_{k+1}$:

$$Df(\vec{x}_k) \cdot (\vec{x}_{k+1} - \vec{x}_k) = -f(\vec{x}_k)$$

This equation can be solved using the pseudoinverse when $m < n$; when $m > n$ one can attempt least-squares but the existence of a root and convergence of this technique are both unlikely. When Df is square, however, corresponding to $f : \mathbb{R}^n \to \mathbb{R}^n$, we obtain the typical iteration for Newton's method:

$$\vec{x}_{k+1} = \vec{x}_k - [Df(\vec{x}_k)]^{-1} f(\vec{x}_k),$$

where as always we do not explicitly compute the matrix $[Df(\vec{x}_k)]^{-1}$ but rather use it to signal solving a linear system.

Convergence of fixed-point methods like Newton's method that iterate $\vec{x}_{k+1} = g(\vec{x}_k)$ requires that the maximum-magnitude eigenvalue of the Jacobian Dg be less than 1. After verifying that assumption, an argument similar to the one-dimensional case shows that Newton's method can have quadratic convergence near roots $\vec{x}^*$ for which $Df(\vec{x}^*)$ is nonsingular.

7.2.2 Making Newton Faster: Quasi-Newton and Broyen

As m and n increase, Newton's method becomes very expensive. For each iteration, a *different* matrix $Df(\vec{x}_k)$ must be inverted; because it changes so often, pre-factoring $Df(\vec{x}_k) = L_k U_k$ does not help.

Some *quasi-Newton* strategies attempt to apply different approximation strategies to simplify individual iterations. For instance, one straightforward approach might reuse Df from previous iterations while recomputing $f(\vec{x}_k)$ under the assumption that the derivative does not change very quickly. We will return to these strategies when we discuss the application of Newton's method to optimization.

Another option is to attempt to parallel our derivation of the secant method. Just as the secant method still contains division, such approximations will not necessarily alleviate the need to invert a matrix, but they do make it possible to carry out optimization without explicitly calculating the Jacobian Df. Such extensions are not totally obvious, since divided differences do not yield a full approximate Jacobian matrix.

Recall, however, that the *directional derivative* of f in the direction $\vec{v}$ is given by $D_{\vec{v}}f = Df \cdot \vec{v}$. As with the secant method, we can use this observation to our advantage by asking that our approximation J of a Jacobian satisfy

$$J \cdot (\vec{x}_k - \vec{x}_{k-1}) \approx f(\vec{x}_k) - f(\vec{x}_{k-1}).$$

Broyden's method is one such extension of the secant method that keeps track not only of an estimate $\vec{x}_k$ of $\vec{x}^*$ but also a matrix J_k estimating the Jacobian; initial estimates J_0 and $\vec{x}_0$ both must be supplied. Suppose we have a previous estimate J_{k-1} of the Jacobian from the previous iteration. We now have a new data point $\vec{x}_k$ at which we have evaluated $f(\vec{x}_k)$, so we would like to update J_{k-1} to a new Jacobian J_k taking into account this new observation. One reasonable model is to ask that the new approximation be as similar to the old approximation except in the $\vec{x}_k - \vec{x}_{k-1}$ direction:

$$\begin{aligned} \text{minimize}_{J_k} \quad & \|J_k - J_{k-1}\|_{\text{Fro}}^2 \\ \text{such that} \quad & J_k \cdot (\vec{x}_k - \vec{x}_{k-1}) = f(\vec{x}_k) - f(\vec{x}_{k-1}) \end{aligned}$$

To solve this problem, define $\Delta J \equiv J_k - J_{k-1}$, $\Delta\vec{x} \equiv \vec{x}_k - \vec{x}_{k-1}$, and $\vec{d} \equiv f(\vec{x}_k) - f(\vec{x}_{k-1}) - J_{k-1} \cdot \Delta\vec{x}$. Making these substitutions yields the following form:

$$\begin{aligned} \text{minimize}_{\Delta J} \quad & \|\Delta J\|_{\text{Fro}}^2 \\ \text{such that} \quad & \Delta J \cdot \Delta\vec{x} = \vec{d} \end{aligned}$$

If we take $\vec{\lambda}$ to be a Lagrange multiplier, this minimization is equivalent to finding critical points of the Lagrangian Λ:

$$\Lambda = \|\Delta J\|_{\text{Fro}}^2 + \vec{\lambda}^\top (\Delta J \cdot \Delta\vec{x} - \vec{d})$$

Differentiating with respect to $(\Delta J)_{ij}$ shows:

$$0 = \frac{\partial \Lambda}{\partial (\Delta J)_{ij}} = 2(\Delta J)_{ij} + \lambda_i (\Delta\vec{x})_j \implies \Delta J = -\frac{1}{2}\vec{\lambda}(\Delta\vec{x})^\top$$

Substituting into $\Delta J \cdot \Delta\vec{x} = \vec{d}$ shows $\vec{\lambda}(\Delta\vec{x})^\top(\Delta\vec{x}) = -2\vec{d}$, or equivalently $\vec{\lambda} = {-2\vec{d}}/{\|\Delta\vec{x}\|^2}$. Finally, we can substitute to find:

$$\Delta J = -\frac{1}{2}\vec{\lambda}(\Delta\vec{x})^\top = \frac{\vec{d}(\Delta\vec{x})^\top}{\|\Delta x\|^2}$$

Expanding out our substitution shows:

$$\begin{aligned} J_k &= J_{k-1} + \Delta J \\ &= J_{k-1} + \frac{\vec{d}(\Delta\vec{x})^\top}{\|\Delta x\|^2} \\ &= J_{k-1} + \frac{(f(\vec{x}_k) - f(\vec{x}_{k-1}) - J_{k-1} \cdot \Delta\vec{x})}{\|\vec{x}_k - \vec{x}_{k-1}\|^2}(\vec{x}_k - \vec{x}_{k-1})^\top \end{aligned}$$

Thus, Broyden's method simply alternates between this update and the corresponding Newton step $\vec{x}_{k+1} = \vec{x}_k - J_k^{-1} f(\vec{x}_k)$. Additional efficiency in some cases can be gained by keeping track of the matrix J_k^{-1} explicitly rather than the matrix J_k, which can be updated using a similar formula.

7.3 Conditioning

We already showed in Example 1.7 that the condition number of root-finding in a single variable is:

$$\text{cond}_{x^*} f = \frac{1}{|f'(x^*)|}$$

As illustrated in Figure NUMBER, this condition number shows that the best possible situation for root-finding occurs when f is changing rapidly near x^*, since in this case perturbing x^* will make f take values far from 0.

Applying an identical argument when f is multidimensional shows a condition number of $\|Df(\vec{x}^*)\|^{-1}$. Notice that when Df is not invertible, the condition number is *infinite*. This oddity occurs because to first order perturbing $\vec{x}^*$ preserves $f(\vec{x}) = \vec{0}$, and indeed such a condition can create challenging root-finding cases like that shown in Figure NUMBER.

7.4 Problems

Many possibilities, including:

- Many possible fixed point iteration schemes for a given root-finding problem, graphical version of fixed point iteration
- Mean field iteration in ML
- Muller's method – complex roots
- Higher-order iterative methods – Householder methods
- Interpretation of eigenstuff as root-finding
- Convergence of secant method
- Roots of polynomials
- Newton-Fourier method (!)
- "Modified Newton's method in case of non-quadratic convergence"
- Convergence –¿ spectral radius for multidimensional Newton; quadratic convergence
- Sherman-Morrison update for Broyden

Chapter 8

Unconstrained Optimization

In previous chapters, we have chosen to take a largely *variational* approach to deriving standard algorithms for computational linear algebra. That is, we define an *objective function*, possibly with constraints, and pose our algorithms as a minimization or maximization problem. A sampling from our previous discussion is listed below:

Problem	**Objective**	**Constraints**
Least-squares	$E(\vec{x}) = \|A\vec{x} - \vec{b}\|^2$	None
Project $\vec{b}$ onto $\vec{a}$	$E(c) = \|c\vec{a} - \vec{b}\|$	None
Eigenvectors of symmetric matrix	$E(\vec{x}) = \vec{x}^\top A\vec{x}$	$\|\vec{x}\| = 1$
Pseudoinverse	$E(\vec{x}) = \|\vec{x}\|^2$	$A^\top A\vec{x} = A^\top\vec{b}$
Principal components analysis	$E(C) = \|X - CC^\top X\|_{\text{Fro}}$	$C^\top C = I_{d\times d}$
Broyden step	$E(J_k) = \|J_k - J_{k-1}\|_{\text{Fro}}^2$	$J_k \cdot (\vec{x}_k - \vec{x}_{k-1}) = f(\vec{x}_k) - f(\vec{x}_{k-1})$

Obviously the formulation of problems in this fashion is a powerful and general approach. For this reason, it is valuable to design algorithms that function in the absence of a special form for the energy E, in the same way that we developed strategies for finding roots of f without knowing the form of f *a priori*.

8.1 Unconstrained Optimization: Motivation

In this chapter, we will consider *unconstrained* problems, that is, problems that can be posed as minimizing or maximizing a function $f : \mathbb{R}^n \to \mathbb{R}$ without any requirements on the input. It is not difficult to encounter such problems in practice; we list a few examples below.

Example 8.1 (Nonlinear least-squares). *Suppose we are given a number of pairs (x_i, y_i) such that $f(x_i) \approx y_i$, and we wish to find the best approximating f within a particular class. For instance, we may expect that f is exponential, in which case we should be able to write $f(x) = ce^{ax}$ for some c and some a; our job is to find these parameters. One simple strategy might be to attempt to minimize the following energy:*

$$E(a, c) = \sum_i (y_i - ce^{ax_i})^2.$$

This form for E is not quadratic in a, so our linear least-squares methods do not apply.

Example 8.2 (Maximum likelihood estimation). *In machine learning, the problem of* parameter estimation *involves examining the results of a randomized experiment and trying to summarize them using a probability distribution of a particular form. For example, we might measure the height of every student in a class, yielding a list of heights h_i for each student i. If we have a lot of students, we might model the distribution of student heights using a* normal distribution*:*

$$g(h;\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(h-\mu)^2/2\sigma^2},$$

where μ is the mean of the distribution and σ is the standard deviation.

Under this normal distribution, the likelihood that we observe height h_i for student i is given by $g(h_i;\mu,\sigma)$, and under the (reasonable) assumption that the height of student i is probabilistically independent of that of student j, the probability of observing the entire set of heights observed is given by the product

$$P(\{h_1,\ldots,h_n\};\mu,\sigma) = \prod_i g(h_i;\mu,\sigma).$$

A common method for estimating the parameters μ and σ of g is to maximize P viewed as a function of μ and σ with $\{h_i\}$ fixed; this is called the maximum-likelihood estimate *of μ and σ. In practice, we usually optimize the* log likelihood $\ell(\mu,\sigma) \equiv \log P(\{h_1,\ldots,h_n\};\mu,\sigma)$*; this function has the same maxima but enjoys better numerical and mathematical properties.*

Example 8.3 (Geometric problems). *Many geometry problems encountered in graphics and vision do not reduce to least-squares energies. For instance, suppose we have a number of points $\vec{x}_1,\ldots,\vec{x}_k \in \mathbb{R}^3$. If we wish to* cluster *these points, we might wish to summarize them with a single $\vec{x}$ minimizing:*

$$E(\vec{x}) \equiv \sum_i \|\vec{x} - \vec{x}_i\|_2.$$

The $\vec{x} \in \mathbb{R}^3$ minimizing E is known as the geometric median *of $\{\vec{x}_1,\ldots,\vec{x}_k\}$. Notice that the norm of the difference $\vec{x} - \vec{x}_i$ in E is not squared, so the energy is no longer quadratic in the components of $\vec{x}$.*

Example 8.4 (Physical equilibria, adapted from CITE). *Suppose we attach an object to a set of springs; each spring is anchored at point $\vec{x}_i \in \mathbb{R}^3$ and has natural length L_i and constant k_i. In the absence of gravity, if our object is located at position $\vec{p} \in \mathbb{R}^3$, the network of springs has potential energy*

$$E(\vec{p}) = \frac{1}{2}\sum_i k_i \left(\|\vec{p} - \vec{x}_i\|_2 - L_i\right)^2$$

Equilibria of this system are given by minima of E and reflect points $\vec{p}$ at which the spring forces are all balanced. Such systems of equations are used to visualize graphs $G = (V,E)$, by attaching vertices in V with springs for each pair in E.

8.2 Optimality

Before discussing how to minimize or maximize a function, we should be clear what it is we are looking for; notice that maximizing f is the same as minimizing $-f$, so the minimization problem is sufficient for our consideration. For a particular $f : \mathbb{R}^n \to \mathbb{R}$ and $\vec{x}^* \in \mathbb{R}^n$, we need to derive *optimality conditions* that verify that $\vec{x}^*$ has the lowest possible value $f(\vec{x}^*)$.

Of course, ideally we would like to find *global* optima of f:

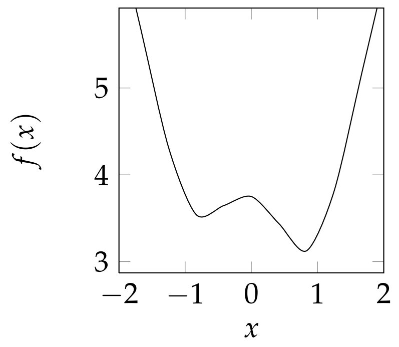

Figure 8.1: A function $f(x)$ with multiple optima.

Definition 8.1 (Global minimum)**.** *The point* $\vec{x}^* \in \mathbb{R}^n$ *is a* global minimum *of* $f : \mathbb{R}^n \to R$ *if* $f(\vec{x}^*) \leq f(\vec{x})$ *for all* $\vec{x} \in \mathbb{R}^n$.

Finding a global minimum of f without any information about the structure of f effectively requires searching in the dark. For instance, suppose an optimization algorithm identifies the local minimum near $x = -1$ in the function in Figure 8.1. It is nearly impossible to realize that there is a second, lower minimum near $x = 1$ simply by guessing x values—for all we know, there may be third even lower minimum of f at $x = 1000$!

Thus, in many cases we satisfy ourselves by finding a *local* minimum:

Definition 8.2 (Local minimum)**.** *The point* $\vec{x}^* \in \mathbb{R}^n$ *is a* local minimum *of* $f : \mathbb{R}^n \to R$ *if* $f(\vec{x}^*) \leq f(\vec{x})$ *for all* $\vec{x} \in \mathbb{R}^n$ *satisfying* $\|\vec{x} - \vec{x}^*\| < \varepsilon$ *for some* $\varepsilon > 0$.

This definition requires that $\vec{x}^*$ attains the smallest value in some *neighborhood* defined by the radius ε. Notice that local optimization algorithms have a severe limitation that they cannot guarantee that they yield the lowest possible value of f, as in Figure 8.1 if the left local minimum is reached; many strategies, heuristic and otherwise, are applied to explore the landscape of possible $\vec{x}$ values to help gain confidence that a local minimum has the best possible value.

8.2.1 Differential Optimality

A familiar story from single- and multi-variable calculus is that finding potential minima and maxima of a function $f : \mathbb{R}^n \to \mathbb{R}$ is more straightforward when f is differentiable. Recall that the gradient vector $\nabla f = (\partial f/\partial x_1, \ldots, \partial f/\partial x_n)$ points in the direction in which f increases the most; the vector $-\nabla f$ points in the direction of greatest decrease. One way to see this is to recall that near a point $\vec{x}_0 \in \mathbb{R}^n$, f looks like the linear function

$$f(\vec{x}) \approx f(\vec{x}_0) + \nabla f(\vec{x}_0) \cdot (\vec{x} - \vec{x}_0).$$

If we take $\vec{x} - \vec{x}_0 = \alpha \nabla f(\vec{x}_0)$, then we find:

$$f(\vec{x}_0 + \alpha \nabla f(\vec{x}_0)) \approx f(\vec{x}_0) + \alpha \|\nabla f(\vec{x}_0)\|^2$$

When $\|\nabla f(\vec{x}_0)\| > 0$, the sign of α determines whether f increases or decreases.

It is not difficult to formalize the above argument to show that if $\vec{x}_0$ is a local minimum, then we must have $\nabla f(\vec{x}_0) = \vec{0}$. Notice this condition is *necessary* but not *sufficient*: maxima and saddle

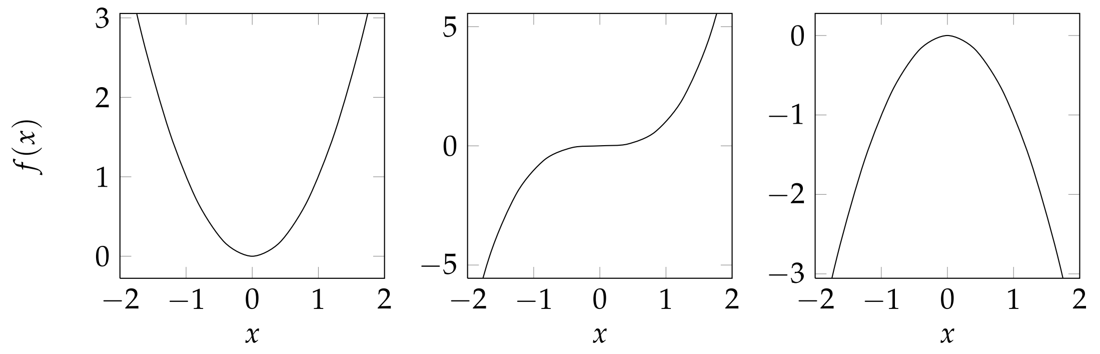

Figure 8.2: Critical points can take many forms; here we show a local minimum, a saddle point, and a local maximum.

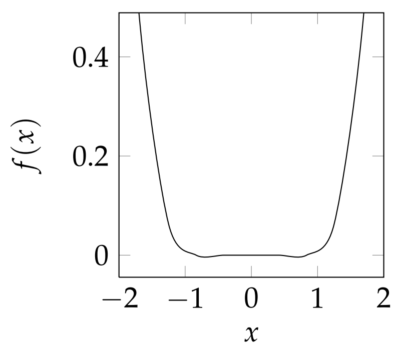

Figure 8.3: A function with many stationary points.

points also have $\nabla f(\vec{x}_0) = \vec{0}$ as illustrated in Figure 8.2. Even so, this observation about minima of differentiable functions yields a common strategy for root-finding:

1. Find points $\vec{x}_i$ satisfying $\nabla f(\vec{x}_i) = \vec{0}$.

2. Check which of these points is a local minimum as opposed to a maximum or saddle point.

Given their important role in this strategy, we give the points we seek a special name:

Definition 8.3 (Stationary point)**.** *A* stationary point *of $f : \mathbb{R}^n \to \mathbb{R}$ is a point $\vec{x} \in \mathbb{R}^n$ satisfying $\nabla f(\vec{x}) = \vec{0}$.*

That is, our strategy for minimization can be to find stationary points of f and then eliminate those that are not minima.

It is important to keep in mind when we can expect our strategies for minimization to succeed. In most cases, such as those shown in Figure 8.2, the stationary points of f are *isolated*, meaning we can write them in a discrete list $\{\vec{x}_0, \vec{x}_1, \ldots\}$. A degenerate case, however, is shown in Figure 8.3; here, the entire interval $[-1/2, 1/2]$ is composed of stationary points, making it impossible to consider them one at a time. For the most part, we will ignore such issues as degenerate cases, but will return to them when we consider the conditioning of the minimization problem.

Suppose we identify a point $\vec{x} \in \mathbb{R}$ as a stationary point of f and now wish to check if it is a local minimum. If f is twice-differentiable, one strategy we can employ is to write its *Hessian* matrix:

$$H_f(\vec{x}) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^1} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial^2 x_2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \cdots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial^2 x_n} \end{pmatrix}$$

We can add another term to our Taylor expansion of f to see the role of H_f:

$$f(\vec{x}) \approx f(\vec{x}_0) + \nabla f(\vec{x}_0) \cdot (\vec{x} - \vec{x}_0) + \frac{1}{2}(\vec{x} - \vec{x}_0)^\top H_f (\vec{x} - \vec{x}_0)$$

If we substitute a stationary point $\vec{x}^*$, then by definition we know:

$$f(\vec{x}) \approx f(\vec{x}^*) + \frac{1}{2}(\vec{x} - \vec{x}^*)^\top H_f (\vec{x} - \vec{x}^*)$$

If H_f is positive definite, then this expression shows $f(\vec{x}) \geq f(\vec{x}^*)$, and thus $\vec{x}^*$ is a local minimum. More generally, one of a few situations can occur:

- If H_f is *positive definite*, then $\vec{x}^*$ is a local minimum of f.
- If H_f is *negative definite*, then $\vec{x}^*$ is a local maximum of f.
- If H_f is *indefinite*, then $\vec{x}^*$ is a saddle point of f.
- If H_f is *not invertible*, then oddities such as the function in Figure 8.3 can occur.

Checking if a matrix is positive definite can be accomplished by checking if its Cholesky factorization exists or—more slowly—by checking that all its eigenvalues are positive. Thus, when the Hessian of f is known we can check stationary points for optimality using the list above; many optimization algorithms including the ones we will discuss simply ignore the final case and notify the user, since it is relatively unlikely.

8.2.2 Optimality via Function Properties

Occasionally, if we know more information about $f : \mathbb{R}^n \to \mathbb{R}$ we can provide optimality conditions that are stronger or easier to check than the ones above.

One property of f that has strong implications for optimization is *convexity*, illustrated in Figure NUMBER:

Definition 8.4 (Convex)**.** *A function $f : \mathbb{R}^n \to \mathbb{R}$ is* convex *when for all $\vec{x}, \vec{y} \in \mathbb{R}^n$ and $\alpha \in (0, 1)$ the following relationship holds:*

$$f((1-\alpha)\vec{x} + \alpha\vec{y}) \leq (1-\alpha)f(\vec{x}) + \alpha f(\vec{y}).$$

When the inequality is strict, the function is strictly convex.

Convexity implies that if you connect in $\mathbb{R}^n$ two points with a line, the values of f along the line are less than or equal to those you would obtain by linear interpolation.

Convex functions enjoy many strong properties, the most basic of which is the following:

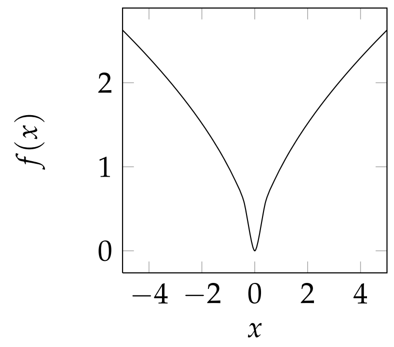

Figure 8.4: A quasiconvex function.

Proposition 8.1. *A local minimum of a convex function $f : \mathbb{R}^n \to \mathbb{R}$ is necessarily a global minimum.*

Proof. Take $\vec{x}$ to be such a local minimum and suppose there exists $\vec{x}^* \neq \vec{x}$ with $f(\vec{x}^*) < f(\vec{x})$. Then, for $\alpha \in (0, 1)$,

$$\begin{aligned} f(\vec{x} + \alpha(\vec{x}^* - \vec{x})) &\leq (1-\alpha)f(\vec{x}) + \alpha f(\vec{x}^*) \text{ by convexity} \\ &< f(\vec{x}) \text{ since } f(\vec{x}^*) < f(\vec{x}) \end{aligned}$$

But taking $\alpha \to 0$ shows that $\vec{x}$ cannot possibly be a local minimum. □

This proposition and related observations show that it is possible to check if you have reached a *global* minimum of a convex function simply by applying first-order optimality. Thus, it is valuable to check by hand if a function being optimized happens to be convex, a situation occurring surprisingly often in scientific computing; one sufficient condition that can be easier to check when f is twice differentiable is that H_f is positive definite *everywhere.*

Other optimization techniques have guarantees under other assumptions about f. For example, one weaker version of convexity is *quasi*-convexity, achieved when

$$f((1-\alpha)\vec{x} + \alpha\vec{y}) \leq \max(f(\vec{x}), f(\vec{y})).$$

An example of a quasiconvex function is shown in Figure 8.4; although it does not have the characteristic "bowl" shape of a convex function, it does have a unique optimum.

8.3 One-Dimensional Strategies

As in the last chapter, we will start with one-dimensional optimization of $f : \mathbb{R} \to \mathbb{R}$ and then expand our strategies to more general functions $f : \mathbb{R}^n \to \mathbb{R}$.

8.3.1 Newton's Method

Our principal strategy for minimizing differentiable functions $f : \mathbb{R}^n \to \mathbb{R}$ will be to find stationary points $\vec{x}^*$ satisfying $\nabla f(\vec{x}^*) = 0$. Assuming we can check whether stationary points are maxima, minima, or saddle points as a post-processing step, we will focus on the problem of finding the stationary points $\vec{x}^*$.

To this end, suppose $f : \mathbb{R} \to \mathbb{R}$ is differentiable. Then, as in our derivation of Newton's method for root-finding, we can approximate:

$$f(x) \approx f(x_k) + f'(x_k)(x - x_k) + \frac{1}{2} f''(x_k)(x - x_k)^2.$$

The approximation on the right hand side is a parabola whose vertex is located at $x_k - f'(x_k)/f''(x_k)$. Of course, in reality f is not necessarily a parabola, so Newton's method simply iterates the formula

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)}.$$

This technique is easily-analyzed given the work we already have put into understanding Newton's method for root-finding in the previous chapter. In particular, an alternative way to derive the formula above comes from root-finding on $f'(x)$, since stationary points satisfy $f'(x) = 0$. Thus, in most cases Newton's method for optimization exhibits quadratic convergence, provided the initial guess x_0 is sufficiently close to x^*.

A natural question to ask is whether the secant method can be applied in an analogous way. Our derivation of Newton's method above finds roots of f', so the secant method could be used to eliminate the evaluation of f'' but not f'; situations in which we know f' but not f'' are relatively rare. A more suitable parallel is to replace the line segments used to approximate f in the secant method with parabolas. This strategy, known as *successive parabolic interpolation,* also minimizes a quadratic approximation of f at each iteration, but rather than using $f(x_k)$, $f'(x_k)$, and $f''(x_k)$ to construct the approximation it uses $f(x_k)$, $f(x_{k-1})$, and $f(x_{k-2})$. The derivation of this technique is relatively straightforward, and it converges superlinearly.

8.3.2 Golden Section Search

We skipped over bisection in our parallel of single-variable root-finding techniques. There are many reasons for this omission. Our motivation for bisection was that it employed only the weakest assumption on f needed to find roots: continuity. The Intermediate Value Theorem does not apply to minima in any intuitive way, however, so it appears such a straightforward approach does not exist.

It is valuable, however, to have at least one minimization strategy available that does not require differentiability of f as an underlying assumption; after all, there are non-differentiable functions that have clear minima, like $f(x) \equiv |x|$ at $x = 0$. To this end, one alternative assumption might be that f is *unimodular*:

Definition 8.5 (Unimodular). *A function $f : [a, b] \to \mathbb{R}$ is* unimodular *if there exists $x^* \in [a, b]$ such that f is decreasing for $x \in [a, x^*]$ and increasing for $x \in [x^*, b]$.*

In other words, a unimodular function decreases for some time, and then begins increasing; no localized minima are allowed. Notice that functions like $|x|$ are not differentiable but still are unimodular.

Suppose we have two values x_0 and x_1 such that $a < x_0 < x_1 < b$. We can make two observations that will help us formulate an optimization technique:

- If $f(x_0) \geq f(x_1)$, then we know that $f(x) \geq f(x_1)$ for all $x \in [a, x_0]$. Thus, the interval $[a, x_0]$ can be discarded in our search for a minimum of f.

- If $f(x_1) \geq f(x_0)$, then we know that $f(x) \geq f(x_0)$ for all $x \in [x_1, b]$, and thus we can discard $[x_1, b]$.

This structure suggests a potential strategy for minimization beginning with the interval $[a, b]$ and iteratively removing pieces according to the rules above.

One important detail remains, however. Our convergence guarantee for the bisection algorithm came from the fact that we could remove half of the interval in question in each iteration. We could proceed in a similar fashion, removing a *third* of the interval each time; this requires two evaluations of f during each iteration at new x_0 and x_1 locations. If evaluating f is expensive, however, we may wish to reuse information from previous iterations to avoid at least one of those two evaluations.

For now $a = 0$ and $b = 1$; the strategies we derive below will work more generally by shifting and scaling. In the absence of more information about f, we might as well make a symmetric choice $x_0 = \alpha$ and $x_1 = 1 - \alpha$ for some $\alpha \in (0, 1/2)$. Suppose our iteration removes the rightmost interval $[x_1, b]$. Then, the search interval becomes $[0, 1 - \alpha]$, and we know $f(\alpha)$ from the previous iteration. The next iteration will divide $[0, 1 - \alpha]$ such that $x_0 = \alpha(1 - \alpha)$ and $x_1 = (1 - \alpha)^2$. If we wish to reuse $f(\alpha)$ from the previous iteration, we could set $(1 - \alpha)^2 = \alpha$, yielding:

$$\begin{aligned} \alpha &= \frac{1}{2}(3 - \sqrt{5}) \\ 1 - \alpha &= \frac{1}{2}(\sqrt{5} - 1) \end{aligned}$$

The value of $1 - \alpha \equiv \tau$ above is the *golden ratio*! It allows for the reuse of one of the function evaluations from the previous iterations; a symmetric argument shows that the same choice of α works if we had removed the left interval instead of the right one.

The *golden section search* algorithm makes use of this construction (CITE):

1. Take $\tau \equiv \frac{1}{2}(\sqrt{5} - 1)$., and initialize a and b so that f is unimodular on $[a, b]$.
2. Make an initial subdivision $x_0 = a + (1 - \tau)(b - a)$ and $x_1 = a + \tau(b - a)$.
3. Initialize $f_0 = f(x_0)$ and $f_1 = f(x_1)$.
4. Iterate until $b - a$ is sufficiently small:
   (a) If $f_0 \geq f_1$, then remove the interval $[a, x_0]$ as follows:
      - Move left side: $a \leftarrow x_0$
      - Reuse previous iteration: $x_0 \leftarrow x_1$, $f_0 \leftarrow f_1$
      - Generate new sample: $x_1 \leftarrow a + \tau(b - a)$, $f_1 \leftarrow f(x_1)$

   (b) If $f_1 > f_0$, then remove the interval $[x_1, b]$ as follows:
      - Move right side: $b \leftarrow x_1$
      - Reuse previous iteration: $x_1 \leftarrow x_0$, $f_1 \leftarrow f_0$
      - Generate new sample: $x_0 \leftarrow a + (1 - \tau)(b - a)$, $f_0 \leftarrow f(x_0)$

This algorithm clearly converges unconditionally and linearly. When f is not globally unimodal, it can be difficult to find $[a, b]$ such that f is unimodal on that interval, limiting the applications of this technique somewhat; generally $[a, b]$ is guessed by attempting to bracket a local minimum of f.

8.4 Multivariable Strategies

We continue in our parallel of our discussion of root-finding by expanding our discussion to multivariable problems. As with root-finding, multivariable problems are considerably more difficult than problems in a single variable, but they appear so many times in practice that they are worth careful consideration.

Here, we will consider only the case that $f : \mathbb{R}^n \to \mathbb{R}$ is differentiable. Optimization methods more similar to golden section search for non-differentiable functions are of limited applications and are difficult to formulate.

8.4.1 Gradient Descent

Recall from our previous discussion that $\nabla f(\vec{x})$ points in the direction of "steepest ascent" of f at $\vec{x}$; similarly, the vector $-\nabla f(\vec{x})$ is the direction of "steepest descent." If nothing else, this definition guarantees that when $\nabla f(\vec{x}) \neq \vec{0}$, for small $\alpha > 0$ we must have

$$f(\vec{x} - \alpha \nabla f(\vec{x})) \leq f(\vec{x}).$$

Suppose our current estimate of the location of the minimum of f is $\vec{x}_k$. Then, we might wish to choose $\vec{x}_{k+1}$ so that $f(\vec{x}_{k+1}) < f(\vec{x}_k)$ for an iterative minimization strategy. One way to simplify the search for $\vec{x}_{k+1}$ would be to use one of our one-dimensional algorithms from §8.3 on a simpler problem. In particular, consider the function $g_k(t) \equiv f(\vec{x}_k - t\nabla f(\vec{x}_k))$, which restricts f to the line through $\vec{x}_k$ parallel to $\nabla f(\vec{x}_k)$. Thanks to our discussion of the gradient, we know that small t will yield a decrease in f.

The *gradient descent* algorithm iteratively solves these one-dimensional problems to improve our estimate of $\vec{x}_k$:

1. Choose an initial estimate $\vec{x}_0$
2. Iterate until convergence of $\vec{x}_k$:
   (a) Take $g_k(t) \equiv f(\vec{x}_k - t\nabla f(\vec{x}_k))$
   (b) Use a one-dimensional algorithm to find t^* minimizing g_k over all $t \geq 0$ ("line search")
   (c) Take $\vec{x}_{k+1} \equiv \vec{x}_k - t^* \nabla f(\vec{x}_k)$

Each iteration of gradient descent decreases $f(\vec{x}_k)$, so the objective values converge. The algorithm only terminates when $\nabla f(\vec{x}_k) \approx \vec{0}$, showing that gradient descent must at least reach a local minimum; convergence is slow for most functions f, however. The line search process can be replaced by a method that simply decreases the objective a non-negligible if suboptimal amount, although it is more difficult to guarantee convergence in this case.

8.4.2 Newton's Method

Paralleling our derivation of the single-variable case, we can write a Taylor series approximation of $f : \mathbb{R}^n \to \mathbb{R}$ using its Hessian H_f:

$$f(\vec{x}) \approx f(\vec{x}_k) + \nabla f(\vec{x}_k)^\top \cdot (\vec{x} - \vec{x}_k) + \frac{1}{2}(\vec{x} - \vec{x}_k)^\top \cdot H_f(\vec{x}_k) \cdot (\vec{x} - \vec{x}_k)$$

Differentiating with respect to $\vec{x}$ and setting the result equal to zero yields the following iterative scheme:

$$\vec{x}_{k+1} = \vec{x}_k - [H_f(\vec{x}_k)]^{-1}\nabla f(\vec{x}_k)$$

It is easy to double check that this expression is a generalization of that in §8.3.1, and once again it converges quadratically when $\vec{x}_0$ is near a minimum.

Newton's method can be more efficient than gradient descent depending on the optimization objective f. Recall that each iteration of gradient descent potentially requires many evaluations of f during the line search procedure. On the other hand, we must evaluate and invert the Hessian H_f during each iteration of Newton's method. Notice that these factors do not affect the number of iterations but do affect runtime: this is a tradeoff that may not be obvious via traditional analysis.

It is intuitive why Newton's method converges quickly when it is near an optimum. In particular, gradient descent has no knowledge of H_f; it proceeds analogously to walking downhill by looking only at your feet. By using H_f, Newton's method has a larger picture of the shape of f nearby.

When H_f is not positive definite, however, the objective locally might look like a saddle or peak rather than a bowl. In this case, jumping to an approximate stationary point might not make sense. Thus, adaptive techniques might check if H_f is positive definite before applying a Newton step; if it is not positive definite, the methods can revert to gradient descent to find a better approximation of the minimum. Alternatively, they can modify H_f by, e.g., projecting onto the closest positive definite matrix.

8.4.3 Optimization without Derivatives: BFGS

Newton's method can be difficult to apply to complicated functions $f : \mathbb{R}^n \to \mathbb{R}$. The second derivative of f might be considerably more involved than the form of f, and H_f changes with each iteration, making it difficult to reuse work from previous iterations. Additionally, H_f has size $n \times n$, so storing H_f requires $O(n^2)$ space, which can be unacceptable.

As in our discussion of root-finding, techniques for minimization that imitate Newton's method but use approximate derivatives are called *quasi-Newton methods*. Often they can have similarly strong convergence properties without the need for explicit re-evaluation and even factorization of the Hessian at each iteration. In our discussion, we will follow the development of (CITE NOCEDAL AND WRIGHT).

Suppose we wish to minimize $f : \mathbb{R}^n \to \mathbb{R}$ using an iterative scheme. Near the current estimate $\vec{x}_k$ of the root, we might estimate f with a quadratic model:

$$f(\vec{x}_k + \delta\vec{x}) \approx f(\vec{x}_k) + \nabla f(\vec{x}_k) \cdot \delta\vec{x} + \frac{1}{2}(\delta\vec{x})^\top B_k(\delta\vec{x}).$$

Notice that we have asked that our approximation agrees with f to first order at $\vec{x}_k$; as in Broyden's method for root-finding, however, we will allow our estimate of the Hessian B_k to vary.

This quadratic model is minimized by taking $\delta\vec{x} = -B_k^{-1}\nabla f(\vec{x}_k)$. In case $\|\delta\vec{x}\|_2$ is large and we do not wish to take such a considerable step, we will allow ourselves to scale this difference by a step size α_k, yielding

$$\vec{x}_{k+1} = \vec{x}_k - \alpha_k B_k^{-1}\nabla f(\vec{x}_k).$$

Our goal is to find a reasonable estimate B_{k+1} by updating B_k, so that we can repeat this process.

The Hessian of f is nothing more than the derivative of ∇f, so we can write a secant-style condition on B_{k+1}:

$$B_{k+1}(\vec{x}_{k+1} - \vec{x}_k) = \nabla f(\vec{x}_{k+1}) - \nabla f(\vec{x}_k).$$

We will substitute $\vec{s}_k \equiv \vec{x}_{k+1} - \vec{x}_k$ and $\vec{y}_k \equiv \nabla f(\vec{x}_{k+1}) - \nabla f(\vec{x}_k)$, yielding an equivalent condition $B_{k+1}\vec{s}_k = \vec{y}_k$.

Given the optimization at hand, we wish for B_k to have two properties:

- B_k should be a symmetric matrix, like the Hessian H_f.
- B_k should be positive (semi-)definite, so that we are seeking minima rather than maxima or saddle points.

The symmetry condition is enough to eliminate the possibility of using the Broyden estimate we developed in the previous chapter.

The positive definite constraint implicitly puts a condition on the relationship between $\vec{s}_k$ and $\vec{y}_k$. In particular, premultiplying the relationship $B_{k+1}\vec{s}_k = \vec{y}_k$ by $\vec{s}_k^\top$ shows $\vec{s}_k^\top B_{k+1}\vec{s}_k = \vec{s}_k^\top \vec{y}_k$. For B_{k+1} to be positive definite, we must then have $\vec{s}_k \cdot \vec{y}_k > 0$. This observation can guide our choice of α_k; it is easy to see that it holds for sufficiently small $\alpha_k > 0$.

Assume that $\vec{s}_k$ and $\vec{y}_k$ satisfy our compatibility condition. With this in place, we can write down a Broyden-style optimization leading to a possible approximation B_{k+1}:

$$\begin{array}{rl} \text{minimize}_{B_{k+1}} & \|B_{k+1} - B_k\| \\ \text{such that} & B_{k+1}^\top = B_{k+1} \\ & B_{k+1}\vec{s}_k = \vec{y}_k \end{array}$$

For appropriate choice of norms $\|\cdot\|$, this optimization yield the well-known DFP (Davidon-Fletcher-Powell) iterative scheme.

Rather than work out the details of the DFP scheme, we move on to a more popular method known as the BFGS (Broyden-Fletcher-Goldfarb-Shanno) formula, which appears in many modern systems. Notice that—ignoring our choice of α_k for now—our second-order approximation was minimized by taking $\delta\vec{x} = -B_k^{-1}\nabla f(\vec{x}_k)$. Thus, in the end the behavior of our iterative scheme is dictated by the *inverse* matrix B_k^{-1}. Asking that $\|B_{k+1} - B_k\|$ is small can still imply relatively bad differences between the action of B_k^{-1} and that of B_{k+1}^{-1}!

With this observation in mind, the BFGS scheme makes a small alteration to the above derivation. Rather than computing B_k at each iteration, we can compute its inverse $H_k \equiv B_k^{-1}$ directly. Now our condition $B_{k+1}\vec{s}_k = \vec{y}_k$ gets reversed to $\vec{s}_k = H_{k+1}\vec{y}_k$; the condition that B_k is symmetric is the same as asking that H_k is symmetric. We solve an optimization

$$\begin{array}{rl} \text{minimize}_{H_{k+1}} & \|H_{k+1} - H_k\| \\ \text{such that} & H_{k+1}^\top = H_{k+1} \\ & \vec{s}_k = H_{k+1}\vec{y}_k \end{array}$$

This construction has the nice side benefit of not requiring matrix inversion to compute $\delta\vec{x} = -H_k\nabla f(\vec{x}_k)$.

To derive a formula for H_{k+1}, we must decide on a matrix norm $\|\cdot\|$. As with our previous discussion, the *Frobenius* norm looks closest to least-squares optimization, making it likely we can generate a closed-form expression for H_{k+1} rather than having to solve the minimization above as a subroutine of BFGS optimization.

The Frobenius norm, however, has one serious drawback for Hessian matrices. Recall that the Hessian matrix has entries $(H_f)_{ij} = {}^{\partial f_i}/_{\partial x_j}$. Often the quantities x_i for different i can have different *units*; e.g. consider maximizing the profit (in dollars) made by selling a cheeseburger of radius r (in inches) and price p (in dollars), leading to f : (inches, dollars) $\to$ dollars. Squaring these different quantities and adding them up does not make sense.

Suppose we find a symmetric positive definite matrix W so that $W\vec{s}_k = \vec{y}_k$; we will check in the exercises that such a matrix exists. Such a matrix takes the units of $\vec{s}_k = \vec{x}_{k+1} - \vec{x}_k$ to those of $\vec{y}_k = \nabla f(\vec{x}_{k+1}) - \nabla f(\vec{x}_k)$. Taking inspiration from our expression $\|A\|^2_{\text{Fro}} = \text{Tr}(A^\top A)$, we can define a *weighted Frobenius norm* of a matrix A as

$$\|A\|^2_W \equiv \text{Tr}(A^\top W^\top AW)$$

It is straightforward to check that this expression has consistent units when applied to our optimization for H_{k+1}. When both W and A are symmetric with columns $\vec{w}_i$ and $\vec{a}_i$, resp., expanding the expression above shows:

$$\|A\|^2_W = \sum_{ij} (\vec{w}_i \cdot \vec{a}_j)(\vec{w}_j \cdot \vec{a}_i).$$

This choice of norm combined with the choice of W yields a particularly clean formula for H_{k+1} given H_k, $\vec{s}_k$, and $\vec{y}_k$:

$$H_{k+1} = (I_{n\times n} - \rho_k \vec{s}_k \vec{y}_k^\top) H_k (I_{n\times n} - \rho_k \vec{y}_k \vec{s}_k^\top) + \rho_k \vec{s}_k \vec{s}_k^\top,$$

where $\rho_k \equiv {}^1/_{\vec{y}\cdot\vec{s}}$. We show in the Appendix to this chapter how to derive this formula.

The BFGS algorithm avoids the need to compute and invert a Hessian matrix for f, but it still requires $O(n^2)$ storage for H_k. A useful variant known as L-BFGS ("Limited-Memory BFGS") avoids this issue by keeping a limited history of vectors $\vec{y}_k$ and $\vec{s}_k$ and applying H_k by expanding its formula recursively. This approach actually can have *better* numerical properties despite its compact use of space; in particular, old vectors $\vec{y}_k$ and $\vec{s}_k$ may no longer be relevant and *should* be ignored.

8.5 Problems

List of ideas:

- Derive Gauss-Newton
- Stochastic methods, AdaGrad
- VSCG algorithm
- Wolfe conditions for gradient descent; plug into BFGS
- Sherman-Morrison-Woodbury formula for B_k for BFGS
- Prove BFGS converges; show existence of a matrix W
- (Generalized) reduced gradient algorithm
- Condition number for optimization

Appendix: Derivation of BFGS Update[1]

Our optimization for H_{k+1} has the following Lagrange multiplier expression (for ease of notation we take $H_{k+1} \equiv H$ and $H_k = H^*$):

$$\begin{aligned}
\Lambda &\equiv \sum_{ij} (\vec{w}_i \cdot (\vec{h}_j - \vec{h}_j^*))(\vec{w}_j \cdot (\vec{h}_i - \vec{h}_i^*)) - \sum_{i<j} \alpha_{ij}(H_{ij} - H_{ji}) - \vec{\lambda}^\top (H\vec{y}_k - \vec{s}_k) \\
&= \sum_{ij} (\vec{w}_i \cdot (\vec{h}_j - \vec{h}_j^*))(\vec{w}_j \cdot (\vec{h}_i - \vec{h}_i^*)) - \sum_{ij} \alpha_{ij} H_{ij} - \vec{\lambda}^\top (H\vec{y}_k - \vec{s}_k) \text{ if we assume } \alpha_{ij} = -\alpha_{ji}
\end{aligned}$$

Taking derivatives to find critical points shows (for $\vec{y} \equiv \vec{y}_k, \vec{s} \equiv \vec{s}_k$):

$$\begin{aligned}
0 = \frac{\partial \Lambda}{\partial H_{ij}} &= \sum_\ell 2 w_{i\ell} (\vec{w}_j \cdot (\vec{h}_\ell - \vec{h}_\ell^*)) - \alpha_{ij} - \lambda_i y_j \\
&= 2 \sum_\ell w_{i\ell} (W^\top (H - H^*))_{j\ell} - \alpha_{ij} - \lambda_i y_j \\
&= 2 \sum_\ell (W^\top (H - H^*))_{j\ell} w_{\ell i} - \alpha_{ij} - \lambda_i y_j \text{ by symmetry of } W \\
&= 2 (W^\top (H - H^*) W)_{ji} - \alpha_{ij} - \lambda_i y_j \\
&= 2 (W (H - H^*) W)_{ij} - \alpha_{ij} - \lambda_i y_j \text{ by symmetry of } W \text{ and } H
\end{aligned}$$

So, in matrix form we have the following list of facts:

$$\begin{aligned}
0 &= 2W(H - H^*)W - A - \vec{\lambda}\vec{y}^\top, \text{ where } A_{ij} = \alpha_{ij} \\
A^\top &= -A, W^\top = W, H^\top = H, (H^*)^\top = H^* \\
H\vec{y} &= \vec{s}, W\vec{s} = \vec{y}
\end{aligned}$$

We can achieve a pair of relationships using transposition combined with symmetry of H and W and asymmetry of A:

$$\begin{aligned}
0 &= 2W(H - H^*)W - A - \vec{\lambda}\vec{y}^\top \\
0 &= 2W(H - H^*)W + A - \vec{y}\vec{\lambda}^\top \\
\implies 0 &= 4W(H - H^*)W - \vec{\lambda}\vec{y}^\top - \vec{y}\vec{\lambda}^\top
\end{aligned}$$

Post-multiplying this relationship by $\vec{s}$ shows:

$$\vec{0} = 4(\vec{y} - WH^*\vec{y}) - \vec{\lambda}(\vec{y} \cdot \vec{s}) - \vec{y}(\vec{\lambda} \cdot \vec{s})$$

Now, take the dot product with $\vec{s}$:

$$0 = 4(\vec{y} \cdot \vec{s}) - 4(\vec{y}^\top H^* \vec{y}) - 2(\vec{y} \cdot \vec{s})(\vec{\lambda} \cdot \vec{s})$$

This shows:

$$\vec{\lambda} \cdot \vec{s} = 2\rho \vec{y}^\top (\vec{s} - H^*\vec{y}), \text{ for } \rho \equiv {}^1/_{\vec{y} \cdot \vec{s}}$$

[1]Special thanks to Tao Du for debugging several parts of this derivation.

Now, we substitute this into our vector equality:

$$\begin{aligned}
\vec{0} &= 4(\vec{y} - WH^*\vec{y}) - \vec{\lambda}(\vec{y}\cdot\vec{s}) - \vec{y}(\vec{\lambda}\cdot\vec{s}) \text{ from before} \\
&= 4(\vec{y} - WH^*\vec{y}) - \vec{\lambda}(\vec{y}\cdot\vec{s}) - \vec{y}[2\rho\vec{y}^\top(\vec{s} - H^*\vec{y})] \text{ from our simplification} \\
\implies \vec{\lambda} &= 4\rho(\vec{y} - WH^*\vec{y}) - 2\rho^2\vec{y}^\top(\vec{s} - H^*\vec{y})\vec{y}
\end{aligned}$$

Post-multiplying by $\vec{y}^\top$ shows:

$$\vec{\lambda}\vec{y}^\top = 4\rho(\vec{y} - WH^*\vec{y})\vec{y}^\top - 2\rho^2\vec{y}^\top(\vec{s} - H^*\vec{y})\vec{y}\vec{y}^\top$$

Taking the transpose,

$$\vec{y}\vec{\lambda}^\top = 4\rho\vec{y}(\vec{y}^\top - \vec{y}^\top H^*W) - 2\rho^2\vec{y}^\top(\vec{s} - H^*\vec{y})\vec{y}\vec{y}^\top$$

Combining these results and dividing by four shows:

$$\frac{1}{4}(\vec{\lambda}\vec{y}^\top + \vec{y}\vec{\lambda}^\top) = \rho(2\vec{y}\vec{y}^\top - WH^*\vec{y}\vec{y}^\top - \vec{y}\vec{y}^\top H^*W) - \rho^2\vec{y}^\top(\vec{s} - H^*\vec{y})\vec{y}\vec{y}^\top$$

Now, we will pre- and post-multiply by W^{-1}. Since $W\vec{s} = \vec{y}$, we can equivalently write $\vec{s} = W^{-1}\vec{y}$; furthermore, by symmetry of W we then know $\vec{y}^\top W^{-1} = \vec{s}^\top$. Applying these identities to the expression above shows:

$$\begin{aligned}
\frac{1}{4}W^{-1}(\vec{\lambda}\vec{y}^\top + \vec{y}\vec{\lambda}^\top)W^{-1} &= 2\rho\vec{s}\vec{s}^\top - \rho H^*\vec{y}\vec{s}^\top - \rho\vec{s}\vec{y}^\top H^* - \rho^2(\vec{y}^\top\vec{s})\vec{s}\vec{s}^\top + \rho^2(\vec{y}^\top H^*\vec{y})\vec{s}\vec{s}^\top \\
&= 2\rho\vec{s}\vec{s}^\top - \rho H^*\vec{y}\vec{s}^\top - \rho\vec{s}\vec{y}^\top H^* - \rho\vec{s}\vec{s}^\top + \vec{s}\rho^2(\vec{y}^\top H^*\vec{y})\vec{s}^\top \text{ by definition of } \rho \\
&= \rho\vec{s}\vec{s}^\top - \rho H^*\vec{y}\vec{s}^\top - \rho\vec{s}\vec{y}^\top H^* + \vec{s}\rho^2(\vec{y}^\top H^*\vec{y})\vec{s}^\top
\end{aligned}$$

Finally, we can conclude our derivation of the BFGS step as follows:

$$\begin{aligned}
0 &= 4W(H - H^*)W - \vec{\lambda}\vec{y}^\top - \vec{y}\vec{\lambda}^\top \text{ from before} \\
\implies H &= \frac{1}{4}W^{-1}(\vec{\lambda}\vec{y}^\top + \vec{y}\vec{\lambda}^\top)W^{-1} + H^* \\
&= \rho\vec{s}\vec{s}^\top - \rho H^*\vec{y}\vec{s}^\top - \rho\vec{s}\vec{y}^\top H^* + \vec{s}\rho^2(\vec{y}^\top H^*\vec{y})\vec{s}^\top + H^* \text{ from the last paragraph} \\
&= H^*(I - \rho\vec{y}\vec{s}^\top) + \rho\vec{s}\vec{s}^\top - \rho\vec{s}\vec{y}^\top H^* + (\rho\vec{s}\vec{y}^\top)H^*(\rho\vec{y}\vec{s}^\top) \\
&= H^*(I - \rho\vec{y}\vec{s}^\top) + \rho\vec{s}\vec{s}^\top - \rho\vec{s}\vec{y}^\top H^*(I - \rho\vec{y}\vec{s}^\top) \\
&= \rho\vec{s}\vec{s}^\top + (I - \rho\vec{s}\vec{y}^\top)H^*(I - \rho\vec{y}\vec{s}^\top)
\end{aligned}$$

This final expression is exactly the BFGS step introduced in the chapter.

Chapter 9

Constrained Optimization

We continue our consideration of optimization problems by studying the *constrained* case. These problems take the following general form:

$$\begin{aligned} \text{minimize } & f(\vec{x}) \\ \text{such that } & g(\vec{x}) = \vec{0} \\ & h(\vec{x}) \geq \vec{0} \end{aligned}$$

Here, $f : \mathbb{R}^n \to \mathbb{R}$, $g : \mathbb{R}^n \to \mathbb{R}^m$, and $h : \mathbb{R}^n \to \mathbb{R}^p$. Obviously this form is extremely generic, so it is not difficult to predict that algorithms for solving such problems in the absence of additional assumptions on f, g, or h can be difficult to formulate and are subject to degeneracies such as local minima and lack of convergence. In fact, this optimization encodes other problems we already have considered; if we take $f(\vec{x}) \equiv 0$, then this constrained optimization becomes root-finding on g, while if we take $g(\vec{x}) = h(\vec{x}) \equiv \vec{0}$ then it reduces to unconstrained optimization on f.

Despite this somewhat bleak outlook, optimizations for general constrained case can be valuable when f, g, and h do not have useful structure or are too specialized to merit specialized treatment. Furthermore, when f is heuristic anyway, simply finding a feasible $\vec{x}$ for which $f(\vec{x}) < f(\vec{x}_0)$ for an initial guess $\vec{x}_0$ is valuable. One simple application in this domain would be an economic system in which f measures costs; obviously we wish to minimize costs, but if $\vec{x}_0$ represents the current configuration, *any* $\vec{x}$ decreasing f is a valuable output.

9.1 Motivation

It is not difficult to encounter constrained optimization problems in practice. In fact, we already listed many applications of these problems when we discussed eigenvectors and eigenvalues, since this problem can be posed as finding critical points of $\vec{x}^\top A\vec{x}$ subject to $\|\vec{x}\|_2 = 1$; of course, the particular case of eigenvalue computation admits special algorithms that make it a simpler problem.

Here we list other optimizations that do not enjoy the structure of eigenvalue problems:

Example 9.1 (Geometric projection)**.** *Many surfaces S in $\mathbb{R}^3$ can be written* implicitly *in the form $g(\vec{x}) = 0$ for some g. For example, the unit sphere results from taking $g(\vec{x}) \equiv \|\vec{x}\|_2^2 - 1$, while a cube can* *be constructed by taking $g(\vec{x}) = \|\vec{x}\|_1 - 1$. In fact, some 3D modeling environments allow users to specify "blobby" objects, as in Figure NUMBER, as sums*

$$g(\vec{x}) \equiv c + \sum_i a_i e^{-b_i \|\vec{x}-\vec{x}_i\|_2^2}.$$

Suppose we are given a point $\vec{y} \in \mathbb{R}^3$ and wish to find the closest point on S to $\vec{y}$. This problem is solved by using the following constrained minimization:

$$\begin{aligned} \text{minimize}_{\vec{x}} \ & \|\vec{x} - \vec{y}\|_2 \\ \text{such that } & g(\vec{x}) = 0 \end{aligned}$$

Example 9.2 (Manufacturing). *Suppose you have m different materials; you have s_i units of each material i in stock. You can manufacture k different products; product j gives you profit p_j and uses c_{ij} of material i to make. To maximize profits, you can solve the following optimization for the total amount x_j you should manufacture of each item j:*

$$\begin{aligned} \text{maximize}_{\vec{x}} \ & \sum_{j=1}^k p_j x_j \\ \text{such that } & x_j \geq 0 \, \forall j \in \{1, \ldots, k\} \\ & \sum_{j=1}^k c_{ij} x_j \leq s_i \, \forall i \in \{1, \ldots, m\} \end{aligned}$$

The first constraint ensures that you do not make negative numbers of any product, and the second ensures that you do not use more than your stock of each material.

Example 9.3 (Nonnegative least-squares). *We already have seen numerous examples of least-squares problems, but sometimes negative values in the solution vector might not make sense. For example, in computer graphics, an animated model might be expressed as a deforming bone structure plus a meshed "skin;" for each point on the skin a list of weights can be computed to approximate the influence of the positions of the bone joints on the position of the skin vertices (CITE). Such weights should be constrained to be nonnegative to avoid degenerate behavior while the surface deforms. In such a case, we can solve the "nonnegative least-squares" problem:*

$$\begin{aligned} \text{minimize}_{\vec{x}} \ & \|A\vec{x} - \vec{b}\|_2 \\ \text{such that } & x_i \geq 0 \, \forall i \end{aligned}$$

Recent research involves characterizing the sparsity *of nonnegative least squares solutions, which often have several values x_i satisfying $x_i = 0$ exactly (CITE).*

Example 9.4 (Bundle adjustment). *In computer vision, suppose we take a picture of an object from several angles. A natural task is to reconstruct the three-dimensional shape of the object. To do so, we might mark a corresponding set of points on each image; in particular, we can take $\vec{x}_{ij} \in \mathbb{R}^2$ to be the position of feature point j on image i. In reality, each feature point has a position $\vec{y}_j \in \mathbb{R}^3$ in space, which we would like to compute. Additionally, we must find the positions of the cameras themselves, which we can represent as* *unknown projection matrices P_i. This problem, known as* bundle adjustment, *can be approached using an optimization strategy:*

$$\begin{gathered}\text{minimize}_{\vec{y}_j, P_i} \sum_{ij} \|P_i\vec{y}_j - \vec{x}_{ij}\|_2^2 \\ \text{such that} P_i \text{ is orthogonal } \forall i\end{gathered}$$

The orthogonality constraint ensures that the camera transformations are reasonable.

9.2 Theory of Constrained Optimization

In our discussion, we will assume that f, g, and h are differentiable. Some methods exist that only make weak continuity or Lipschitz assumptions, but these techniques are quite specialized and require advanced analytical consideration.

Although we have not yet developed algorithms for general constrained optimization, we implicitly have made use of the *theory* of such problems in considering eigenvalue methods. Specifically, recall the method of Lagrange multipliers, introduced in Theorem 0.1. In this technique, critical points $f(\vec{x})$ subject to $g(\vec{x})$ are characterized as critical points of the unconstrained Lagrange multiplier function $\Lambda(\vec{x}, \vec{\lambda}) \equiv f(\vec{x}) - \vec{\lambda} \cdot \vec{g}(\vec{x})$ with respect to both $\vec{\lambda}$ and $\vec{x}$ simultaneously. This theorem allowed us to provide variational interpretations of eigenvalue problems; more generally, it gives an alternative (necessary but not sufficient) criterion for $\vec{x}$ to be a critical point of an *equality-constrained* optimization.

Simply finding an $\vec{x}$ satisfying the constraints, however, can be a considerable challenge. We can separate these issues by making a few definitions:

Definition 9.1 (Feasible point and feasible set)**.** *A* feasible point *of a constrained optimization problem is any point $\vec{x}$ satisfying $g(\vec{x}) = \vec{0}$ and $h(\vec{x}) \geq \vec{0}$. The* feasible set *is the set of all points $\vec{x}$ satisfying these constraints.*

Definition 9.2 (Critical point of constrained optimization)**.** *A critical point of a constrained optimization is one satisfying the constraints that also is a local maximum, minimum, or saddle point of f within the feasible set.*

Constrained optimizations are difficult because they simultaneously solve root-finding problems (the $g(\vec{x}) = \vec{0}$ constraint), satisfiability problems (the $h(\vec{x}) \geq \vec{0}$ constraint), and minimization (the function f). This aside, to push our differential techniques to complete generality, we must find a way to add inequality constraints to the Lagrange multiplier system. Suppose we have found the minimum of the optimization, denoted $\vec{x}^*$. For each inequality constraint $h_i(\vec{x}^*) \geq 0$, we have two options:

- $h_i(\vec{x}^*) = 0$: Such a constraint is *active*, likely indicating that if the constraint were removed the optimum might change.
- $h_i(\vec{x}^*) > 0$: Such a constraint is *inactive*, meaning in a neighborhood of $\vec{x}^*$ if we had removed this constraint we still would have reached the same minimum.

Of course, we do not know which constraints will be active or inactive at $\vec{x}^*$ until it is computed.

If all of our constraints were active, then we could change our $h(\vec{x}) \geq \vec{0}$ constraint to an equality without affecting the minimum. This might motivate studying the following Lagrange multiplier system:

$$\Lambda(\vec{x}, \vec{\lambda}, \vec{\mu}) \equiv f(\vec{x}) - \vec{\lambda} \cdot g(\vec{x}) - \vec{\mu} \cdot h(x)$$

We no longer can say that $\vec{x}^*$ is a critical point of Λ, however, because inactive constraints would remove terms above. Ignoring this (important!) issue for the time being, we could proceed blindly and ask for critical points of this new Λ with respect to $\vec{x}$, which satisfy the following:

$$\vec{0} = \nabla f(\vec{x}) - \sum_i \lambda_i \nabla g_i(\vec{x}) - \sum_j \mu_j \nabla h_j(\vec{x})$$

Here we have separated out the individual components of g and h and treated them as scalar functions to avoid complex notation.

A clever trick can extend this optimality condition to inequality-constrained systems. Notice that if we had taken $\mu_j = 0$ whenever h_j is inactive, then this removes the irrelevant terms from the optimality conditions. In other words, we can *add* a constraint on the Lagrange multipliers:

$$\mu_j h_j(\vec{x}) = 0.$$

With this constraint in place, we know that at least one of μ_j and $h_j(\vec{x})$ must be zero, and therefore our first-order optimality condition still holds!

So far, our construction has not distinguished between the constraint $h_j(\vec{x}) \geq 0$ and the constraint $h_j(\vec{x}) \leq 0$. If the constraint is inactive, it could have been dropped without affecting the outcome of the optimization locally, so we consider the case when the constraint is active. Intuitively,[1] in this case we expect there to be a way to decrease f by violating the constraint. Locally, the direction in which f decreases is $-\nabla f(\vec{x}^*)$ and the direction in which h_j decreases is $-\nabla h_j(\vec{x}^*)$. Thus, starting at $\vec{x}^*$ we can decrease f even more by violating the constraint $h_j(\vec{x}) \geq 0$ when $\nabla f(\vec{x}^*) \cdot \nabla h_j(\vec{x}^*) \geq 0$.

Of course, products of gradients of f and h_j are difficult to work with. However, recall that at $\vec{x}^*$ our first-order optimality condition tells us:

$$\nabla f(\vec{x}^*) = \sum_i \lambda_i^* \nabla g_i(\vec{x}^*) + \sum_{j \text{ active}} \mu_j^* \nabla h_j(\vec{x}^*)$$

The inactive μ_j values are zero and can be removed. In fact, we can remove the $g(\vec{x}) = 0$ constraints by adding inequality constraints $g(\vec{x}) \geq \vec{0}$ and $g(\vec{x}) \leq \vec{0}$ to h; this is a mathematical convenience for writing a proof rather than a numerically-wise maneuver. Then, taking dot products with ∇h_k for any fixed k shows:

$$\sum_{j \text{ active}} \mu_j^* \nabla h_j(\vec{x}^*) \cdot \nabla h_k(\vec{x}^*) = \nabla f(\vec{x}^*) \cdot \nabla h_k(\vec{x}^*) \geq 0$$

Vectorizing this expression shows $Dh(\vec{x}^*) Dh(\vec{x}^*)^\top \vec{\mu}^* \geq \vec{0}$. Since $Dh(\vec{x}^*) Dh(x^*)^\top$ is positive semidefinite, this implies $\vec{\mu}^* \geq \vec{0}$. Thus, the $\nabla f(\vec{x}^*) \cdot \nabla h_j(\vec{x}^*) \geq 0$ observation manifests itself simply by the fact that $\mu_j \geq 0$.

Our observations can be formalized to prove a first-order optimality condition for inequality-constrained optimizations:

[1]You should not consider our discussion a formal proof, since we are not considering many boundary cases.

Theorem 9.1 (Karush-Kuhn-Tucker (KKT) conditions). *The vector $\vec{x}^* \in \mathbb{R}^n$ is a critical point for minimizing f subject to $g(\vec{x}) = \vec{0}$ and $h(\vec{x}) \geq \vec{0}$ when there exists $\vec{\lambda} \in \mathbb{R}^m$ and $\vec{\mu} \in \mathbb{R}^p$ such that:*

- $\vec{0} = \nabla f(\vec{x}^*) - \sum_i \lambda_i \nabla g_i(\vec{x}^*) - \sum_j \mu_j \nabla h_j(\vec{x}^*)$ *("stationarity")*
- $g(\vec{x}^*) = \vec{0}$ *and* $h(\vec{x}^*) \geq \vec{0}$ *("primal feasibility")*
- $\mu_j h_j(\vec{x}^*) = 0$ *for all j ("complementary slackness")*
- $\mu_j \geq 0$ *for all j ("dual feasibility")*

Notice that when h is removed this theorem reduces to the Lagrange multiplier criterion.

Example 9.5 (Simple optimization[2]). *Suppose we wish to solve*

$$\begin{aligned} &\textit{maximize } xy \\ &\textit{such that } x + y^2 \leq 2 \\ &x, y \geq 0 \end{aligned}$$

In this case we will have no λ's and three μ's. We take $f(x,y) = -xy$, $h_1(x,y) \equiv 2 - x - y^2$, $h_2(x,y) = x$, and $h_3(x,y) = y$. The KKT conditions are:

$$\begin{aligned} \textit{Stationarity: } & 0 = -y + \mu_1 - \mu_2 \\ & 0 = -x + 2\mu_1 y - \mu_3 \\ \textit{Primal feasibility: } & x + y^2 \leq 2 \\ & x, y \geq 0 \\ \textit{Complementary slackness: } & \mu_1(2 - x - y^2) = 0 \\ & \mu_2 x = 0 \\ & \mu_3 y = 0 \\ \textit{Dual feasibility: } & \mu_1, \mu_2, \mu_3 \geq 0 \end{aligned}$$

Example 9.6 (Linear programming). *Consider the optimization:*

$$\begin{aligned} &\textit{minimize}_{\vec{x}} \; \vec{b} \cdot \vec{x} \\ &\textit{such that } A\vec{x} \geq \vec{c} \end{aligned}$$

Notice Example 9.2 can be written this way. The KKT conditions for this problem are:

$$\begin{aligned} \textit{Stationarity: } & A^\top \vec{\mu} = \vec{b} \\ \textit{Primal feasibility: } & A\vec{x} \geq \vec{c} \\ \textit{Complementary slackness: } & \mu_i(\vec{a}_i \cdot \vec{x} - c_i) = 0 \; \forall i, \textit{ where } \vec{a}_i^\top \textit{ is row } i \textit{ of } A \\ \textit{Dual feasibility: } & \vec{\mu} \geq \vec{0} \end{aligned}$$

As with the Lagrange multipliers case, we cannot assume that any $\vec{x}^*$ satisfying the KKT conditions automatically minimizes f subject to the constraints, even locally. One way to check for local optimality is to examine the Hessian of f restricted to the subspace of $\mathbb{R}^n$ in which $\vec{x}$ can move without violating the constraints; if this "reduced" Hessian is positive definite then the optimization has reached a local minimum.

[2]From `http://www.math.ubc.ca/~israel/m340/kkt2.pdf`

9.3 Optimization Algorithms

A careful consideration of algorithms for constrained optimization is out of the scope of our discussion; thankfully many stable implementations exist of these techniques and much can be accomplished as a "client" of this software rather than rewriting it from scratch. Even so, it is useful to sketch some potential approaches to gain some intuition for how these libraries function.

9.3.1 Sequential Quadratic Programming (SQP)

Similar to BFGS and other methods we considered in our discussion of unconstrained optimization, one typical strategy for constrained optimization is to approximate f, g, and h with simpler functions, solve the approximated optimization, and iterate.

Suppose we have a guess $\vec{x}_k$ of the solution to the constrained optimization problem. We could apply a second-order Taylor expansion to f and first-order approximation to g and h to define a next iterate as the following:

$$\begin{aligned}\vec{x}_{k+1} \equiv \vec{x}_k + \arg\min_{\vec{d}} \; & \frac{1}{2}\vec{d}^\top H_f(\vec{x}_k)\vec{d} + \nabla f(\vec{x}_k)\cdot\vec{d} + f(\vec{x}_k) \\ \text{such that } & g_i(\vec{x}_k) + \nabla g_i(\vec{x}_k)\cdot\vec{d} = 0 \\ & h_i(\vec{x}_k) + \nabla h_i(\vec{x}_k)\cdot\vec{d} \geq 0\end{aligned}$$

The optimization to find $\vec{d}$ has a quadratic objective with linear constraints, for which optimization can be considerably easier using one of many strategies. It is known as a *quadratic program*.

Of course, this Taylor approximation only works in a neighborhood of the optimal point. When a good initial guess $\vec{x}_0$ is unavailable, these strategies likely will fail.

Equality constraints When the only constraints are equalities and h is removed, the quadratic program for $\vec{d}$ has Lagrange multiplier optimality conditions derived as follows:

$$\begin{aligned}\Lambda(\vec{d},\vec{\lambda}) &\equiv \frac{1}{2}\vec{d}^\top H_f(\vec{x}_k)\vec{d} + \nabla f(\vec{x}_k)\cdot\vec{d} + f(\vec{x}_k) + \vec{\lambda}^\top(g(\vec{x}_k) + Dg(\vec{x}_k)\vec{d}) \\ \implies \vec{0} &= \nabla_{\vec{d}}\Lambda = H_f(\vec{x}_k)\vec{d} + \nabla f(\vec{x}_k) + [Dg(\vec{x}_k)]^\top\vec{\lambda}\end{aligned}$$

Combining this with the equality condition yields a linear system:

$$\begin{pmatrix} H_f(\vec{x}_k) & [Dg(\vec{x}_k)]^\top \\ Dg(\vec{x}_k) & 0 \end{pmatrix}\begin{pmatrix} \vec{d} \\ \vec{\lambda} \end{pmatrix} = \begin{pmatrix} -\nabla f(\vec{x}_k) \\ -g(\vec{x}_k) \end{pmatrix}$$

Thus, each iteration of sequential quadratic programming in the presence of only equality constraints can be accomplished by solving this linear system at each iteration to get $\vec{x}_{k+1} \equiv \vec{x}_k + \vec{d}$. It is important to note that the linear system above is *not* positive definite, so on a large scale it can be difficult to solve.

Extensions of this strategy operate as BFGS and similar approximations work for unconstrained optimization, by introducing approximations of the Hessian H_f. Stability also can be introduced by limiting the distance traveled in a single iteration.

Inequality Constraints Specialized algorithms exist for solving quadratic programs rather than general nonlinear programs, and these can be used to generate steps of SQP. One notable strategy is to keep an "active set" of constraints that are active at the minimum with respect to $\vec{d}$; then the equality-constrained methods above can be applied by ignoring inactive constraints. Iterations of active-set optimization update the active set of constraints by adding violated constraints to the active set and removing those inequality constraints h_j for which $\nabla f \cdot \nabla h_j \leq 0$ as in our discussion of the KKT conditions.

9.3.2 Barrier Methods

Another option for minimizing in the presence of constraints is to change the constraints to energy terms. For example, in the equality constrained case we could minimize an "augmented" objective as follows:

$$f_\rho(\vec{x}) = f(\vec{x}) + \rho\|g(\vec{x})\|_2^2$$

Notice that taking $\rho \to \infty$ will force $g(\vec{x})$ to be as small as possible, so eventually we will reach $g(\vec{x}) \approx \vec{0}$. Thus, the *barrier method* of constrained optimization applies iterative *unconstrained* optimization techniques to f_ρ and checks how well the constraints are satisfied; if they are not within a given tolerance, ρ is increased and the optimization continues using the previous iterate as a starting point.

Barrier methods are simple to implement and use, but they can exhibit some pernicious failure modes. In particular, as ρ increases, the influence of f on the objective function diminishes and the Hessian of f_ρ becomes more and more poorly conditioned.

Barrier methods can be applied to inequality constraints as well. Here we must ensure that $h_i(\vec{x}) \geq 0$ for all i; typical choices of barrier functions might include $1/h_i(\vec{x})$ (the "inverse barrier") or $-\log h_i(\vec{x})$ (the "logarithmic barrier").

9.4 Convex Programming

Generally speaking, methods like the ones we have described for constrained optimization come with few if any guarantees on the quality of the output. Certainly these methods are unable to obtain global minima without a good initial guess $\vec{x}_0$, and in certain cases, e.g. when the Hessian near $\vec{x}^*$ is not positive definite, they may not converge at all.

There is one notable exception to this rule, which appears in any number of important optimizations: convex programming. The idea here is that when f is a convex function and the feasible set itself is convex, then the optimization possesses a unique minimum. We already have defined a convex function, but need to understand what it means for a set of constraints to be convex:

Definition 9.3 (Convex set)**.** *A set $S \subseteq \mathbb{R}^n$ is* convex *if for any $\vec{x}, \vec{y} \in S$, the point $t\vec{x} + (1-t)\vec{y}$ is also in S for any $t \in [0,1]$.*

As shown in Figure NUMBER, intuitively a set is convex if its boundary shape cannot bend both inward and outward.

Example 9.7 (Circles)**.** *The disc $\{\vec{x} \in \mathbb{R}^n : \|\vec{x}\|_2 \leq 1\}$ is convex, while the unit circle $\{\vec{x} \in \mathbb{R}^n : \|\vec{x}\|_2 = 1\}$ is not.*

It is easy to see that a convex function has a unique minimum even when that function is restricted to a convex domain. In particular, if the function had two local minima, then the line of points between those minima must yield values of f not greater than those on the endpoints.

Strong convergence guarantees are available for convex optimizations that guarantee finding the global minimum so long as f is convex and the constraints on g and h make a convex feasible set. Thus, a valuable exercise for nearly any optimization problem is to check if it is convex, since such an observation can increase confidence in the solution quality and the chances of success by a large factor.

A new field called *disciplined convex programming* attempts to chain together simple rules about convexity to generate convex optimizations (CITE CVX), allowing the end user to combine simple convex energy terms and constraints so long as they satisfy criteria making the final optimization convex. Useful statements about convexity in this domain include the following:

- The intersection of convex sets is convex; thus, adding multiple convex constraints is an allowable operation.
- The sum of convex functions is convex.
- If f and g are convex, so is $h(\vec{x}) \equiv \max\{f(\vec{x}), g(\vec{x})\}$.
- If f is a convex function, the set $\{\vec{x} : f(\vec{x}) \leq c\}$ is convex.

Tools such as the `CVX` library help separate implementation of assorted convex objectives from their minimization.

Example 9.8 (Convex programming)**.**

- *The nonnegative least squares problem in Example 9.3 is convex because $\|A\vec{x} - \vec{b}\|_2$ is a convex function of $\vec{x}$ and the set $\vec{x} \geq \vec{0}$ is convex.*
- *The linear programming problem in Example 9.6 is convex because it has a linear objective and linear constraints.*
- *We can include $\|\vec{x}\|_1$ in a convex optimization objective by introducing a variable $\vec{y}$. To do so, we add constraints $y_i \geq x_i$ and $y_i \geq -x_i$ for each i and an objective $\sum_i y_i$. This sum has terms that are at least as large as $|x_i|$ and that the energy and constraints are convex. At the minimum we must have $y_i = |x_i|$ since we have constrained $y_i \geq |x_i|$ and we wish to minimize the energy. "Disciplined" convex libraries can do such operations behind the scenes without revealing such substitutions to the end user.*

A particularly important example of a convex optimization is *linear programing* from Example 9.6. The famous *simplex algorithm* keeps track of the active constraints, solves for the resulting $\vec{x}^*$ using a linear system, and checks if the active set must be updated; no Taylor approximations are needed because the objective and feasible set are given by linear machinery. *Interior point* linear programming strategies such as the barrier method also are successful for these problems. For this reason, linear programs can be solved on a *huge* scale—up to millions or billions of variables!—and often appear in problems like scheduling or pricing.

9.5 Problems

- Derive simplex?
- Linear programming duality

Chapter 10

Iterative Linear Solvers

In the previous two chapters, we developed strategies for solving a new class of problems involving minimizing a function $f(\vec{x})$ with or without constraints on $\vec{x}$. In doing so, we relaxed our viewpoint from numerical linear algebra and in particular Gaussian elimination that we must find an *exact* solution to a system of equations and instead turned to iterative schemes that are guaranteed to approximate the minimum of a function better and better as they iterate more and more. Even if we never find the minimum exactly, we know that eventually we will find an $\vec{x}_0$ with $f(\vec{x}_0) \approx \vec{0}$ with arbitrary levels quality, depending on the number of iterations we run.

We now have a reprise of our favorite problem from numerical linear algebra, solving $A\vec{x} = \vec{b}$ for $\vec{x}$, but apply an *iterative* approach rather than expecting to find a solution in closed form. This strategy reveals a new class of linear system solvers that can find reliable approximations of $\vec{x}$ in amazingly few iterations. We already have suggested how to approach this in our discussion of linear algebra, by suggesting that solutions to linear systems are minima of the energy $\|A\vec{x} - \vec{b}\|_2^2$, among others.

Why bother deriving yet another class of linear system solvers? So far, most of our direct approaches require us to represent A as a full $n \times n$ matrix, and algorithms such as LU, QR, or Cholesky factorization all take around $O(n^3)$ time. There are two cases to keep in mind for potential reasons to try iterative schemes:

1. When A is sparse, methods like Gaussian elimination tend to induce *fill*, meaning that even if A contains $O(n)$ nonzero values, intermediate steps of elimination may introduce $O(n^2)$ nonzero values. This property rapidly can cause linear algebra systems to run out of memory. Contrastingly, the algorithms in this chapter require only that you can *apply A* to vectors, which can be done in time proportional to the number of nonzero values in a matrix.

2. We may wish to defeat the $O(n^3)$ runtime of standard matrix factorization techniques. In particular, if an iterative scheme can uncover a fairly accurate solution to $A\vec{x} = \vec{b}$ in a few iterations, runtimes can be decreased considerably.

Also, notice that many of the nonlinear optimization methods we have discussed, in particular those depending on a Newton-like step, require solving a linear system in each iteration! Thus, formulating the fastest possible solver can make a considerable difference when implementing large-scale optimization methods that require one or more linear solves per iteration. In fact, in this case an inaccurate but fast solve to a linear system might be acceptable, since it feeds into a larger iterative technique anyway.

Please note that much of our discussion is due to CITE, although our development can be somewhat shorter given the development in previous chapters.

10.1 Gradient Descent

We will focus our discussion on solving $A\vec{x} = \vec{b}$ where A has three properties:

1. $A \in \mathbb{R}^{n\times n}$ is square
2. A is symmetric, that is, $A^\top = A$
3. A is positive definite, that is, for all $\vec{x} \neq \vec{0}$, $\vec{x}^\top A\vec{x} > 0$

Toward the end of this chapter we will relax these assumptions. For now, notice that we can replace $A\vec{x} = \vec{b}$ with the normal equations $A^\top A\vec{x} = A^\top\vec{b}$ to satisfy these criteria, although as we have discussed this substitution can create numerical conditioning issues.

10.1.1 Deriving the Iterative Scheme

In this case, it is easy to check that solutions of $A\vec{x} = \vec{b}$ are minima of the function $f(\vec{x})$ given by the *quadratic form*

$$f(\vec{x}) \equiv \frac{1}{2}\vec{x}^\top A\vec{x} - \vec{b}^\top\vec{x} + c$$

for any $c \in \mathbb{R}$. In particular, taking the derivative of f shows

$$\nabla f(\vec{x}) = A\vec{x} - \vec{b},$$

and setting $\nabla f(\vec{x}) = \vec{0}$ yields the desired result.

Rather than solving $\nabla f(\vec{x}) = \vec{0}$ directly as we have done in the past, suppose we apply the gradient descent strategy to this minimization. Recall the basic gradient descent algorithm:

1. Compute the search direction $\vec{d}_k \equiv -\nabla f(\vec{x}_{k-1}) = \vec{b} - A\vec{x}_{k-1}$.
2. Define $\vec{x}_k \equiv \vec{x}_{k-1} + \alpha_k\vec{d}_k$, where α_k is chosen such that $f(\vec{x}_k) < f(\vec{x}_{k-1})$

For a generic function f, deciding on the value of α_k can be a difficult one-dimensional "line search" problem, boiling down to minimizing $f(\vec{x}_{k-1} + \alpha_k\vec{d}_k)$ as a function of a single variable $\alpha_k \geq 0$. For our particular choice of the quadratic form $f(\vec{x}) = \frac{1}{2}\vec{x}^\top A\vec{x} - \vec{b}^\top\vec{x} + c$, however, we can do line search in closed form. In particular, define

$$\begin{aligned}
g(\alpha) &\equiv f(\vec{x} + \alpha\vec{d}) \\
&= \frac{1}{2}(\vec{x} + \alpha\vec{d})^\top A(\vec{x} + \alpha\vec{d}) - \vec{b}^\top(\vec{x} + \alpha\vec{d}) + c \\
&= \frac{1}{2}(\vec{x}^\top A\vec{x} + 2\alpha\vec{x}^\top A\vec{d} + \alpha^2\vec{d}^\top A\vec{d}) - \vec{b}^\top\vec{x} - \alpha\vec{b}^\top\vec{d} + c \text{ by symmetry of } A \\
&= \frac{1}{2}\alpha^2\vec{d}^\top A\vec{d} + \alpha(\vec{x}^\top A\vec{d} - \vec{b}^\top\vec{d}) + \text{const.} \\
\implies \frac{dg}{d\alpha}(\alpha) &= \alpha\vec{d}^\top A\vec{d} + \vec{d}^\top(A\vec{x} - \vec{b})
\end{aligned}$$

Thus, if we wish to minimize g with respect to α, we simply choose

$$\alpha = \frac{\vec{d}^\top(\vec{b} - A\vec{x})}{\vec{d}^\top A\vec{d}}$$

In particular, for gradient descent we chose $\vec{d}_k = \vec{b} - A\vec{x}_k$, so in fact α_k takes a nice form:

$$\alpha_k = \frac{\vec{d}_k^\top \vec{d}_k}{\vec{d}_k^\top A\vec{d}_k}$$

In the end, our formula for line search yields the following iterative gradient descent scheme for solving $A\vec{x} = \vec{b}$ in the symmetric positive definite case:

$$\begin{aligned}
\vec{d}_k &= \vec{b} - A\vec{x}_{k-1}\\
\alpha_k &= \frac{\vec{d}_k^\top \vec{d}_k}{\vec{d}_k^\top A\vec{d}_k}\\
\vec{x}_k &= \vec{x}_{k-1} + \alpha_k\vec{d}_k
\end{aligned}$$

10.1.2 Convergence

By construction our strategy for gradient descent decreases $f(\vec{x}_k)$ as $k \to \infty$. Even so, we have not shown that the algorithm reaches the minimal possible value of f, and we have not been able to characterize how many iterations we should run to reach a reasonable level of confidence that $A\vec{x}_k \approx \vec{b}$.

One simple strategy for understanding the convergence of the gradient descent algorithm for our choice of f is to examine the change in backward error from iteration to iteration.[1] Suppose $\vec{x}^*$ is the solution that we are seeking, that is, $A\vec{x}^* = \vec{b}$. Then, we can study the ratio of backward error from iteration to iteration:

$$R_k \equiv \frac{f(\vec{x}_k) - f(\vec{x}^*)}{f(\vec{x}_{k-1}) - f(\vec{x}^*)}$$

Obviously bounding $R_k < \beta < 1$ for some β shows that gradient descent converges.

For convenience, we can expand $f(\vec{x}_k)$:

$$\begin{aligned}
f(\vec{x}_k) &= f(\vec{x}_{k-1} + \alpha_k\vec{d}_k) \text{ by our iterative scheme}\\
&= \frac{1}{2}(\vec{x}_{k-1} + \alpha_k\vec{d}_k)^\top A(\vec{x}_{k-1} + \alpha_k\vec{d}_k) - \vec{b}^\top(\vec{x}_{k-1} + \alpha_k\vec{d}_k) + c\\
&= f(\vec{x}_{k-1}) + \alpha_k\vec{d}_k^\top A\vec{x}_{k-1} + \frac{1}{2}\alpha_k^2\vec{d}_k^\top A\vec{d}_k - \alpha_k\vec{b}^\top\vec{d}_k \text{ by definition of } f\\
&= f(\vec{x}_{k-1}) + \alpha_k\vec{d}_k^\top(\vec{b} - \vec{d}_k) + \frac{1}{2}\alpha_k^2\vec{d}_k^\top A\vec{d}_k - \alpha_k\vec{b}^\top\vec{d}_k \text{ since } \vec{d}_k = \vec{b} - A\vec{x}_{k-1}\\
&= f(\vec{x}_{k-1}) - \alpha_k\vec{d}_k^\top\vec{d}_k + \frac{1}{2}\alpha_k^2\vec{d}_k^\top A\vec{d}_k
\end{aligned}$$

[1]This argument is presented e.g. in `http://www-personal.umich.edu/~mepelman/teaching/IOE511/Handouts/511notes07-7.pdf`.

$$\begin{aligned}
&= f(\vec{x}_{k-1}) - \frac{\vec{d}_k^\top \vec{d}_k}{\vec{d}_k^\top A \vec{d}_k} \vec{d}_k^\top \vec{d}_k + \frac{1}{2}\left(\frac{\vec{d}_k^\top \vec{d}_k}{\vec{d}_k^\top A \vec{d}_k}\right)^2 \vec{d}_k^\top A \vec{d}_k \text{ by definition of } \alpha_k \\
&= f(\vec{x}_{k-1}) - \frac{(\vec{d}_k^\top \vec{d}_k)^2}{\vec{d}_k^\top A \vec{d}_k} + \frac{1}{2}\frac{(\vec{d}_k^\top \vec{d}_k)^2}{\vec{d}_k^\top A \vec{d}_k} = f(\vec{x}_{k-1}) - \frac{(\vec{d}_k^\top \vec{d}_k)^2}{2\vec{d}_k^\top A \vec{d}_k}
\end{aligned}$$

Thus, we can return to our fraction:

$$\begin{aligned}
R_k &= \frac{f(\vec{x}_{k-1}) - \frac{(\vec{d}_k^\top \vec{d}_k)^2}{2\vec{d}_k^\top A \vec{d}_k} - f(\vec{x}^*)}{f(\vec{x}_{k-1}) - f(\vec{x}^*)} \text{ by our formula for } f(\vec{x}_k) \\
&= 1 - \frac{(\vec{d}_k^\top \vec{d}_k)^2}{2\vec{d}_k^\top A \vec{d}_k (f(\vec{x}_{k-1}) - f(\vec{x}^*))}
\end{aligned}$$

Notice that $A\vec{x}^* = \vec{b}$, so we can write:

$$\begin{aligned}
f(\vec{x}_{k-1}) - f(\vec{x}^*) &= \left[\frac{1}{2}\vec{x}_{k-1}^\top A \vec{x}_{k-1} - \vec{b}^\top \vec{x}_{k-1} + c\right] - \left[\frac{1}{2}(\vec{x}^*)^\top \vec{b} - \vec{b}^\top \vec{x}^* + c\right] \\
&= \frac{1}{2}\vec{x}_{k-1}^\top A \vec{x}_{k-1} - \vec{b}^\top \vec{x}_{k-1} - \frac{1}{2}\vec{b}^\top A^{-1}\vec{b} \\
&= \frac{1}{2}(A\vec{x}_{k-1} - \vec{b})^\top A^{-1}(A\vec{x}_{k-1} - \vec{b}) \text{ by symmetry of } A \\
&= \frac{1}{2}\vec{d}_k^\top A^{-1}\vec{d}_k \text{ by definition of } \vec{d}_k
\end{aligned}$$

Thus,

$$\begin{aligned}
R_k &= 1 - \frac{(\vec{d}_k^\top \vec{d}_k)^2}{2\vec{d}_k^\top A \vec{d}_k (f(\vec{x}_{k-1}) - f(\vec{x}^*))} \\
&= 1 - \frac{(\vec{d}_k^\top \vec{d}_k)^2}{\vec{d}_k^\top A \vec{d}_k \cdot \vec{d}_k^\top A^{-1} \vec{d}_k} \text{ by our latest simplification} \\
&= 1 - \frac{\vec{d}_k^\top \vec{d}_k}{\vec{d}_k^\top A \vec{d}_k} \cdot \frac{\vec{d}_k^\top \vec{d}_k}{\vec{d}_k^\top A^{-1} \vec{d}_k} \\
&\leq 1 - \left(\min_{\|\vec{d}\|=1} \frac{1}{\vec{d}^\top A \vec{d}}\right)\left(\min_{\|\vec{d}\|=1} \frac{1}{\vec{d}^\top A^{-1} \vec{d}}\right) \text{ since this makes the second term smaller} \\
&= 1 - \left(\max_{\|\vec{d}\|=1} \vec{d}^\top A \vec{d}\right)^{-1}\left(\max_{\|\vec{d}\|=1} \vec{d}^\top A^{-1} \vec{d}\right)^{-1} \\
&= 1 - \frac{\sigma_{\min}}{\sigma_{\max}} \text{ where } \sigma_{\min} \text{ and } \sigma_{\max} \text{ are the minimum and maximum singular values of } A \\
&= 1 - \frac{1}{\operatorname{cond} A}
\end{aligned}$$

It took a considerable amount of algebra, but we proved an important fact:

Convergence of gradient descent on f depends on the conditioning of A.

That is, the better conditioned A is, the faster gradient descent will converge. Additionally, since $\operatorname{cond} A \geq 1$, we know that our gradient descent strategy above converges *unconditionally* to $\vec{x}^*$, although convergence can be slow when A is poorly-conditioned.

Figure NUMBER illustrates behavior of gradient descent for well- and poorly-conditioned matrices. As you can see, gradient descent can struggle to find the minimum of our quadratic function f when the eigenvalues of A have a wide spread.

10.2 Conjugate Gradients

Recall that solving $A\vec{x} = \vec{b}$ for $A \in \mathbb{R}^{n\times n}$ took $O(n^3)$ time. Reexamining the gradient descent strategy above, we see each iteration takes $O(n^2)$ time, since we must compute matrix-vector products between A, $\vec{x}_{k-1}$ and $\vec{d}_k$. Thus, if gradient descent takes more than n iterations, we might as well have applied Gaussian elimination, which will recover the *exact* solution in the same amount of time. Unfortunately, we are not able to show that gradient descent has to take a finite number of iterations, and in fact in poorly-conditioned cases it can take a huge number of iterations to find the minimum.

For this reason, we will design an algorithm that is *guaranteed* to converge in at most n steps, preserving the $O(n^3)$ worst-case timing for solving linear systems. Along the way, we will find that this algorithm in fact exhibits better convergence properties overall, making it a reasonable choice even if we do not run it to completion.

10.2.1 Motivation

Our derivation of the conjugate gradients algorithm is motivated by a fairly straightforward observation. Suppose we knew the solution $\vec{x}^*$ to $A\vec{x}^* = \vec{b}$. Then, we can write our quadratic form f in a different way:

$$\begin{aligned}
f(\vec{x}) &= \frac{1}{2}\vec{x}^\top A\vec{x} - \vec{b}^\top\vec{x} + c \text{ by definition}\\
&= \frac{1}{2}(\vec{x}-\vec{x}^*)^\top A(\vec{x}-\vec{x}^*) + \vec{x}^\top A\vec{x}^* - \frac{1}{2}(\vec{x}^*)^\top A\vec{x}^* - \vec{b}^\top\vec{x} + c\\
&\qquad \text{by adding and subtracting the same terms}\\
&= \frac{1}{2}(\vec{x}-\vec{x}^*)^\top A(\vec{x}-\vec{x}^*) + \vec{x}^\top\vec{b} - \frac{1}{2}(\vec{x}^*)^\top\vec{b} - \vec{b}^\top\vec{x} + c \text{ since } A\vec{x}^* = \vec{b}\\
&= \frac{1}{2}(\vec{x}-\vec{x}^*)^\top A(\vec{x}-\vec{x}^*) + \text{const. since the } \vec{x}^\top\vec{b} \text{ terms cancel}
\end{aligned}$$

Thus, up to a constant shift f is the same as the product $\frac{1}{2}(\vec{x}-\vec{x}^*)^\top A(\vec{x}-\vec{x}^*)$. Of course, we do not know $\vec{x}^*$, but this observation shows us the nature of f; it is simply measuring the distance from $\vec{x}$ to $\vec{x}^*$ with respect to the "A-norm" $\|\vec{v}\|_A^2 \equiv \vec{v}^\top A\vec{v}$.

In fact, since A is symmetric and positive definite, even if it might be slow to carry out in practice, we know that it can be factorized using the Cholesky strategy as $A = LL^\top$. With this factorization in hand, f takes an even nicer form:

$$f(\vec{x}) = \frac{1}{2}\|L^\top(\vec{x}-\vec{x}^*)\|_2^2 + \text{const.}$$

Since $L^\top$ is an invertible matrix, this norm truly is a distance measure between $\vec{x}$ and $\vec{x}^*$.

Define $\vec{y} \equiv L^\top \vec{x}$ and $\vec{y}^* \equiv L^\top \vec{x}^*$. Then, from this new standpoint, we are minimizing $\bar{f}(\vec{y}) = \|\vec{y} - \vec{y}^*\|_2^2$. Of course, if we truly could get to this point via Cholesky factorization, optimizing $\bar{f}$ would be exceedingly easy, but to derive a scheme for this minimization without L we consider the possibility of minimizing $\bar{f}$ using only line searches derived in §10.1.1.

We make a simple observation about minimizing our simplified function $\bar{f}$ using such a strategy, illustrated in Figure NUMBER:

Proposition 10.1. *Suppose $\{\vec{w}_1, \ldots, \vec{w}_n\}$ are orthogonal in $\mathbb{R}^n$. Then, $\bar{f}$ is minimized in at most n steps by line searching in direction $\vec{w}_1$, then direction $\vec{w}_2$, and so on.*

Proof. Take the columns of $Q \in \mathbb{R}^{n\times n}$ to be the vectors $\vec{w}_i$; Q is an orthogonal matrix. Since Q is orthogonal, we can write $\bar{f}(\vec{y}) = \|\vec{y} - \vec{y}^*\|_2^2 = \|Q^\top \vec{y} - Q^\top \vec{y}^*\|_2^2$; in other words, we rotate so that $\vec{w}_1$ is the first standard basis vector, $\vec{w}_2$ is the second, and so on. If we write $\vec{z} \equiv Q^\top \vec{y}$ and $\vec{z}^* \equiv Q^\top \vec{y}^*$, then clearly after the first iteration we must have $z_1 = z_1^*$, after the second iteration $z_2 = z_2^*$, and so on. After n steps we reach $z_n = z_n^*$, yielding the desired result. □

So, optimizing $\bar{f}$ can always be accomplished using n line searches so long as those searches are in *orthogonal* directions.

All we did to pass from f to $\bar{f}$ is rotate coordinates using $L^\top$. Such a linear transformation takes straight lines to straight lines, so doing a line search on $\bar{f}$ along some vector $\vec{w}$ is equivalent to doing a line search along $(L^\top)^{-1}\vec{w}$ on our original quadratic function f. Conversely, if we do n line searches on f on directions $\vec{v}_i$ such that $L^\top \vec{v}_i \equiv \vec{w}_i$ are orthogonal, then by Proposition 10.1 we must have found $\vec{x}^*$. Notice that asking $\vec{w}_i \cdot \vec{w}_j = 0$ is the same as asking

$$0 = \vec{w}_i \cdot \vec{w}_j = (L^\top \vec{v}_i)^\top (L^\top \vec{v}_j) = \vec{v}_i^\top (LL^\top)\vec{v}_j = \vec{v}_i^\top A \vec{v}_j.$$

We have just argued an important corollary to Proposition 10.1. Define *conjugate* vectors as follows:

Definition 10.1 (A-conjugate vectors)**.** *Two vectors $\vec{v}, \vec{w}$ are A-*conjugate *if $\vec{v}^\top A\vec{w} = 0$.*

Then, based on our discussion we have shown:

Proposition 10.2. *Suppose $\{\vec{v}_1, \ldots, \vec{v}_n\}$ are A-conjugate. Then, f is minimized in at most n steps by line searching in direction $\vec{v}_1$, then direction $\vec{v}_2$, and so on.*

At a high level, the conjugate gradients algorithm simply applies this proposition, generating and searching along A-conjugate directions rather than moving along $-\nabla f$. Notice that this result might appear somewhat counterintuitive: We do *not* necessarily move along the steepest descent direction, but rather ask that our *set* of search directions satisfies a global criterion to make sure we do not repeat work. This setup guarantees convergence in a finite number of iterations and acknowledges the structure of f in terms of $\bar{f}$ discussed above.

Recall that we motivated A-conjugate directions by noting that they are orthogonal after applying $L^\top$ from the factorization $A = LL^\top$. From this standpoint, we are dealing with two dot products: $\vec{x}_i \cdot \vec{x}_j$ and $\vec{y}_i \cdot \vec{y}_j \equiv (L^\top \vec{x}_i)\cdot(L^\top \vec{x}_j) = x_i^\top LL^\top \vec{x}_j = \vec{x}_i^\top A\vec{x}_j$. These two products will figure into our subsequent discussion in equal amounts, so we denote the "A-inner product" as

$$\langle \vec{u}, \vec{v}\rangle_A \equiv (L^\top \vec{u}) \cdot (L^\top \vec{v}) = \vec{u}^\top A\vec{v}.$$

10.2.2 Suboptimality of Gradient Descent

So far, we know that if we can find n A-conjugate search directions, we can solve $A\vec{x} = \vec{b}$ in n steps via line searches along these directions. What remains is to uncover a strategy for finding these directions as efficiently as possible. To do so, we will examine one more property of the gradient descent algorithm that will inspire a more refined approach.

Suppose we are at $\vec{x}_k$ during an iterative line search method on $f(\vec{x})$; we will call the direction of steepest descent of f at $\vec{x}_k$ the *residual* $\vec{r}_k \equiv \vec{b} - A\vec{x}_k$. We may not decide to do a line search along $\vec{r}_k$ as in gradient descent, since the gradient directions are not necessarily A-conjugate. So, generalizing slightly, we will find $\vec{x}_{k+1}$ via line search along a yet-undetermined direction $\vec{v}_{k+1}$.

From our derivation of gradient descent, we should choose $\vec{x}_{k+1} = \vec{x}_k + \alpha_{k+1}\vec{v}_{k+1}$, where α_{k+1} is given by

$$\alpha_{k+1} = \frac{\vec{v}_{k+1}^\top \vec{r}_k}{\vec{v}_{k+1}^\top A\vec{v}_{k+1}}.$$

Applying this expansion of $\vec{x}_{k+1}$, we can write an alternative update formula for the residual:

$$\begin{aligned}
\vec{r}_{k+1} &= \vec{b} - A\vec{x}_{k+1} \\
&= \vec{b} - A(\vec{x}_k + \alpha_{k+1}\vec{v}_{k+1}) \text{ by definition of } \vec{x}_{k+1} \\
&= (\vec{b} - A\vec{x}_k) - \alpha_{k+1}A\vec{v}_{k+1} \\
&= \vec{r}_k - \alpha_{k+1}A\vec{v}_{k+1} \text{ by definition of } \vec{r}_k
\end{aligned}$$

This formula holds regardless of our choice of $\vec{v}_{k+1}$ and can be applied to any iterative line search method.

In the case of gradient descent, however, we chose $\vec{v}_{k+1} \equiv \vec{r}_k$. This choice gives a recurrence relation:

$$\vec{r}_{k+1} = \vec{r}_k - \alpha_{k+1}A\vec{r}_k.$$

This simple formula leads to an instructive proposition:

Proposition 10.3. *When performing gradient descent on f, span$\{\vec{r}_0, \ldots, \vec{r}_k\}$ = span$\{\vec{r}_0, A\vec{r}_0, \ldots, A^k\vec{r}_0\}$.*

Proof. This statement follows inductively from our formula for $\vec{r}_{k+1}$ above. □

The structure we are uncovering is beginning to look a lot like the Krylov subspace methods mentioned in Chapter 5: This is not a mistake!

Gradient descent gets to $\vec{x}_k$ by moving along $\vec{r}_0$, then $\vec{r}_1$, and so on through $\vec{r}_k$. Thus, in the end we know that the iterate $\vec{x}_k$ of gradient descent on f lies somewhere in the plane $\vec{x}_0 + \text{span}\{\vec{r}_0, \vec{r}_1, \ldots, \vec{r}_{k-1}\} = \vec{x}_0 + \text{span}\{\vec{r}_0, A\vec{r}_0, \ldots, A^{k-1}\vec{r}_0\}$, by Proposition 10.3. Unfortunately, it is *not* true that if we run gradient descent, the iterate $\vec{x}_k$ is optimal in this subspace. In other words, in general it can be the case that

$$\vec{x}_k - \vec{x}_0 \neq \underset{\vec{v} \in \text{span}\{\vec{r}_0, A\vec{r}_0, \ldots, A^{k-1}\vec{r}_0\}}{\arg\min} f(\vec{x}_0 + \vec{v})$$

Ideally, switching this inequality to an equality would make sure that generating $\vec{x}_{k+1}$ from $\vec{x}_k$ does not "cancel out" any work done during iterations 1 to $k-1$.

If we reexamine our proof of Proposition 10.1 with this fact in mind, we can make an observation suggesting how we might use conjugacy to improve gradient descent. In particular, once z_i switches to z_i^*, it never changes value in a future iteration. In other words, after rotating from $\vec{z}$ to $\vec{x}$ the following proposition holds:

Proposition 10.4. *Take $\vec{x}_k$ to be the k-th iterate of the process from Proposition 10.1 after searching along $\vec{v}_k$. Then,*

$$\vec{x}_k - \vec{x}_0 = \underset{\vec{v} \in span\,\{\vec{v}_1, \ldots, \vec{v}_k\}}{\arg\min} f(\vec{x}_0 + \vec{v})$$

Thus, in the best of all possible worlds, in an attempt to outdo gradient descent we might hope to find A-conjugate directions $\{\vec{v}_1, \ldots, \vec{v}_n\}$ such that $\operatorname{span}\{\vec{v}_1, \ldots, \vec{v}_k\} = \operatorname{span}\{\vec{r}_0, A\vec{r}_0, \ldots, A^{k-1}\vec{r}_0\}$ for each k; then our iterative scheme is guaranteed to do no worse than gradient descent during any given iteration. But, greedily we wish to do so without orthogonalization or storing more than a finite number of vectors at a time.

10.2.3 Generating A-Conjugate Directions

Of course, given any set of directions, we can make them A-orthogonal using a method like Gram-Schmidt orthogonalization. Unfortunately, orthogonalizing $\{\vec{r}_0, A\vec{r}_0, \ldots\}$ to find the set of search directions is expensive and would require us to maintain a complete list of directions $\vec{v}_k$; this construction likely would exceed the time and memory requirements even of Gaussian elimination. We will reveal one final observation *about* Gram-Schmidt that makes conjugate gradients tractable by generating conjugate directions without an expensive orthogonalization process.

Ignoring these issues, we might write a "method of conjugate directions" as follows:

$$\begin{aligned}
\textbf{Update search direction (bad Gram-Schmidt step): } & \vec{v}_k = A^{k-1}\vec{r}_0 - \sum_{i<k} \frac{\langle A^{k-1}\vec{r}_0, \vec{v}_i\rangle_A}{\langle \vec{v}_i, \vec{v}_i\rangle_A}\vec{v}_i \\
\textbf{Line search: } & \alpha_k = \frac{\vec{v}_k^\top \vec{r}_{k-1}}{\vec{v}_k^\top A\vec{v}_k} \\
\textbf{Update estimate: } & \vec{x}_k = \vec{x}_{k-1} + \alpha_k\vec{v}_k \\
\textbf{Update residual: } & \vec{r}_k = \vec{r}_{k-1} - \alpha_k A\vec{v}_k
\end{aligned}$$

Here, we compute the k-th search direction $\vec{v}_k$ simply by projecting $\vec{v}_1, \ldots, \vec{v}_{k-1}$ out of the vector $A^{k-1}\vec{r}_0$. This algorithm obviously has the property $\operatorname{span}\{\vec{v}_1, \ldots, \vec{v}_k\} = \operatorname{span}\{\vec{r}_0, A\vec{r}_0, \ldots, A^{k-1}\vec{r}_0\}$ suggested in §10.2.2, but has two issues:

1. Similar to power iteration for eigenvectors, the power $A^{k-1}\vec{r}_0$ is likely to look mostly like the first eigenvector of A, making the projection more and more poorly conditioned

2. We have to keep $\vec{v}_1, \ldots, \vec{v}_{k-1}$ around to compute $\vec{v}_k$; thus, each iteration of this algorithm needs more memory and time than the last.

We can fix the first issue in a relatively straightforward manner. In particular, right now we project the previous search directions out of $A^{k-1}\vec{r}_0$, but in reality we can project out previous directions from *any* vector $\vec{w}$ so long as

$$\vec{w} \in \operatorname{span}\{\vec{r}_0, A\vec{r}_0, \ldots, A^{k-1}\vec{r}_0\} \backslash \operatorname{span}\{\vec{r}_0, A\vec{r}_0, \ldots, A^{k-2}\vec{r}_0\},$$

that is, as long as $\vec{w}$ has some component in the new part of the space.

An alternative choice of $\vec{w}$ with this property is the residual $\vec{r}_{k-1}$. This property follows from the residual update $\vec{r}_k = \vec{r}_{k-1} - \alpha_k A\vec{v}_k$; in this expression, we multiply $\vec{v}_k$ by A, introducing the new power of A that we need. This choice also more closely mimics the gradient descent algorithm, which took $\vec{v}_k = \vec{r}_{k-1}$. Thus we can update our algorithm a bit:

$$\begin{aligned}\textbf{Update search direction (bad Gram-Schmidt on residual):}\ \vec{v}_k &= \vec{r}_{k-1} - \sum_{i<k} \frac{\langle \vec{r}_{k-1}, \vec{v}_i\rangle_A}{\langle \vec{v}_i, \vec{v}_i\rangle_A}\vec{v}_i \\ \textbf{Line search:}\ \alpha_k &= \frac{\vec{v}_k^\top \vec{r}_{k-1}}{\vec{v}_k^\top A\vec{v}_k} \\ \textbf{Update estimate:}\ \vec{x}_k &= \vec{x}_{k-1} + \alpha_k \vec{v}_k \\ \textbf{Update residual:}\ \vec{r}_k &= \vec{r}_{k-1} - \alpha_k A\vec{v}_k\end{aligned}$$

Now we do not do arithmetic involving the poorly-conditioned vector $A^{k-1}\vec{r}_0$ but still have the "memory" problem above.

In fact, the surprising observation about the orthogonalizing step above is that most terms in the sum are exactly zero! This amazing observation allows each iteration of conjugate gradients to happen without increasing memory usage. We memorialize this result in a proposition:

Proposition 10.5. *In the "conjugate direction" method above, $\langle \vec{r}_k, \vec{v}_\ell\rangle_A = 0$ for all $\ell < k$.*

Proof. We proceed inductively. There is nothing to prove for the base case $k = 1$, so assume $k > 1$ and that the result holds for all $k' < k$. By the residual update formula, we know:

$$\langle \vec{r}_k, \vec{v}_\ell\rangle_A = \langle \vec{r}_{k-1}, \vec{v}_\ell\rangle_A - \alpha_k\langle A\vec{v}_k, \vec{v}_\ell\rangle_A = \langle \vec{r}_{k-1}, \vec{v}_\ell\rangle_A - \alpha_k\langle \vec{v}_k, A\vec{v}_\ell\rangle_A,$$

where the second equality follows from symmetry of A.

First, suppose $\ell < k-1$. Then the first term of the difference above is zero by induction. Furthermore, by construction $A\vec{v}_\ell \in \operatorname{span}\{\vec{v}_1, \ldots, \vec{v}_{\ell+1}\}$, so since we have constructed our search directions to be A-conjugate we know the second term must be zero as well.

To conclude the proof, we consider the case $\ell = k-1$. Using the residual update formula, we know:

$$A\vec{v}_{k-1} = \frac{1}{\alpha_{k-1}}(\vec{r}_{k-2} - \vec{r}_{k-1})$$

Premultiplying by $\vec{r}_k$ shows:

$$\langle \vec{r}_k, \vec{v}_{k-1}\rangle_A = \frac{1}{\alpha_{k-1}}\vec{r}_k^\top(\vec{r}_{k-2} - \vec{r}_{k-1})$$

The difference $\vec{r}_{k-2} - \vec{r}_{k-1}$ lives in $\operatorname{span}\{\vec{r}_0, A\vec{r}_0, \ldots, A^{k-1}\vec{r}_0\}$, by the residual update formula. Proposition 10.4 shows that $\vec{x}_k$ is optimal in this subspace. Since $\vec{r}_k = -\nabla f(\vec{x}_k)$, this implies that we must have $\vec{r}_k \perp \operatorname{span}\{\vec{r}_0, A\vec{r}_0, \ldots, A^{k-1}\vec{r}_0\}$, since otherwise there would exist a direction in the subspace to move from $\vec{x}_k$ to decrease f. In particular, this shows the inner product above $\langle \vec{r}_k, \vec{v}_{k-1}\rangle_A = 0$, as desired. □

Thus, our proof above shows that we can find a new direction $\vec{v}_k$ as follows:

$$\begin{aligned}\vec{v}_k &= \vec{r}_{k-1} - \sum_{i<k} \frac{\langle \vec{r}_{k-1}, \vec{v}_i \rangle_A}{\langle \vec{v}_i, \vec{v}_i \rangle_A} \vec{v}_i \text{ by the Gram-Schmidt formula} \\ &= \vec{r}_{k-1} - \frac{\langle \vec{r}_{k-1}, \vec{v}_{k-1} \rangle_A}{\langle \vec{v}_{k-1}, \vec{v}_{k-1} \rangle_A} \vec{v}_{k-1} \text{ because the remaining terms vanish}\end{aligned}$$

Since the summation over i disappears, the cost of computing $\vec{v}_k$ has no dependence on k.

10.2.4 Formulating the Conjugate Gradients Algorithm

Now that we have a strategy that yields A-conjugate search directions with relatively little computational effort, we simply apply this strategy to formulate the conjugate gradients algorithm. In particular, suppose $\vec{x}_0$ is an initial guess of the solution to $A\vec{x} = \vec{b}$, and take $\vec{r}_0 \equiv \vec{b} - A\vec{x}_0$. For convenience take $\vec{v}_0 \equiv \vec{0}$. Then, we iteratively update $\vec{x}_{k-1}$ to $\vec{x}_k$ using a series of steps for $k = 1, 2, \ldots$:

$$\begin{aligned}\textbf{Update search direction: } \vec{v}_k &= \vec{r}_{k-1} - \frac{\langle \vec{r}_{k-1}, \vec{v}_{k-1} \rangle_A}{\langle \vec{v}_{k-1}, \vec{v}_{k-1} \rangle_A} \vec{v}_{k-1} \\ \textbf{Line search: } \alpha_k &= \frac{\vec{v}_k^\top \vec{r}_{k-1}}{\vec{v}_k^\top A \vec{v}_k} \\ \textbf{Update estimate: } \vec{x}_k &= \vec{x}_{k-1} + \alpha_k \vec{v}_k \\ \textbf{Update residual: } \vec{r}_k &= \vec{r}_{k-1} - \alpha_k A \vec{v}_k\end{aligned}$$

This iterative scheme is only a minor adjustment to the gradient descent algorithm but has many desirable properties by construction:

- $f(\vec{x}_k)$ is upper-bounded by that of the k-th iterate of gradient descent
- The algorithm converges to $\vec{x}^*$ in n steps
- At each step, the iterate $\vec{x}_k$ is optimal in the subspace spanned by the first k search directions

In the interests of squeezing maximal numerical quality out of conjugate gradients, we can try to simplify the numerics of the expressions above. For instance, if we plug search direction update into the formula for α_k, by orthogonality we can write:

$$\alpha_k = \frac{\vec{r}_{k-1}^\top \vec{r}_{k-1}}{\vec{v}_k^\top A \vec{v}_k}$$

The numerator of this fraction now is guaranteed to be nonnegative without numerical precision issues.

Similarly, we can define a constant β_k to split the search direction update into two steps:

$$\begin{aligned}\beta_k &\equiv -\frac{\langle \vec{r}_{k-1}, \vec{v}_{k-1} \rangle_A}{\langle \vec{v}_{k-1}, \vec{v}_{k-1} \rangle_A} \\ \vec{v}_k &= \vec{r}_{k-1} + \beta_k \vec{v}_{k-1}\end{aligned}$$

We can simplify our formula for β_k:

$$\begin{aligned}
\beta_k &\equiv -\frac{\langle \vec{r}_{k-1}, \vec{v}_{k-1}\rangle_A}{\langle \vec{v}_{k-1}, \vec{v}_{k-1}\rangle_A} \\
&= -\frac{\vec{r}_{k-1}A\vec{v}_{k-1}}{\vec{v}_{k-1}^\top A\vec{v}_{k-1}} \text{ by definition of } \langle\cdot,\cdot\rangle_A \\
&= -\frac{\vec{r}_{k-1}^\top(\vec{r}_{k-2}-\vec{r}_{k-1})}{\alpha_{k-1}\vec{v}_{k-1}^\top A\vec{v}_{k-1}} \text{ since } \vec{r}_k = \vec{r}_{k-1} - \alpha_k A\vec{v}_k \\
&= \frac{\vec{r}_{k-1}^\top \vec{r}_{k-1}}{\alpha_{k-1}\vec{v}_{k-1}^\top A\vec{v}_{k-1}} \text{ by a calculation below} \\
&= \frac{\vec{r}_{k-1}^\top \vec{r}_{k-1}}{\vec{r}_{k-2}^\top \vec{r}_{k-2}} \text{ by our last formula for } \alpha_k
\end{aligned}$$

This expression reveals that $\beta_k \geq 0$, a property which might not have held after numerical precision issues. We do have one remaining calculation below:

$$\begin{aligned}
\vec{r}_{k-2}^\top \vec{r}_{k-1} &= \vec{r}_{k-2}^\top(\vec{r}_{k-2} - \alpha_{k-1}A\vec{v}_{k-1}) \text{ by our residual update formula} \\
&= \vec{r}_{k-2}^\top \vec{r}_{k-2} - \frac{\vec{r}_{k-2}^\top \vec{r}_{k-2}}{\vec{v}_{k-1}^\top A\vec{v}_{k-1}}\vec{r}_{k-2}^\top A\vec{v}_{k-1} \text{ by our formula for } \alpha_k \\
&= \vec{r}_{k-2}^\top \vec{r}_{k-2} - \frac{\vec{r}_{k-2}^\top \vec{r}_{k-2}}{\vec{v}_{k-1}^\top A\vec{v}_{k-1}}\vec{v}_{k-1}^\top A\vec{v}_{k-1} \text{ by the update for } \vec{v}_k \text{ and } A\text{-conjugacy of the } \vec{v}_k\text{'s} \\
&= 0, \text{ as needed.}
\end{aligned}$$

With these simplifications, we have an alternative version of conjugate gradients:

$$\begin{aligned}
\textbf{Update search direction: } \beta_k &= \frac{\vec{r}_{k-1}^\top \vec{r}_{k-1}}{\vec{r}_{k-2}^\top \vec{r}_{k-2}} \\
\vec{v}_k &= \vec{r}_{k-1} + \beta_k \vec{v}_{k-1} \\
\textbf{Line search: } \alpha_k &= \frac{\vec{r}_{k-1}^\top \vec{r}_{k-1}}{\vec{v}_k^\top A\vec{v}_k} \\
\textbf{Update estimate: } \vec{x}_k &= \vec{x}_{k-1} + \alpha_k \vec{v}_k \\
\textbf{Update residual: } \vec{r}_k &= \vec{r}_{k-1} - \alpha_k A\vec{v}_k
\end{aligned}$$

For numerical reasons, occasionally rather than using the update formula for $\vec{r}_k$ it is advisable to use the residual formula $\vec{r}_k = \vec{b} - A\vec{x}_k$. This formula requires an extra matrix-vector multiply but repairs numerical "drift" caused by finite-precision rounding. Notice that there is no need to store a long list of previous residuals or search directions: Conjugate gradients takes a constant amount of space from iteration to iteration.

10.2.5 Convergence and Stopping Conditions

By construction the conjugate gradients (CG) algorithm is guaranteed to converge no more slowly than gradient descent on f, while being no harder to implement and having a number of other positive properties. A detailed discussion of CG convergence is out of the scope of our discussion, but in general the algorithm behaves best on matrices with evenly-distributed eigenvalues over a small range. One rough estimate paralleling our estimate in §10.1.2 shows that the CG algorithm satisfies:

$$\frac{f(\vec{x}_k) - f(\vec{x}^*)}{f(\vec{x}_0) - f(\vec{x}^*)} \leq 2\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^k$$

where $\kappa \equiv \operatorname{cond} A$. More generally, the number of iterations needed for conjugate gradient to reach a given error value usually can be bounded by a function of $\sqrt{\kappa}$, whereas bounds for convergence of gradient descent are proportional to κ.

We know that conjugate gradients is guaranteed to converge to $\vec{x}^*$ exactly in n steps, but when n is large it may be preferable to stop earlier than that. In fact, the formula for β_k will divide by zero when the residual gets very short, which can cause numerical precision issues near the minimum of f. Thus, in practice CG usually his halted when the ration ${\|\vec{r}_k\|}/{\|\vec{r}_0\|}$ is sufficiently small.

10.3 Preconditioning

We now have two powerful iterative schemes for finding solutions to $A\vec{x} = \vec{b}$ when A is symmetric and positive definite: gradient descent and conjugate gradients. Both strategies converge *unconditionally*, meaning that regardless of the initial guess $\vec{x}_0$ with enough iterations we will get arbitrarily close to the true solution $\vec{x}^*$; in fact, conjugate gradients guarantees we will reach $\vec{x}^*$ exactly in a finite number of iterations. Of course, the time taken to reach a solution of $A\vec{x} = \vec{b}$ for both of these methods is directly proportional to the number of iterations needed to reach $\vec{x}^*$ within an acceptable tolerance. Thus, it makes sense to tune a strategy as much as possible to minimize the number of iterations for convergence.

To this end, we notice that we are able to characterize the convergence rates of both algorithms and many more related iterative techniques in terms of the condition number $\operatorname{cond} A$. That is, the small the value of $\operatorname{cond} A$, the less time it should take to solve $A\vec{x} = \vec{b}$. Notice that this situation is somewhat different for Gaussian elimination, which takes the same amount of steps regardless of A; in other words, the conditioning of A affects not only the quality of the output of iterative methods but also the speed at which $\vec{x}^*$ is approached.

Of course, for any invertible matrix P, it is the case that solving $PA\vec{x} = P\vec{b}$ is equivalent to solving $A\vec{x} = \vec{b}$. The trick, however, is that the condition number of PA does not need to be the same as that of A; in the (unachievable) extreme, of course if we took $P = A^{-1}$ we would remove conditioning issues altogether! More generally, suppose $P \approx A^{-1}$. Then, we expect $\operatorname{cond} PA \ll \operatorname{cond} A$, and thus it may be advisable to apply P before solving the linear system. In this case, we will call P a *preconditioner*.

While the idea of preconditioning appears attractive, two issues remain:

1. While A may be symmetric and positive definite, the product PA in general will not enjoy these properties.

2. We need to find $P \approx A^{-1}$ that is easier to compute than A^{-1} itself.

We address these issues in the sections below.

10.3.1 CG with Preconditioning

We will focus our discussion on conjugate gradients since it has better convergence properties, although most of our constructions will apply fairly easily to gradient descent as well. From this standpoint, if we look over our constructions in §10.2.1 it is clear that our construction of CG depends strongly on both the symmetry and positive definiteness of A, so running CG on PA usually will not converge out of the box.

Suppose, however, that the preconditioner P is itself symmetric and positive definite. This is a reasonable assumption since A^{-1} must satisfy these properties. Then, we again can write a Cholesky factorization of the inverse $P^{-1} = EE^\top$. We make the following observation:

Proposition 10.6. *The condition number of PA is the same as that of $E^{-1}AE^{-\top}$.*

Proof. We show that PA and $E^{-1}AE^{-\top}$ have the same singular values; the condition number is the ratio of the maximum to the minimum singular value, so this statement is more than sufficient. In particular, it is clear that $E^{-1}AE^{-\top}$ is symmetric and positive definite, so its eigenvectors are its singular values. Thus, suppose $E^{-1}AE^{-\top}\vec{x} = \lambda\vec{x}$. We know $P^{-1} = EE^\top$, so $P = E^{-\top}E^{-1}$., Thus, if we pre-multiply both sides of our eigenvector expression by $E^{-\top}$ we find $PAE^{-\top}\vec{x} = \lambda E^{-\top}\vec{x}$. Defining $\vec{y} \equiv E^{-\top}\vec{x}$ shows $PA\vec{y} = \lambda\vec{y}$. Thus, PA and $E^{-1}AE^{-\top}$ both have full eigenspaces and identical eigenvalues. □

This proposition implies that if we do CG on the symmetric positive definite matrix $E^{-1}AE^{-\top}$, we will receive the same conditioning benefits we would have if we could iterate on PA. As in our proof of Proposition 10.6, we could accomplish our new solve for $\vec{y} = E^\top\vec{x}$ in two steps:

1. Solve $E^{-1}AE^{-\top}\vec{y} = E^{-1}\vec{b}$.
2. Solve $\vec{x} = E^{-\top}\vec{y}$.

Finding E would be integral to this strategy but likely is difficult, but we will prove shortly that it is unnecessary.

Ignoring the computation of E, we could accomplish step 1 using CG as follows:

$$\textbf{Update search direction: } \beta_k = \frac{\vec{r}_{k-1}^\top \vec{r}_{k-1}}{\vec{r}_{k-2}^\top \vec{r}_{k-2}}$$

$$\vec{v}_k = \vec{r}_{k-1} + \beta_k \vec{v}_{k-1}$$

$$\textbf{Line search: } \alpha_k = \frac{\vec{r}_{k-1}^\top \vec{r}_{k-1}}{\vec{v}_k^\top E^{-1}AE^{-\top}\vec{v}_k}$$

$$\textbf{Update estimate: } \vec{y}_k = \vec{y}_{k-1} + \alpha_k \vec{v}_k$$

$$\textbf{Update residual: } \vec{r}_k = \vec{r}_{k-1} - \alpha_k E^{-1}AE^{-\top}\vec{v}_k$$

This iterative scheme will converge according to the conditioning of our matrix $E^{-1}AE^{-\top}$.

Define $\tilde{r}_k \equiv E\vec{r}_k$, $\tilde{v}_k \equiv E^{-\top}\vec{v}_k$, and $\vec{x}_k \equiv E\vec{y}_k$. If we recall the relationship $P = E^{-\top}E^{-1}$, we can rewrite our preconditioned conjugate gradients iteration using these new variables:

$$\textbf{Update search direction: } \beta_k = \frac{\tilde{r}_{k-1}^\top P\tilde{r}_{k-1}}{\tilde{r}_{k-2}^\top P\tilde{r}_{k-2}}$$

$$\begin{aligned}
\tilde{v}_k &= P\tilde{r}_{k-1} + \beta_k \tilde{v}_{k-1} \\
\textbf{Line search: } \alpha_k &= \frac{\vec{r}_{k-1}^\top P \vec{r}_{k-1}}{\tilde{v}_k^\top A \tilde{v}_k} \\
\textbf{Update estimate: } \vec{x}_k &= \vec{x}_{k-1} + \alpha_k \tilde{v}_k \\
\textbf{Update residual: } \tilde{r}_k &= \tilde{r}_{k-1} - \alpha_k A \tilde{v}_k
\end{aligned}$$

This iteration does not depend on the Cholesky factorization of P^{-1} at all, but instead can be carried out using solely applications of P and A. It is easy to see that $\vec{x}_k \to \vec{x}^*$, so in fact this scheme enjoys the benefits of preconditioning without the need to factor the preconditioner.

As a side note, even more effective preconditioning can be carried out by replacing A with PAQ for a second matrix Q, although this second matrix will require additional computations to apply. This example represents a common trade-off: If a preconditioner itself takes too long to apply in a single iteration of CG or another method, it may not be worth the reduced number of iterations.

10.3.2 Common Preconditioners

Finding good preconditioners in practice is as much an art as it is a science. Finding the best approximation P of A^{-1} depends on the structure of A, the particular application at hand, and so on. Even rough approximations, however, can help convergence considerably, so rarely do applications of CG appear that do *not* use a preconditioner.

The best strategy for formulating P often is application-specific, and an interesting engineering approximation problem involves designing and testing assorted P's for the best preconditioner. Two common strategies are below:

- A *diagonal* (or "*Jacobi*") preconditioner simply takes P to be the matrix obtained by inverting diagonal elements of A; that is, P is the diagonal matrix with entries $1/a_{ii}$. This strategy can alleviate nonuniform scaling from row to row, which is a common cause of poor conditioning.

- The *sparse approximate inverse* preconditioner is formulated by solving a subproblem $\min_{P \in S} \|AP - I\|_{\text{Fro}}$, where P is restricted to be in a set S of matrices over which it is less difficult to optimize such an objective. For instance, a common constraint is to prescribe a sparsity pattern for P, e.g. only nonzeros on the diagonal or where A has nonzeros.

- The *incomplete Cholesky* precondtioner factors $A \approx L_* L_*^\top$ and then approximates A^{-1} by solving the appropriate forward- and back-substitution problems. For instance, a popular strategy involves going through the steps of Cholesky factorization but only saving the output in positions (i, j) where $a_{ij} \neq 0$.

- The nonzero values in A can be considered a graph, and removing edges in the graph or grouping nodes may disconnect assorted components; the resulting system is block-diagonal after permuting rows and columns and thus can be solved using a sequence of smaller solves. Such a *domain decomposition* strategy can be effective for linear systems arising from differential equations such as those considered in Chapter NUMBER.

Some preconditioners come with bounds describing changes to the conditioning of A after replacing it with PA, but for the most part these are heuristic strategies that should be tested and refined.

10.4 Other Iterative Schemes

The algorithms we have developed in detail this chapter apply for solving $A\vec{x} = \vec{b}$ when A is square, symmetric, and positive definite. We have focused on this case because it appears so often in practice, but there are cases when A is asymmetric, indefinite, or even rectangular. It is out of the scope of our discussion to derive iterative algorithms in each case, since many require some specialized analysis or advanced development, but we summarize some techniques here from a high-level (CITE EACH):

- *Splitting* methods decompose $A = M - N$ and note that $A\vec{x} = \vec{b}$ is equivalent to $M\vec{x} = N\vec{x} + \vec{b}$. If M is easy to invert, then a fixed-point scheme can be derived by writing $M\vec{x}_k = N\vec{x}_{k-1} + \vec{b}$ (CITE); these techniques are easy to implement but have convergence depending on the spectrum of the matrix $G = M^{-1}N$ and in particular can diverge when the spectral radius of G is greater than one. One popular choice of M is the diagonal of A. Methods such as *successive over-relaxation* (SOR) weight these two terms for better convergence.

- The *conjugate gradient normal equation residual* (CGNR) method simply applies the CG algorithm to the normal equations $A^\top A\vec{x} = A^\top\vec{b}$. This method is simple to implement and guaranteed to converge so long as A is full-rank, but convergence can be slow thanks to poor conditioning of $A^\top A$ as discussed in Chapter NUMBER.

- The *conjugate gradient normal equation error* (CGNE) method similarly solves $AA^\top\vec{y} = \vec{b}$; then the solution of $A\vec{x} = \vec{b}$ is simply $A^\top\vec{y}$.

- Methods such as MINRES and SYMMLQ apply to symmetric but not necessarily positive definite matrices A by replacing our quadratic form $f(\vec{x})$ with $g(\vec{x}) \equiv \|\vec{b} - A\vec{x}\|_2$; this function g is minimized at solutions to $A\vec{x} = \vec{b}$ regardless of the definiteness of A.

- Given the poor conditioning of CGNR and CGNE, the LSQR and LSMR algorithms also minimize $g(\vec{x})$ with fewer assumptions on A, in particular allowing for solution of least-squares systems..

- Generalized methods including GMRES, QMR, BiCG, CGS, and BiCGStab solve $A\vec{x} = \vec{b}$ with the only caveat that A is square and invertible. They optimize similar energies but often have to store more information about previous iterations and may have to factor intermediate matrices to guarantee convergence with such generality.

- Finally, the *Fletcher-Reeves*, *Polak-Ribière*, and other methods return to the more general problem of minimizing a non-quadratic function f, applying conjugate gradient steps to finding new line search directions. Functions f that are well-approximated by quadratics can be minimized very effectively using these strategies, although they do not necessarily make use of the Hessian; for instance, the Fletcher-Reeves method simply replaces the residual in CG iterations with the negative gradient $-\nabla f$. It is possible to characterize convergence of these methods when they are accompanied with sufficiently effective line search strategies.

Many of these algorithms are nearly as easy to implement as CG or gradient descent, and many implementations exist that simply require inputting A and $\vec{b}$. Many of the algorithms listed above require application of both A and $A^\top$, which can be a technical challenge in some cases. As a rule of thumb, the more generalized a method is—that is, the fewer the assumptions a method makes on the structure of the matrix A—the more iterations it is likely to take to compensate for this lack of assumptions. This said, there are no hard-and-fast rules simply by looking at A most successful iterative scheme, although limited theoretical discussion exists comparing the advantages and disadvantages of each of these methods (CITE).

10.5 Problems

- Derive CGNR and/or CGNE
- Derive MINRES
- Derive Fletcher-Reeves
- Slide 13 of `http://math.ntnu.edu.tw/~min/matrix_computation/Ch4_Slide4_CG_2011.pdf`

Part IV

Functions, Derivatives, and Integrals

Chapter 11

Interpolation

So far we have derived methods for *analyzing* functions f, e.g. finding their minima and roots. Evaluating $f(\vec{x})$ at a particular $\vec{x} \in \mathbb{R}^n$ might be expensive, but a fundamental assumption of the methods we developed in previous chapters is that we can obtain $f(\vec{x})$ when we want it, regardless of $\vec{x}$.

There are many contexts when this assumption is not realistic. For instance, if we take a photograph with a digital camera, we receive an $n \times m$ grid of pixel color values sampling the continuum of light coming into a camera lens. We might think of a photograph as a continuous function from image position (x, y) to color (r, g, b), but in reality we only know the image value at nm separated locations on the image plane. Similarly, in machine learning and statistics, often we only are given samples of a function at points where we collected data, and we must interpolate it to have values elsewhere; in a medical setting we may monitor a patient's response to different dosages of a drug but only can predict what will happen at a dosage we have not tried explicitly.

In these cases, before we can minimize a function, find its roots, or even compute values $f(\vec{x})$ at arbitrary locations $\vec{x}$, we need a model for interpolating $f(\vec{x})$ to all of $\mathbb{R}^n$ (or some subset thereof) given a collection of samples $f(\vec{x}_i)$. Of course, techniques solving this *interpolation* problem are inherently approximate, since we do not know the true values of f, so instead we seek for the interpolated function to be smooth and serve as a "reasonable" prediction of function values.

In this chapter, we will assume that the values $f(\vec{x}_i)$ are known with complete certainty; in this case we might as well think of the problem as extending f to the remainder of the domain without perturbing the value at any of the input locations. In Chapter NUMBER (WRITE ME IN 2014), we will consider the *regression* problem, in which the value $f(\vec{x}_i)$ is known with some uncertainty, in which case we may forgo matching $f(\vec{x}_i)$ completely in favor of making f more smooth.

11.1 Interpolation in a Single Variable

Before considering the most general case, we will design methods for interpolating functions of a single variable $f : \mathbb{R} \to \mathbb{R}$. As input, we will take a set of k pairs (x_i, y_i) with the assumption $f(x_i) = y_i$; our job is to find $f(x)$ for $x \notin \{x_1, \ldots, x_k\}$.

Our strategy in this section and others will take inspiration from linear algebra by writing $f(x)$ in a *basis*. That is, the set of all possible functions $f : \mathbb{R} \to \mathbb{R}$ is far too large to work with and includes many functions that are not practical in a computational setting. Thus, we simplify the search space by forcing f to be written as a linear combination of simpler building block basis functions. This strategy is already familiar from basic calculus: The Taylor expansion writes functions in the basis of polynomials, while Fourier series use sine and cosine.

11.1.1 Polynomial Interpolation

Perhaps the most straightforward interpolant is to assume that $f(x)$ is in $\mathbb{R}[x]$, the set of polynomials. Polynomials are smooth, and it is straightforward to find a degree $k-1$ polynomial through k sample points.

In fact, Example 3.3 already works out the details of such an interpolation technique. As a reminder, suppose we wish to find $f(x) \equiv a_0 + a_1 x + a_2 x^2 + \cdots + a_{k-1}x^{k-1}$; here our unknowns are the values $a_0, \ldots, a_{k-1}$. Plugging in the expression $y_i = f(x_i)$ for each i shows that the vector $\vec{a}$ satisfies the $k \times k$ *Vandermonde* system:

$$\begin{pmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{k-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{k-1} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & x_{k-1} & x_{k-1}^2 & \cdots & x_{k-1}^{k-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{k-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{k-1} \end{pmatrix}$$

Thus, carrying out degree-k polynomial interpolation can be accomplished using a $k \times k$ linear solve by applying our generic strategies from previous chapters, but in fact we can do better.

One way to think about our form for $f(x)$ is that it is written in a basis. Just like a basis for $\mathbb{R}^n$ is a set of n linearly-independent vectors $\vec{v}_1, \ldots, \vec{v}_n$, here the space of polynomials of degree $k-1$ is written in the span of *monomials* $\{1, x, x^2, \ldots, x^{k-1}\}$. It may be the most obvious basis for $\mathbb{R}[x]$, but our current choice has few properties that make it useful for the interpolation problem. One way to see this problem is to plot the sequence of functions $1, x, x^2, x^3, \ldots$ for $x \in [0,1]$; in this interval, it is easy to see that as k gets large, the functions x^k all start looking similar.

Continuing to apply our intuition from linear algebra, we may choose to write our polynomial in a basis that is more suited to the problem at hand. This time, recall that we are given k pairs $(x_1, y_1), \ldots, (x_k, y_k)$. We will use these (fixed) points to define the *Lagrange interpolation* basis $\phi_1, \ldots, \phi_k$ by writing:

$$\phi_i(x) \equiv \frac{\prod_{j \neq i}(x - x_j)}{\prod_{j \neq i}(x_i - x_j)}$$

Although it is not written in the basis $1, x, x^2, \ldots, x^{k-1}$, it is easy to see that each ϕ_i is still a polynomial of degree $k-1$. Furthermore, the Lagrange basis has the following desirable property:

$$\phi_i(x_\ell) = \begin{cases} 1 & \text{when } \ell = i \\ 0 & \text{otherwise.} \end{cases}$$

Thus, finding the unique degree $k-1$ polynomial fitting our (x_i, y_i) pairs is easy in the Lagrange basis:

$$f(x) \equiv \sum_i y_i \phi_i(x)$$

In particular, if we substitute $x = x_j$ we find:

$$\begin{aligned} f(x_j) &= \sum_i y_i \phi_i(x_j) \\ &= y_j \text{ by our expression for } \phi_i(x_\ell) \text{ above.} \end{aligned}$$

Thus, in the Lagrange basis we can write a closed formula for $f(x)$ that does not require solving the Vandermonde system. The drawback, however, is that each $\phi_i(x)$ takes $O(k)$ time to evaluate using the formula above for a given x, so finding $f(x)$ takes $O(n^2)$ time; if we find the coefficients a_i from the Vandermonde system explicitly, however, the evaluation time can be reduced to $O(n)$.

Computation time aside, the Lagrange basis has an additional numerical drawback. Notice that the denominator is the product of a number of terms. If the x_i's are close together, then the product may include many terms close to zero, so we are dividing by a potentially small number. As we have seen this operation can create numerical issues that we wish to avoid.

One basis for polynomials of degree $k-1$ that attempts to compromise between the numerical quality of the monomials and the efficiency of the Lagrange basis is the *Newton* basis, defined as follows:

$$\psi_i(x) = \prod_{j=1}^{i-1}(x - x_j)$$

We define $\psi_1(x) \equiv 1$. Notice that $\psi_i(x)$ is a degree $i-1$ polynomial. By definition of ψ_i, it is clear that $\psi_i(x_\ell) = 0$ for all $\ell < i$. If we wish to write $f(x) = \sum_i c_i \psi_i(x)$ and write out this observation more explicitly, we find:

$$\begin{aligned}
f(x_1) &= c_1\psi_1(x_1) \\
f(x_2) &= c_1\psi_1(x_2) + c_2\psi_2(x_2) \\
f(x_3) &= c_1\psi_1(x_3) + c_2\psi_2(x_3) + c_3\psi_3(x_3) \\
\vdots &\quad \vdots
\end{aligned}$$

In other words, we can solve the following lower triangular system for $\vec{c}$:

$$\begin{pmatrix}
\psi_1(x_1) & 0 & 0 & \cdots & 0 \\
\psi_1(x_2) & \psi_2(x_2) & 0 & \cdots & 0 \\
\psi_1(x_3) & \psi_2(x_3) & \psi_3(x_3) & \cdots & 0 \\
\vdots & \vdots & \vdots & \cdots & \vdots \\
\psi_1(x_k) & \psi_2(x_k) & \psi_3(x_k) & \cdots & \psi_k(x_k)
\end{pmatrix}
\begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_k \end{pmatrix}
=
\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_k \end{pmatrix}$$

This system can be solved in $O(n^2)$ time using forward substitution, rather than the $O(n^3)$ time needed to solve the Vandermonde system.

We now have three strategies of interpolating k data points using a degree $k-1$ polynomial by writing it in the monomial, Lagrange, and Newton bases. All three represent different compromises between numerical quality and speed. An important property, however, is that the resulting interpolated function $f(x)$ is the *same* in each case. More explicitly, there is exactly one polynomial of degree $k-1$ going through a set of k points, so since all our interpolants are degree $k-1$ they must have the same output.

11.1.2 Alternative Bases

Although polynomial functions are particularly amenable to mathematical analysis, there is no fundamental reason why our interpolation basis cannot consist of different types of functions. For example, a crowning result of Fourier analysis implies that a large class of functions are well-approximated by sums of trigonometric functions $\cos(kx)$ and $\sin(kx)$ for $k \in \mathbb{N}$. A construction like the Vandermonde system still applies in this case, and in fact the Fast Fourier Transform algorithm (which merits a larger discussion) shows how to carry out such an interpolation even faster.

A smaller extension of the development in §11.1.1 is to *rational* functions of the form:

$$f(x) = \frac{p_0 + p_1 x + p_2 x^2 + \cdots + p_m x^m}{q_0 + q_1 x + q_2 x^2 + \cdots + q_n x^n}$$

Notice that if we are given k pairs (x_i, y_i), then we will need $m + n + 1 = k$ for this function to be well-defined. One additional degree of freedom must be fixed to account for the fact that the same rational function can be expressed multiple ways by identical scaling of the numerator and the denominator.

Rational functions can have asymptotes and other patterns not achievable using only polynomials, so they can be desirable interpolants for functions that change quickly or have poles. In fact, once m and n are fixed, the coefficients p_i and q_i still can be found using linear techniques by multiplying both sides by the denominator:

$$y_i(q_0 + q_1 x_i + q_2 x_i^2 + \cdots + q_n x_i^n) = p_0 + p_1 x_i + p_2 x_i^2 + \cdots + p_m x_i^m$$

Again, the unknowns in this expression are the p's and q's.

The flexibility of rational functions, however, can cause some issues. For instance, consider the following example:

Example 11.1 (Failure of rational interpolation, Bulirsch-Stoer §2.2)**.** *Suppose we wish to find $f(x)$ with the following data points: $(0,1)$, $(1,2)$, $(2,2)$. We could choose $m = n = 1$. Then, our linear conditions become:*

$$\begin{aligned} q_0 &= p_0 \\ 2(q_0 + q_1) &= p_0 + p_1 \\ 2(q_0 + 2q_1) &= p_0 + 2p_1 \end{aligned}$$

One nontrivial solution to this system is:

$$\begin{aligned} p_0 &= 0 \\ p_1 &= 2 \\ q_0 &= 0 \\ q_1 &= 1 \end{aligned}$$

This implies the following form for $f(x)$:

$$f(x) = \frac{2x}{x}$$

This function has a degeneracy at $x = 0$, and in fact canceling the x in the numerator and denominator does not yield $f(0) = 1$ as we might desire.

This example illustrates a larger phenomenon. Our linear system for finding the p's and q's can run into issues when the resulting denominator $\sum_\ell p_\ell x^\ell$ has a root at any of the fixed x_i's. It can be shown that when this is the case, no rational function exists with the fixed choice of m and n interpolating the given values. A typical partial resolution in this case is presented in (CITE), which increments m and n alternatingly until a nontrivial solution exists. From a practical standpoint, however, the specialized nature of these methods is a good indicator that alternative interpolation strategies may be preferable when the basic rational methods fail.

11.1.3 Piecewise Interpolation

So far, we have constructed our interpolation strategies by combining simple functions on all of $\mathbb{R}$. When the number k of data points becomes high, however, many degeneracies become apparent. For example, Figure NUMBER shows examples in which fitting high-degree polynomials to input data can yield unexpected results. Furthermore, Figure NUMBER illustrates how these strategies are *nonlocal*, meaning that changing any single value y_i in the input data can change the behavior of f for all x, even those that are far away from the corresponding x_i. Somehow this property is unrealistic: We expect only the input data near a given x to affect the value of $f(x)$, especially when there is a large cloud of input points.

For these reasons, when we design a set of basis functions $\phi_1, \ldots, \phi_k$, a desirable property is not only that they are easy to work with but also that they have *compact support*:

Definition 11.1 (Compact support). *A function $g(x)$ has* compact support *if there exists $C \in \mathbb{R}$ such that $g(x) = 0$ for any x with $|x| > C$.*

That is, compactly supported functions only have a finite range of points in which they can take nonzero values.

A common strategy for constructing interpolating bases with compact support is to do so in a *piecewise* fashion. In particular, much of the literature on computer graphics depends on the construction of *piecewise polynomials*, which are defined by breaking $\mathbb{R}$ into a set of intervals and writing a different polynomial in each interval. To do so, we will order our data points so that $x_1 < x_2 < \cdots < x_k$. Then, two simple examples of piecewise interpolants are the following:

- Piecewise constant (Figure NUMBER): For a given x, find the data point x_i minimizing $|x - x_i|$ and define $f(x) = y_i$.

- Piecewise linear (Figure NUMBER): If $x < x_1$ take $f(x) = y_1$, and if $x > x_k$ take $f(x) = y_k$. Otherwise, find an interval with $x \in [x_i, x_{i+1}]$ and define

$$f(x) = y_{i+1} \cdot \frac{x - x_i}{x_{i+1} - x_i} + y_i \cdot \left(1 - \frac{x - x_i}{x_{i+1} - x_i}\right).$$

More generally, we can write a different polynomial in each interval $[x_i, x_{i+1}]$. Notice our pattern so far: Piecewise constant polynomials are discontinuous, while piecewise linear functions are continuous. It is easy to see that piecewise quadratics can be C^1, piecewise cubics can be C^2, and so on. This increased continuity and differentiability occurs even though each y_i has local support; this theory is worked out in detail in constructing "splines," or curves interpolating between points given function values and tangents.

This increased continuity, however, has its own drawbacks. With each additional degree of differentiability, we put a stronger smoothness assumption on f. This assumption can be unrealistic: Many physical phenomena truly are noisy or discontinuous, and this increased smoothness can negatively affect interpolatory results. One domain in which this effect is particularly clear is when interpolation is used in conjunction with physics simulation tools. Simulating turbulent fluid flows with oversmoothed functions can remove discontinuous phenomena like shock waves that are desirable as output.

These issues aside, piecewise polynomials still can be written as linear combinations of basis functions. For instance, the following functions serve as a basis for the piecewise constant functions:

$$\phi_i(x) = \begin{cases} 1 & \text{when } \frac{x_{i-1}+x_i}{2} \leq x < \frac{x_i+x_{i+1}}{2} \\ 0 & \text{otherwise} \end{cases}$$

This basis simply puts the constant 1 near x_i and 0 elsewhere; the piecewise constant interpolation of a set of points (x_i, y_i) is written as $f(x) = \sum_i y_i \phi_i(x)$. Similarly, the so-called "hat" basis shown in Figure NUMBER spans the set of piecewise linear functions with sharp edges at our data points x_i:

$$\psi_i(x) = \begin{cases} \frac{x-x_{i-1}}{x_i-x_{i-1}} & \text{when } x_{i-1} < x \leq x_i \\ \frac{x_{i+1}-x}{x_{i+1}-x_i} & \text{when } x_i < x \leq x_{i+1} \\ 0 & \text{otherwise} \end{cases}$$

Once again, by construction the piecewise linear interpolation of the given data points is $f(x) = \sum_i y_i \psi_i(x)$.

11.1.4 Gaussian Processes and Kriging

Not covered in CS 205A, Fall 2013.

11.2 Multivariable Interpolation

Many extensions of the strategies above exist for interpolating a function given datapoints $(\vec{x}_i, y_i)$ where $\vec{x}_i \in \mathbb{R}^n$ now can be multidimensional. Strategies for interpolation in this case, however, are not quite as clear, however, because it is less obvious to partition $\mathbb{R}^n$ into a small number of regions around the x_i. For this reason, a common pattern is to interpolate using relatively *low-order* functions, that is, to prefer simplistic and efficient interpolation strategies over ones that output C^∞ functions.

If all we are given is the set of inputs and outputs $(\vec{x}_i, y_i)$, then one piecewise constant strategy for interpolation is to use *nearest-neighbor interpolation*. In this case $f(\vec{x})$ simply takes the value y_i corresponding to $\vec{x}_i$ minimizing $\|\vec{x} - \vec{x}_i\|_2$; simple implementations iterate over all i to find this value, although data structures like k-d trees can find nearest neighbors more quickly. Just as piecewise constant interpolations divided $\mathbb{R}$ into intervals about the data points x_i, the nearest-neighbor strategy divides $\mathbb{R}^n$ into a set of *Voronoi cells*:

Definition 11.2 (Voronoi cell)**.** *Given a set of points $S = \{\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_k\} \subseteq \mathbb{R}^n$, the* Voronoi cell *corresponding to a specific $\vec{x}_i$ is the set $V_i \equiv \{\vec{x} : \|\vec{x} - \vec{x}_i\|_2 < \|\vec{x} - \vec{x}_j\|_2 \text{ for all } j \neq i\}$. That is, it is the set of points closer to $\vec{x}_i$ than to any other $\vec{x}_j$ in S.*

Figure NUMBER shows an example of the Voronoi cells about a set of data points in $\mathbb{R}^2$. These cells have many favorable properties; for example, they are convex polygons and are localized about each $\vec{x}_i$. In fact, the connectivity of Voronoi cells is a well-studied problem in computational geometry leading to the construction of the celebrated Delaunay triangulation.

There are many options for continuous interpolation of functions on $\mathbb{R}^n$, each with its own advantages and disadvantages. If we wish to extend our nearest-neighbor strategy above, for example, we could compute multiple nearest neighbors of $\vec{x}$ and interpolate $f(\vec{x})$ based on $\|\vec{x} -$

$\vec{x}_i\|_2$ for each nearest neighbor $\vec{x}_i$. Certain "k-nearest neighbor" data structures can accelerate queries where you want to find multiple points in a dataset closest to a given $\vec{x}$.

Another strategy appearing frequently in the computer graphics literature is *barycentric* interpolation. Suppose we have exactly $n+1$ sample points $(\vec{x}_1, y_1), \ldots, (\vec{x}_{n+1}, y_{n+1})$, where $\vec{x}_i \in \mathbb{R}^n$, and as always we wish to interpolate the y values to all of $\mathbb{R}^n$; for example, on the plane we would be given three values associated with the vertices of a triangle. Any point $\vec{x} \in \mathbb{R}^n$ can be written uniquely as a linear combination $\vec{x} = \sum_{i=1}^{n+1} a_i \vec{x}_i$ with an additional constraint that $\sum_i a_i = 1$; in other words, we write $\vec{x}$ as a weighted average of the points $\vec{x}_i$. Barycentric interpolation in this case simply writes $f(\vec{x}) = \sum_i a_i(\vec{x}) y_i$.

On the plane $\mathbb{R}^2$, barycentric interpolation has a straightforward geometric interpolation involving triangle areas, illustrated in Figure NUMBER. Furthermore, it is easy to check that the resulting interpolated function $f(\vec{x})$ is *affine*, meaning it can be written $f(\vec{x}) = c + \vec{d} \cdot x$ for some $c \in \mathbb{R}$ and $\vec{d} \in \mathbb{R}^n$.

In general, the system of equations we wish to solve for barycentric interpolation at some $\vec{x} \in \mathbb{R}^n$ is:

$$\sum_i a_i \vec{x}_i = \vec{x}$$
$$\sum_i a_i = 1$$

In the absence of degeneracies, this system for $\vec{a}$ is invertible when there are $n+1$ points $\vec{x}_i$. In the presence of more $\vec{x}_i$'s, however, the system for $\vec{a}$ becomes *underdetermined*. This means that there are multiple ways of writing a given $\vec{x}$ as a weighted average of the $\vec{x}_i$'s.

One resolution of this issue is to add more conditions on the vector of averaging weights $\vec{a}$. This strategy results in *generalized barycentric coordinates*, a topic of research in modern mathematics and engineering. Typical constraints on $\vec{a}$ ask that it is smooth as a function on $\mathbb{R}^n$ and nonnegative on the interior of the set of $\vec{x}_i$'s when these points define a polygon or polyhedron. Figure NUMBER shows an example of generalized barycentric coordinates computed from data points on a polygon with more than $n+1$ points.

An alternative resolution of the underdetermined problem for barycentric coordinates relates to the idea of using piecewise functions for interpolation; we will restrict our discussion here to $\vec{x}_i \in \mathbb{R}^2$ for simplicity, although the extensions to higher dimensions are relatively obvious. Many times, we are given not only the set of points $\vec{x}_i$ but also a decomposition of the domain we care about (in this case some subset of $\mathbb{R}^2$) into $n+1$-dimensional objects using those points as vertices. For example, Figure NUMBER shows such a tessellation of a part of $\mathbb{R}^2$ into triangles. Interpolation in this case is straightforward: the interior of each triangle is interpolated using barycentric coordinates.

Example 11.2 (Shading). *In computer graphics, one of the most common representations of a shape is as a set of triangles in a* mesh. *In the* per-vertex *shading model, one color is computed for each vertex on a mesh. Then to render the image to the screen, those per-vertex values are interpolated using barycentric interpolation to the interiors of the triangles. Similar strategies are used for texturing and other common tasks. Figure NUMBER shows an example of this simple shading model. As an aside, one issue specific to computer graphics is the interplay between perspective transformations and interpolation strategies. Barycentric interpolation of color on a 3D surface and then projecting that color to the image plane is not the same as projecting triangles to the image plane and subsequently interpolating colors to the interior of the triangle; thus algorithms in this domain must apply* perspective correction *to account for this mistake.*

Given a set of points in $\mathbb{R}^2$, the problem of triangulation is far from trivial, and algorithms for doing this sort of computation often extend poorly to $\mathbb{R}^n$. Thus, in higher dimensions nearest-neighbor or regression strategies become preferable (see Chapter NUMBER).

Barycentric interpolation leads to a generalization of the piecewise linear hat functions from §11.1.3 illustrated in Figure NUMBER. Recall that our interpolatory output is determined completely by the values y_i at the vertices of the triangles. In fact, we can think of $f(\vec{x})$ as a linear combination $\sum_i y_i \phi_i(\vec{x})$, where each $\phi_i(\vec{x})$ is the piecewise barycentric function obtained by putting a 1 on $\vec{x}_i$ and 0 everywhere else, as in Figure NUMBER. These triangular hat functions form the basis of the "first-order finite elements method," which we will explore in future chapters; specialized constructions using higher-order polynomials are known as "higher-order elements" can can be used to guarantee differentiability along triangle edges.

An alternative and equally important decomposition of the domain of f occurs when the points $\vec{x}_i$ occur on a regular grid in $\mathbb{R}^n$. The following examples illustrate situations when this is the case:

Example 11.3 (Image processing)**.** *As mentioned in the introduction, a typical digital photograph is represented as an $m \times n$ grid of red, green, and blue color intensities. We can think of these values as living on a lattice in $\mathbb{Z} \times \mathbb{Z}$. Suppose we wish to rotate the image by an angle that is not a multiple of $90°$, however. Then, as illustrated in Figure NUMBER, we must look up image values at potentially non-integer positions, requiring the interpolation of colors to $\mathbb{R} \times \mathbb{R}$.*

Example 11.4 (Medical imaging)**.** *The typical output of a magnetic resonance imaging (MRI) device is a $m \times n \times p$ grid of values representing the density of tissue at different points; theoretically the typical model for this function is as $f : \mathbb{R}^3 \to \mathbb{R}$. We can extract the outer surface of a particular organ, showed in Figure NUMBER, by finding the level set $\{\vec{x} : f(\vec{x}) = c\}$ for some c. Finding this level set requires us to extend f to the entire voxel grid to find exactly where it crosses c.*

Grid-based interpolation strategies typically apply the one-dimensional formulae from §11.1.3 one dimension at a time. For example, *bilinear* interpolation schemes in $\mathbb{R}^2$ linearly interpolate one dimension at a time to obtain the output value:

Example 11.5 (Bilinear interpolation)**.** *Suppose f takes on the following values:*

- $f(0,0) = 1$
- $f(0,1) = -3$
- $f(1,0) = 5$
- $f(1,1) = -11$

and that in between f is obtained by bilinear interpolation. To find $f(\frac{1}{4}, \frac{1}{2})$, we first interpolate in x_1 to find:

$$f\left(\frac{1}{4}, 0\right) = \frac{3}{4} f(0,0) + \frac{1}{4} f(1,0) = 2$$
$$f\left(\frac{1}{4}, 1\right) = \frac{3}{4} f(0,1) + \frac{1}{4} f(1,1) = -5$$

Next, we interpolate in x_2:

$$f\left(\frac{1}{4},\frac{1}{2}\right)=\frac{1}{2}f\left(\frac{1}{4},0\right)+\frac{1}{2}f\left(\frac{1}{4},1\right)=-\frac{3}{2}$$

An important property of bilinear interpolation is that we receive the same output interpolating first in x_2 and second in x_1.

Higher-order methods like bicubic and Lanczos interpolation once again use more polynomial terms but are slower to compute. In particular, in the case of interpolating images, bicubic strategies require more data points than the square of function values closest to a point $\vec{x}$; this additional expense can slow down graphical tools for which every lookup in memory incurs additional computation time.

11.3 Theory of Interpolation

So far our treatment of interpolation has been fairly heuristic. While relying on our intuition for what a "reasonable" interpolation for a set of function values for the most part is an acceptable strategy, subtle issues can arise with different interpolation methods that are important to acknowledge.

11.3.1 Linear Algebra of Functions

We began our discussion by posing assorted interpolation strategies as different bases for the set of functions $f:\mathbb{R}\to\mathbb{R}$. This analogy of to vector spaces extends to a complete geometric theory of functions, and in fact early work in the field of *functional analysis* essentially extends the geometry of $\mathbb{R}^n$ to sets of functions. Here we will discuss functions of one variable, although many aspects of the extension to more general functions are easy to carry out.

Just as we can define notions of span and linear combination for functions, for fixed $a,b\in\mathbb{R}$ we can define an *inner product* of functions $f(x)$ and $g(x)$ as follows:

$$\langle f,g\rangle\equiv\int_a^b f(x)g(x)\,dx.$$

Just as the A-inner product of vectors helped us derive the conjugate gradients algorithm and had much in common with the dot product, the functional inner product can be used to define linear algebra methods for dealing with spaces of functions and understanding their span. We also define a norm of a function to be $\|f\|\equiv\sqrt{\langle f,f\rangle}$.

Example 11.6. *Function inner product Take $p_n(x)=x^n$ to be the n-th monomial. Then, for $a=0$ and $b=1$ we have:*

$$\begin{aligned}\langle p_n,p_m\rangle&=\int_0^1 x^n\cdot x^m\,dx\\&=\int_0^1 x^{n+m}\,dx\\&=\frac{1}{n+m+1}\end{aligned}$$

Notice that this shows:

$$\begin{aligned}\left\langle \frac{p_n}{\|p_n\|}, \frac{p_m}{\|p_m\|} \right\rangle &= \frac{\langle p_n, p_m\rangle}{\|p_n\|\|p_m\|} \\ &= \frac{\sqrt{(2n+1)(2m+1)}}{n+m+1}\end{aligned}$$

This value is approximately 1 *when* $n \approx m$ *but* $n \neq m$*, substantiating our earlier claim that the monomials "overlap" considerably on* $[0,1]$.

Given this inner product, we can apply the Gram-Schmidt algorithm to find an orthonormal basis for the set of polynomials. If we take $a = -1$ and $b = 1$, we get the Legendre polynomials, plotted in Figure NUMBER:

$$\begin{aligned}P_0(x) &= 1 \\ P_1(x) &= x \\ P_2(x) &= \frac{1}{2}(3x^2 - 1) \\ P_3(x) &= \frac{1}{2}(5x^3 - 3x) \\ P_4(x) &= \frac{1}{8}(35x^4 - 30x^2 + 3) \\ &\vdots \quad \vdots\end{aligned}$$

These polynomials have many useful properties thanks to their orthogonality. For example, suppose we wish to approximate $f(x)$ with a sum $\sum_i a_i P_i(x)$. If we wish to minimize $\|f - \sum_i a_i P_i\|$ in the functional norm, this is a *least squares* problem! By orthogonality of the Legendre basis for $\mathbb{R}[x]$, a simple extension of our methods for projection shows:

$$a_i = \frac{\langle f, P_i\rangle}{\langle P_i, P_i\rangle}$$

Thus, approximating f using polynomials can be accomplished simply by integrating f against the members of the Legendre basis; in the next chapter we will learn how this integral might be carried out approximately.

Given a positive function $w(x)$, We can define a more general inner product $\langle\cdot,\cdot\rangle_w$ by writing

$$\langle f, g\rangle_w = \int_a^b w(x)f(x)g(x)\,dx.$$

If we take $w(x) = \frac{1}{\sqrt{1-x^2}}$ with $a = -1$ and $b = 1$, then applying Gram-Schmidt yields the *Chebyshev* polynomials:

$$\begin{aligned}T_0(x) &= 1 \\ T_1(x) &= x \\ T_2(x) &= 2x^2 - 1 \\ T_3(x) &= 4x^3 - 3x \\ T_4(x) &= 8x^4 - 8x^2 + 1 \\ &\vdots \quad \vdots\end{aligned}$$

In fact, a surprising identity holds for these polynomials:

$$T_k(x) = \cos(k \arccos(x)).$$

This formula can be checked by explicitly checking it for T_0 and T_1, and then inductively applying the observation:

$$\begin{aligned}
T_{k+1}(x) &= \cos((k+1)\arccos(x)) \\
&= 2x\cos(k\arccos(x)) - \cos((k-1)\arccos(x)) \text{ by the identity} \\
&\qquad\qquad\cos((k+1)\theta) = 2\cos(k\theta)\cos(\theta) - \cos((k-1)\theta) \\
&= 2xT_k(x) - T_{k-1}(x)
\end{aligned}$$

This "three-term recurrence" formula also gives an easy way to generate the Chebyshev polynomials.

As illustrated in Figure NUMBER, thanks to the trigonometric formula for the Chebyshev polynomials it is easy to see that the minima and maxima of T_k oscillate between $+1$ and -1. Furthermore, these extrema are located at $\cos(i\pi/k)$ (the so-called "Chebyshev points") for i from 0 to k; this nice distribution of extrema avoids oscillatory phenomena like that shown in Figure NUMBER when using a finite number of polynomial terms to approximate a function. In fact, more technical treatments of polynomial interpolation recommend placing x_i's for interpolation near Chebyshev points to obtain smooth output.

11.3.2 Approximation via Piecewise Polynomials

Suppose we wish to approximate a function $f(x)$ with a polynomial of degree n on an interval $[a, b]$. Define Δx to be the spacing $b - a$. One measure of error of an approximation is as a function of Δx, which should vanish as $\Delta x \to 0$. Then, if we approximate f with piecewise polynomials, this type of analysis tells us how far apart we should space the polynomials to achieve a desired level of approximation.

For example, suppose we approximate f with a constant $c = f(\frac{a+b}{2})$, as in piecewise constant interpolation. If we assume $|f'(x)| < M$ for all $x \in [a, b]$, we have:

$$\begin{aligned}
\max_{x\in[a,b]} |f(x) - c| &\leq \Delta x \max_{x\in[a,b]} M \text{ by the mean value theorem} \\
&\leq M\Delta x
\end{aligned}$$

Thus, we expect $O(\Delta x)$ error when using piecewise constant interpolation.

Suppose instead we approximate f using piecewise linear interpolation, that is, by taking

$$\tilde{f}(x) = \frac{b-x}{b-a}f(a) + \frac{x-a}{b-a}f(b).$$

By the mean value theorem, we know $\tilde{f}'(x) = f'(\theta)$ for some $\theta \in [a, b]$. Writing the Taylor expansion about θ shows $f(x) = f(\theta) + f'(\theta)(x-\theta) + O(\Delta x^2)$ on $[a, b]$, while we can rewrite our linear approximation as $\tilde{f}(x) = f(\theta) + f'(\theta)(x - \theta)$. Thus, subtracting these two expressions shows that the approximation error of f decreases to $O(\Delta x^2)$. It is not difficult to predict that approximation with a degree n polynomial makes $O(\Delta x^{n+1})$ error, although in practice the quadratic convergence of piecewise linear approximations suffices for most applications.

11.4 Problems

Ideas:

- Horner's method for evaluating polynomials
- Recursive strategy for Newton polynomial coefficients.
- Splines, deCasteljeau
- Check triangle area interpolation of barycentric interpolation

Chapter 12

Numerical Integration and Differentiation

In the previous chapter, we developed tools for filling in reasonable values of a function $f(\vec{x})$ given a sampling of values $(\vec{x}_i, f(\vec{x}_i))$ in the domain of f. Obviously this interpolation problem is useful in itself for completing functions that are known to be continuous or differentiable but whose values only are known at a set of isolated points, but in some cases we then wish to study properties of these functions. In particular, if we wish to apply tools from calculus to f, we must be able to approximate its integrals and derivatives.

In fact, there are many applications in which numerical integration and differentiation play key roles in computation. In the most straightforward instance, some well-known functions are *defined* as integrals. For instance, the "error function" used as the cumulative distribution of a Gaussian or bell curve is written:

$$\operatorname{erf}(x) \equiv \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt$$

Approximations of $\operatorname{erf}(x)$ are needed in many statistical contexts, and one reasonable approach to finding these values is to carry out the integral above numerically.

Other times, numerical approximations of derivatives and integrals are part of a larger system. For example, methods we develop in future chapters for approximating solutions to differential equations will depend strongly on these approximations. Similarly, in computational electrodynamics, *integral equations* solving for an unknown function ϕ given a kernel K and output f appear in the relationship:

$$f(\vec{x}) = \int_{\mathbb{R}^n} K(\vec{x}, \vec{y})\phi(\vec{y})\, d\vec{y}.$$

These types of equations must be solved to estimate electric and magnetic fields, but unless the ϕ and K are very special we cannot hope to find such an integral in closed form, yet alone solve this equation for the unknown function ϕ.

In this chapter, we will develop assorted methods for numerical integration and differentiation given a sampling of function values. These algorithms are usually fairly straightforward approximations, so to compare them we will also develop some strategies that evaluate how well we expect different methods to perform.

12.1 Motivation

It is not hard to formulate simple applications of numerical integration and differentiation given how often the tools of calculus appear in the basic formulae and techniques of physics, statistics, and other fields. Here we suggest a few less obvious places where integration and differentiation appear.

Example 12.1 (Sampling from a distribution). *Suppose we are given a probability distribution $p(t)$ on the interval $[0,1]$; that is, if we randomly sample values according to this distribution, we expect $p(t)$ to be proportional to the number of times we draw a value near t. A common task is to generate random numbers distributed like $p(t)$.*

Rather than develop a specialized method to do so every time we receive a new $p(t)$, it is possible to make a useful observation. We define the cumulative distribution function *of p to be*

$$F(t) = \int_0^t p(x)\,dx.$$

Then, if X is a random number distributed evenly in $[0,1]$, one can show that $F^{-1}(X)$ is distributed like p, where F^{-1} is the inverse of F. Thus, if we can approximate F or F^{-1}, we can generate random numbers according to an arbitrary distribution p; this approximation amounts to integrating p, which may have to be done numerically when the integrals are not known in closed form.

Example 12.2 (Optimization). *Recall that most of our methods for minimizing and finding roots of a function f depended on having not only values $f(\vec{x})$ but also its gradient $\nabla f(\vec{x})$ and even Hessian H_f. We have seen that algorithms like BFGS and Broyden's method build up rough approximations of the derivatives of f during the process of optimization. When f has high frequencies, however, it may be better to approximate ∇f near the current iterate $\vec{x}_k$ rather than using values from potentially far-away points $\vec{x}_\ell$ for $\ell < k$.*

Example 12.3 (Rendering). *The* rendering equation *from ray tracing and other algorithms for high-quality rendering is an integral stating that the light leaving a surface is equal to the integral of the light coming into the surface over all possible incoming directions after it is reflected and diffused; essentially it states that light energy must be conserved before and after light interacts with an object. Algorithms for rendering must approximate this integral to compute the amount of light emitted from a surface reflecting light in a scene.*

Example 12.4 (Image processing). *Suppose we think of an image as a function of two variables $I(x,y)$. Many filters, including Gaussian blurs, can be thought of as* convolutions, *given by*

$$(I * g)(x,y) = \iint I(u,v)g(x-u,y-v)\,du\,dv.$$

For example, to blur an image we could take g to be a Gaussian; in this case $(I * g)(x,y)$ can be thought of as a weighted average of the colors of I near the point (x,y). In practice images are discrete grids of pixels, so this integral must be approximated.

Example 12.5 (Bayes' Rule). *Suppose X and Y are continuously-valued random variables; we can use $P(X)$ and $P(Y)$ to express the probabilities that X and Y take particular values. Sometimes, knowing X may affect our knowledge of Y. For instance, if X is a patient's blood pressure and Y is a patient's weight,* *then knowing a patient has high weight may suggest that they also have high blood pressure. We thus can also write conditional probability distributions $P(X|Y)$ (read "the probability of X given Y") expressing such relationships.*

A foundation of modern probability theory states that $P(X|Y)$ and $P(Y|X)$ are related as follows:

$$P(X|Y) = \frac{P(Y|X)P(X)}{\int P(Y|X)P(X)\,dY}$$

Estimating the integral in the denominator can be a serious problem in machine learning algorithms where the probability distributions take complex forms. Thus, approximate and often randomized integration schemes are needed for algorithms in parameter selection that use this value as part of a larger optimization technique.

12.2 Quadrature

We will begin by considering the problem of numerical integration, or *quadrature*. This problem–in a single variable– can be expressed as, "Given a sampling of n points from some function $f(x)$, find an approximation of $\int_a^b f(x)\,dx$." In the previous section, we presented several situations that boil down to exactly this technique.

There are a few variations of the problem that require slightly different treatment or adaptation:

- The endpoints a and b may be fixed, or we may wish to find a quadrature scheme that efficiently can approximate integrals for many (a, b) pairs.
- We may be able to query $f(x)$ at any x but wish to approximate the integral using relatively few samples, or we may be given a list of precomputed pairs $(x_i, f(x_i))$ and are constrained to using these data points in our approximation.

These considerations should be kept in mind as we design assorted algorithms for the quadrature problem.

12.2.1 Interpolatory Quadrature

Many of the interpolation strategies developed in the previous chapter can be extended to methods for quadrature using a very simple observation. Suppose we write a function $f(x)$ in terms of a set of basis functions $\phi_i(x)$:

$$f(x) = \sum_i a_i\phi_i(x).$$

Then, we can find the integral of f as follows:

$$\begin{aligned}
\int_a^b f(x)\,dx &= \int_a^b \left[\sum_i a_i\phi_i(x)\right] dx \text{ by definition of } f \\
&= \sum_i a_i \left[\int_a^b \phi_i(x)\,dx\right] \\
&= \sum_i c_i a_i \text{ if we make the definition } c_i \equiv \int_a^b \phi_i(x)\,dx
\end{aligned}$$

In other words, integrating f simply involves linearly combining the integrals of the basis functions that make up f.

Example 12.6 (Monomials). *Suppose we write $f(x) = \sum_k a_k x^k$. We know*

$$\int_0^1 x^k \, dx = \frac{1}{k+1},$$

so applying the derivation above we know

$$\int_0^1 f(x) \, dx = \sum_k \frac{a_k}{k+1}.$$

In other words, in our notation above we have defined $c_k = \frac{1}{k+1}$.

Schemes where we integrate a function by interpolating samples and integrating the interpolated function are known as *interpolatory quadrature* rules; nearly all the methods we will present below can be written this way. Of course, we can be presented with a chicken-and-egg problem, if the integral $\int \phi_i(x) \, dx$ itself is not known in closed form. Certain methods in higher-order finite elements deal with this problem by putting extra computational time into making a high-quality numerical approximation of the integral of a single ϕ_i, and then since all the ϕ's have similar form apply change-of-coordinates formulas to write integrals for the remaining basis functions. This canonical integral can be approximated offline using a high-accuracy scheme and then reused.

12.2.2 Quadrature Rules

If we are given a set of $(x_i, f(x_i))$ pairs, our discussion above suggests the following form for a *quadrature rule* for approximating the integral of f on some interval:

$$Q[f] \equiv \sum_i w_i f(x_i).$$

Different weights w_i yield different approximations of the integral, which we hope become increasingly similar as we sample the x_i's more densely.

In fact, even the classical theory of integration suggests that this formula is a reasonable starting point. For example, the *Riemann integral* presented in many introductory calculus classes takes the form:

$$\int_a^b f(x) = \lim_{\Delta x_k \to 0} \sum_k f(\tilde{x}_k)(x_{k+1} - x_k)$$

Here, the interval $[a, b]$ is partitioned into pieces $a = x_1 < x_2 < \cdots < x_n = b$, where $\Delta x_k = x_{k+1} - x_k$ and $\tilde{x}_k$ is any point in $[x_k, x_{k+1}]$. For a fixed set of x_k's before taking the limit, this integral clearly can be written in the $Q[f]$ form above.

From this perspective, the choices of $\{x_i\}$ and $\{w_i\}$ completely determine a strategy for quadrature. There are many ways to determine these values, as we will see in the coming section and as we already have seen for interpolatory quadrature.

Example 12.7 (Method of undetermined coefficients). *Suppose we fix $x_1, \ldots, x_n$ and wish to find a reasonable set of accompanying weights w_i so that $\sum_i w_i f(x_i)$ is a suitable approximation of the integral* *of f. An alternative to the basis function strategy listed above is to use the* method of undetermined *coefficients. In this strategy, we choose n functions $f_1(x), \ldots, f_n(x)$ whose integrals are known, and ask that our quadrature rule recover the integrals of these functions exactly:*

$$\begin{aligned}
\int_a^b f_1(x)\,dx &= w_1 f_1(x_1) + w_2 f_1(x_2) + \cdots + w_n f_1(x_n) \\
\int_a^b f_2(x)\,dx &= w_1 f_2(x_1) + w_2 f_2(x_2) + \cdots + w_n f_2(x_n) \\
&\vdots \qquad \vdots \\
\int_a^b f_n(x)\,dx &= w_1 f_n(x_1) + w_2 f_n(x_2) + \cdots + w_n f_n(x_n)
\end{aligned}$$

This creates an $n \times n$ linear system of equations for the w_i's.

One common choice is to take $f_k(x) = x^{k-1}$, that is, to make sure that the quadrature scheme recovers the integrals of low-order polynomials. We know

$$\int_a^b x^k\,dx = \frac{b^{k+1} - a^{k+1}}{k+1}.$$

Thus, we get the following linear system of equations for the w_i's:

$$\begin{aligned}
w_1 + w_2 + \cdots + w_n &= b - a \\
x_1 w_1 + x_2 w_2 + \cdots + x_n w_n &= \frac{b^2 - a^2}{2} \\
x_1^2 w_1 + x_2^2 w_2 + \cdots + x_n^2 w_n &= \frac{b^3 - a^3}{2} \\
&\vdots \qquad \vdots \\
x_1^{n-1} w_1 + x_2^{n-1} w_2 + \cdots + x_n^{n-1} w_n &= \frac{b^2 - a^2}{2}
\end{aligned}$$

This system is exactly the Vandermonde system discussed in §11.1.1.

12.2.3 Newton-Cotes Quadrature

Quadrature rules when the $x_i' s$ are evenly spaced in $[a, b]$ are known as *Newton-Cotes* quadrature rules. As illustrated in Figure NUMBER, there are two reasonable choices of evenly-spaced samples:

- *Closed* Newton-Cotes quadrature places x_i's at a and b. In particular, for $k \in \{1, \ldots, n\}$ we take
$$x_k \equiv a + \frac{(k-1)(b-a)}{n-1}.$$

- *Open* Newton-Cotes quadrature does not place an x_i at a or b:

$$x_k \equiv a + \frac{k(b-a)}{n+1}.$$

After making this choice, the Newton-Cotes formulae simply apply polynomial interpolation to approximate the integral from a to b; the degree of the polynomial obviously must be $n-1$ to keep the quadrature rule well-defined.

In general, we will keep n relatively small. This way we avoid oscillatory and noise phenomena that occur when fitting high-degree polynomials to a set of data points. As in piecewise polynomial interpolation, we will then chain together small pieces into *composite* rules when integrating over a large interval $[a,b]$.

Closed rules. Closed Newton-Cotes quadrature strategies require $n \geq 2$ to avoid dividing by zero. Two strategies appear often in practice:

- The *trapezoidal* rule is obtained for $n = 2$ (so $x_1 = a$ and $x_2 = b$) by linearly interpolating from $f(a)$ to $f(b)$. It states that

$$\int_a^b f(x)\,dx \approx (b-a)\frac{f(a)+f(b)}{2}.$$

- *Simpson's rule* comes from taking $n = 3$, so we now have

$$\begin{aligned} x_1 &= a \\ x_2 &= \frac{a+b}{2} \\ x_3 &= b \end{aligned}$$

Integrating the parabola that goes through these three points yields

$$\int_a^b f(x)\,dx \approx \frac{b-a}{6}\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right).$$

Open rules. Open rules for quadrature allow the possibility of $n = 1$, giving the simplistic midpoint rule:

$$\int_a^b f(x)\,dx \approx (b-a)f\left(\frac{a+b}{2}\right).$$

Larger values of n yield rules similar to Simpson's rule and the trapezoidal rule.

Composite integration. Generally we might wish to integrate $f(x)$ with more than one, two, or three values x_i. It is obvious how to construct a composite rule out of the midpoint or trapezoidal rules above, as illustrated in Figure NUMBER; simply sum up the values along each interval. For example, if we subdivide $[a,b]$ into k intervals, then we can take $\Delta x \equiv \frac{b-a}{k}$ and $x_i \equiv a + i\Delta x$. Then, the composite midpoint rule is:

$$\int_a^b f(x)\,dx \approx \sum_{i=1}^{k} f\left(\frac{x_{i+1}+x_i}{2}\right)\Delta x$$

Similarly, the composite trapezoid rule is:

$$\begin{aligned}\int_a^b f(x)\,dx &\approx \sum_{i=1}^{k}\left(\frac{f(x_i)+f(x_{i+1})}{2}\right)\Delta x\\ &= \Delta x\left(\frac{1}{2}f(a)+f(x_1)+f(x_2)+\cdots+f(x_{k-1})+\frac{1}{2}f(b)\right)\\ &\qquad\text{by separating the two averaged values of } f \text{ in the first line and re-indexing}\end{aligned}$$

An alternative treatment of the composite midpoint rule is to apply the interpolatory quadrature formula from §12.2.1 to piecewise linear interpolation; similarly, the composite version of the trapezoidal rule comes from piecewise linear interpolation.

The composite version of Simpson's rule, illustrated in Figure NUMBER, chains together three points at a time to make parabolic approximations. Adjacent parabolas meet at even-indexed x_i's and may not share tangents. This summation, which only exists when n is even, becomes:

$$\begin{aligned}\int_a^b f(x)\,dx &\approx \frac{\Delta x}{3}\left[f(a)+2\sum_{i=1}^{n-2-1} f(x_{2i})+4\sum_{i=1}^{n/2} f(x_{2i-1})+f(b)\right]\\ &= \frac{\Delta x}{3}\left[f(a)+4f(x_1)+2f(x_2)+4f(x_3)+2f(x_4)+\cdots+4f(x_{n-1})+f(b)\right]\end{aligned}$$

Accuracy. So far, we have developed a number of quadrature rules that effectively combine the same set of $f(x_i)$'s in different ways to obtain different approximations of the integral of f. Each approximation is based on a different engineering assumption, so it is unclear that any of these rules is better than any other. Thus, we need to develop error estimates characterizing their respective behavior. We will use our Newton-Cotes integrators above to show how such comparisons might be carried out, as presented in CITE.

First, consider the midpoint quadrature rule on a single interval $[a,b]$. Define $c \equiv \frac{1}{2}(a+b)$. The Taylor series of f about c is:

$$f(x) = f(c) + f'(c)(x-c) + \frac{1}{2}f''(c)(x-c)^2 + \frac{1}{6}f'''(c)(x-c)^3 + \frac{1}{24}f''''(c)(x-c)^4 + \cdots$$

Thus, by symmetry about c the odd terms drop out:

$$\int_a^b f(x)\,dx = (b-a)f(c) + \frac{1}{24}f''(c)(b-a)^3 + \frac{1}{1920}f''''(c)(b-a)^5 + \cdots$$

Notice that the first term of this sum exactly the estimate of $\int_a^b f(x)\,dx$ provided by the midpoint rule, so this rule is accurate up to $O(\Delta x^3)$.

Now, plugging a and b into our Taylor series for f about c shows:

$$\begin{aligned}f(a) &= f(c) + f'(c)(a-c) + \frac{1}{2}f''(c)(a-c)^2 + \frac{1}{6}f'''(c)(a-c)^3 + \cdots\\ f(b) &= f(c) + f'(c)(b-c) + \frac{1}{2}f''(c)(b-c)^2 + \frac{1}{6}f'''(c)(b-c)^3 + \cdots\end{aligned}$$

Adding these together and multiplying both sides by $b-a/2$ shows:

$$(b-a)\frac{f(a)+f(b)}{2} = f(c)(b-a) + \frac{1}{4}f''(c)(b-a)((a-c)^2+(b-c)^2) + \cdots$$

The $f'(c)$ term vanishes by definition of c. Notice that the left hand side is the trapezoidal rule integral estimate, and the right hand side agrees with our Taylor series for $\int_a^b f(x)\,dx$ up to the cubic term. In other words, the trapezoidal rule is also $O(\Delta x^3)$ accurate in a single interval.

We pause here to note an initially surprising result: The trapezoidal and midpoint rules have the same order of accuracy! In fact, examining the third-order term shows that the midpoint rule is approximately two times more accurate than the trapezoidal rule. This result seems counterintuitive, since the trapezoidal rule uses a linear approximation while the midpoint rule is constant. As illustrated in Figure NUMBER, however, the midpoint rule actually recovers the integral of linear functions, explaining its extra degree of accuracy.

A similar argument applies to finding an error estimate for Simpson's rule. [WRITE EXPLANATION HERE; OMIT FROM 205A]. In the end we find that Simpson's rule has error like $O(\Delta x^5)$.

An important caveat applies to this sort of analysis. In general, Taylor's theorem only applies when Δx is sufficiently *small*. If samples are far apart, then the drawbacks of polynomial interpolation apply, and oscillatory phenomena as discussed in Section NUMBER can cause unstable results for high-order integration schemes.

Thus, returning to the case when a and b are far apart, we now divide $[a,b]$ into intervals of width Δx and apply any of our quadrature rules inside these intervals. Notice that our total number of intervals is ${}^{b-a}/_{\Delta x}$, so we must multiply our error estimates by ${}^{1}/_{\Delta x}$ in this case. In particular, the following orders of accuracy hold:

- Composite midpoint: $O(\Delta x^2)$
- Composite trapezoid: $O(\Delta x^2)$
- Composite Simpson: $O(\Delta x^4)$

12.2.4 Gaussian Quadrature

In some applications, we can choose the locations x_i at which f is sampled. In this case, we can optimize not only the weights for the quadrature rule but also the locations x_i to get the highest quality. This observation leads to challenging but theoretically appealing quadrature rules.

The details of this technique are outside the scope of our discussion, but we provide one simple path to its derivation. In particular, as in Example 12.7, suppose that we wish to optimize $x_1, \ldots, x_n$ and $w_1, \ldots, w_n$ simultaneously to increase the order of an integration scheme. Now we have $2n$ instead of n knowns, so we can enforce equality for $2n$ examples:

$$\begin{aligned}
\int_a^b f_1(x)\,dx &= w_1 f_1(x_1) + w_2 f_1(x_2) + \cdots + w_n f_1(x_n)\\
\int_a^b f_2(x)\,dx &= w_1 f_2(x_1) + w_2 f_2(x_2) + \cdots + w_n f_2(x_n)\\
\vdots \quad & \quad \vdots\\
\int_a^b f_{2n}(x)\,dx &= w_1 f_n(x_1) + w_2 f_n(x_2) + \cdots + w_n f_n(x_n)
\end{aligned}$$

Now both the x_i's and the w_i's are unknown, so this system of equations is no longer linear. For example, if we wish to optimize these values for polynomials on the interval $[-1,1]$ we would have to solve the following system of polynomials (CITE):

$$\begin{aligned}
w_1 + w_2 &= \int_{-1}^{1} 1\, dx = 2 \\
w_1 x_1 + w_2 x_2 &= \int_{-1}^{1} x\, dx = 0 \\
w_1 x_1^2 + w_2 x_2^2 &= \int_{-1}^{1} x^2\, dx = \frac{2}{3} \\
w_1 x_1^3 + w_2 x_2^3 &= \int_{-1}^{1} x^3\, dx = 0
\end{aligned}$$

It can be the case that systems like this have multiple roots and other degeneracies that depend not only on the choice of f_i's (typically polynomials) but also the interval over which we are approximating an integral. Furthermore, these rules are not *progressive*, in the sense that the set of x_i's for n data points has nothing in common with those for k data points when $k \neq n$, so it is difficult to reuse data to achieve a better estimate. On the other hand, when they are applicable Gaussian quadrature has the highest possible degree for fixed n. The *Kronrod* quadrature rules attempt to avoid this issue by optimizing quadrature with $2n+1$ points while reusing the Gaussian points.

12.2.5 Adaptive Quadrature

As we already have shown, there are certain functions f whose integrals are better approximated with a given quadrature rule than others; for example, the midpoint and trapezoidal rules integrate linear functions with full accuracy while sampling issues and other problems can occur if f oscillates rapidly.

Recall that the Gaussian quadrature rule suggests that the placement of the x_i's can have an effect on the quality of a quadrature scheme. There still is one piece of information we have not used, however: the values $f(x_i)$. After all, these determine the quality of our quadrature scheme.

With this in mind, *adaptive* quadrature strategies examine the current estimate and generate new x_i where the integrand is more complicated. Strategies for adaptive integration often compare the output of multiple quadrature techniques, e.g. trapezoid and midpoint, with the assumption that they agree where sampling of f is sufficient (see Figure NUMBER). If they do not agree with some tolerance on a given interval, an additional sample point is generated and the integral estimates are updated.

ADD MORE DETAIL OR AN EXAMPLE; DISCUSS RECURSIVE ALGORITHM; GANDER AND GAUTSCHI

12.2.6 Multiple Variables

Many times we wish to integrate functions $f(\vec{x})$ where $\vec{x} \in \mathbb{R}^n$. For example, when $n = 2$ we might integrate over a rectangle by computing

$$\int_a^b \int_c^d f(x,y)\, dx\, dy.$$

More generally, as illustrated in Figure NUMBER¡ we might wish to find an integral $\int_\Omega f(\vec{x})\, d\vec{x}$, where Ω is some subset of $\mathbb{R}^n$.

A "curse of dimensionality" makes integration exponentially more difficult as the dimension increases. In particular, the number of samples of f needed to achieve comparable quadrature accuracy for an integral in $\mathbb{R}^k$ increases like $O(n^k)$. This observation may be disheartening but is somewhat reasonable: the more input dimensions for f, the more samples are needed to understand its behavior in all dimensions.

The simplest strategy for integration in $\mathbb{R}^k$ is the integrated integral. For example, if f is a function of two variables, suppose we wish to find $\int_a^b \int_c^d f(x,y)\,dx\,dy$. For fixed y, we can approximate the inner integral over x using a one-dimensional quadrature rule; then, we integrate these values over y using another quadrature rule. Obviously both integration schemes induce some error, so we may need to sample $\vec{x}_i$'s more densely than in one dimension to achieve desired output quality.

Alternatively, just as we subdivided $[a,b]$ into intervals, we can subdivide Ω into triangles and rectangles in 2D, polyhedra or boxes in 3D, and so on and use simple interpolatory quadrature rules in each piece. For instance, one popular option is to integrate the output of barycentric interpolation inside polyhedra, since this integral is known in closed form.

When n is high, however, it is not practical to divide the domain as suggested. In this case, we can use the randomized *Monte Carlo method*. In this case, we simply generate k random points $\vec{x}_i \in \Omega$ with, for example, uniform probability. Averaging the values $f(\vec{x}_i)$ yields an approximation of $\int_\Omega f(\vec{x})\,d\vec{x}$ that converges like $1/\sqrt{k}$ – independent of the dimension of Ω! So, in large dimensions the Monte Carlo estimate is preferable to the deterministic quadrature methods above.

MORE DETAIL ON MONTE CARLO CONVERGENCE AND CHOICE OF DISTRIBUTIONS OVER Ω

12.2.7 Conditioning

So far we have considered the quality of a quadrature method using accuracy values $O(\Delta x^k)$; obviously by this metric a set of quadrature weights with large k is preferable.

Another measure, however, balances out the accuracy measurements obtained using Taylor arguments. In particular, recall that we wrote our quadrature rule as $Q[f] \equiv \sum_i w_i f(x_i)$. Suppose we perturb f to some other $\hat{f}$. Define $\|f-\hat{f}\|_\infty \equiv \max_{x\in[a,b]} |f(x)-\hat{f}(x)|$. Then,

$$\begin{aligned}\frac{|Q[f]-Q[\hat{f}]|}{\|f-\hat{f}\|_\infty} &= \frac{|\sum_i w_i(f(x_i)-\hat{f}(x_i))|}{\|f-\hat{f}\|_\infty}\\ &\leq \frac{\sum_i |w_i||f(x_i)-\hat{f}(x_i)|}{\|f-\hat{f}\|_\infty} \text{ by the triangle inequality}\\ &\leq \|\vec{w}\|_\infty \text{ since } |f(x_i)-\hat{f}(x_i)| \leq \|f-\hat{f}\|_\infty \text{ by definition.}\end{aligned}$$

Thus, the stability or conditioning of a quadrature rule depends on the norm of the set of weights $\vec{w}$.

In general, it is easy to verify that as we increase the order of quadrature accuracy, the conditioning $\|\vec{w}\|$ gets worse because the w_i's take large negative values; this contrasts with the all-positive case, where conditioning is bounded by $b-a$ because $\sum_i w_i = b-a$ for polynomial interpolatory schemes and most low-order methods have only positive coefficients (CHECK). This fact is a reflection of the same intuition that we should not interpolate functions using high-order polynomials. Thus, in practice we usually prefer composite quadrature to high-order methods, that may provide better estimates but can be unstable under numerical perturbation.

12.3 Differentiation

Numerical integration is a relatively stable problem. in that the influence of any single value $f(x)$ on $\int_a^b f(x)\,dx$ shrinks to zero as a and b become far apart. Approximating the derivative of a function $f'(x)$, on the other hand, has no such stability property. From the Fourier analysis perspective, one can show that the integral $\int f(x)$ generally has lower frequencies than f, while differentiating to produce f' amplifies the high frequencies of f, making sampling constraints, conditioning, and stability particularly challenging for approximating f'.

Despite the challenging circumstances, approximations of derivatives usually are relatively easy to compute and can be stable depending on the function at hand. In fact, while developing the secant rule, Broyden's method, and so on we used simple approximations of derivatives and gradients to help guide optimization routines.

Here we will focus on approximating f' for $f : \mathbb{R} \to \mathbb{R}$. Finding gradients and Jacobians often is accomplished by differentiating in one dimension at a time, effectively reducing to the one-dimensional problem we consider here.

12.3.1 Differentiating Basis Functions

The simplest case for differentiation comes for functions that are constructed using interpolation routines. Just as in §12.2.1, if we can write $f(x) = \sum_i a_i \phi_i(x)$ then by linearity we know

$$f'(x) = \sum_i a_i \phi_i'(x).$$

In other words, we can think of the functions ϕ_i' as a *basis* for derivatives of functions written in the ϕ_i basis!

An example of this procedure is shown in Figure NUMBER. This phenomenon often connects different interpolatory schemes. For example, piecewise linear functions have piecewise constant derivatives, polynomial functions have polynomial derivatives of lower degree, and so on; we will return to this structure when we consider discretizations of partial differential equations. In the meantime, it is valuable to know in this case that f' is known with full certainty, although as in Figure NUMBER its derivatives may exhibit undesirable discontinuities.

12.3.2 Finite Differences

A more common case is that we have a function $f(x)$ that we can query but whose derivatives are unknown. This often happens when f takes on a complex form or when a user provides $f(x)$ as a subroutine without analytical information about its structure.

The definition of the derivative suggests a reasonable approach:

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

As we might expect, for a finite $h > 0$ with small $|h|$ the expression in the limit provides a possible value approximating $f'(x)$.

To substantiate this intuition, we can use Taylor series to write:

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

Rearranging this expression shows:

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

Thus, the following *forward difference approximation* of f' has linear convergence:

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

Similarly, flipping the sign of h shows that *backward differences* also have linear convergence:

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

We actually can improve the convergence of our approximation using a trick. By Taylor's theorem we can write:

$$\begin{aligned}
f(x+h) &= f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + \frac{1}{6}f'''(x)h^3 + \cdots \\
f(x-h) &= f(x) - f'(x)h + \frac{1}{2}f''(x)h^2 - \frac{1}{6}f'''(x)h^3 + \cdots \\
\implies f(x+h) - f(x-h) &= 2f'(x)h + \frac{1}{3}f'''(x)h^3 + \cdots \\
\implies \frac{f(x+h) - f(x-h)}{2h} &= f'(x) + O(h^2)
\end{aligned}$$

Thus, this *centered difference* gives an approximation of $f'(x)$ with quadratic convergence; this is the highest order of convergence we can expect to achieve with a divided difference. We can, however, achieve more accuracy by evaluating f at other points, e.g. $x + 2h$, although this approximation is not used much in practice in favor of simply decreasing h.

Constructing estimates of higher-order derivatives can take place by similar constructions. For example, if we add together the Taylor expansions of $f(x+h)$ and $f(x-h)$ we see

$$\begin{aligned}
f(x+h) + f(x-h) &= 2f(x) + f''(x)h^2 + O(h^3) \\
\implies \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} &= f''(x) + O(h^2)
\end{aligned}$$

To predict similar combinations for higher derivatives, one trick is to notice that our second derivative formula can be factored differently:

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

That is, our approximation of the second derivative is a "finite difference of finite differences." One way to interpret this formula is shown in Figure NUMBER. When we compute the forward difference approximation of f' between x and $x+h$, we can think of this slope as living at $x + h/2$; we similarly can use backward differences to place a slope at $x - h/2$. Finding the slope between these values puts the approximation back on x.

One strategy that can improve convergence of the approximations above is *Richardson extrapolation*. As an example of a more general pattern, suppose we wish to use forward differences to approximate f'. Define

$$D(h) \equiv \frac{f(x+h) - f(x)}{h}.$$

Obviously $D(h)$ approaches $f'(x)$ as $h \to 0$. More specifically, however, from our discussion in §12.3.2 we know that $D(h)$ takes the form:

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

Suppose we know $D(h)$ and $D(\alpha h)$ for some $0 < \alpha < 1$. We know:

$$D(\alpha h) = f'(x) + \frac{1}{2}f''(x)\alpha h + O(h^2)$$

We can write these two relationships in a matrix:

$$\begin{pmatrix} 1 & \frac{1}{2}h \\ 1 & \frac{1}{2}\alpha h \end{pmatrix} \begin{pmatrix} f'(x) \\ f''(x) \end{pmatrix} = \begin{pmatrix} D(h) \\ D(\alpha h) \end{pmatrix} + O(h^2)$$

Or equivalently,

$$\begin{pmatrix} f'(x) \\ f''(x) \end{pmatrix} = \begin{pmatrix} 1 & \frac{1}{2}h \\ 1 & \frac{1}{2}\alpha h \end{pmatrix}^{-1} \begin{pmatrix} D(h) \\ D(\alpha h) \end{pmatrix} + O(h^2)$$

That is, we took an $O(h)$ approximation of $f'(x)$ using $D(h)$ and made it into an $O(h^2)$ approximation! This clever technique is a method for *sequence acceleration*, since it improves the order of convergence of the approximation $D(h)$. The same trick is applicable more generally to many other problems by writing an approximation $D(h) = a + bh^n + O(h^m)$ where $m > n$, where a is the quantity we hope to estimate and b is the next term in the Taylor expansion. In fact, Richardson extrapolation even can be applied recursively to make higher and higher order approximations.

12.3.3 Choosing the Step Size

Unlike quadrature, numerical differentiation has a curious property. It appears that any method we choose can be arbitrarily accurate simply by choosing a sufficiently small h. This observation is appealing from the perspective that we can achieve higher-quality approximations without additional computation time. The catch, however, is that we must divide by h and compare more and more similar values $f(x)$ and $f(x+h)$; in finite-precision arithmetic, adding and/or dividing by near-zero values induces numerical issues and instabilities. Thus, there is a range of h values that are not large enough to induce significant discretization error and not small enough to make for numerical problems; Figure NUMBER shows an example for differentiating a simple function in IEEE floating point arithmetic.

12.3.4 Integrated Quantities

Not covered in CS 205A, fall 2013.

12.4 Problems

- Gaussian quadrature – always contains midpoints, strategy using orthogonal polynomials
- Adaptive quadrature
- Applications of Richardson extrapolation elsewhere

Chapter 13

Ordinary Differential Equations

We motivated the problem of interpolation in Chapter 11 by transitioning from *analzying* to *finding* functions. That is, in problems like interpolation and regression, the unknown is a function f, and the job of the algorithm is to fill in missing data.

We continue this discussion by considering similar problems involving filling in function values. Here, our unknown continues to be a function f, but rather than simply guessing missing values we would like to solve more complex design problems. For example, consider the following problems:

- Find f approximating some other function f_0 but satisfying additional criteria (smoothness, continuity, low-frequency, etc.).

- Simulate some dynamical or physical relationship as $f(t)$ where t is time.

- Find f with similar values to f_0 but certain properties in common with a different function g_0.

In each of these cases, our unknown is a function f, but our criteria for success is more involved than "matches a given set of data points."

The theories of ordinary differential equations (ODEs) and partial differential equations (PDEs) study the case where we wish to find a function $f(\vec{x})$ based on information about or relationships between its derivatives. Notice we already have solved a simple version of this problem in our discussion of quadrature: Given $f'(t)$, methods for quadrature provide ways of approximating $f(t)$ using integration.

In this chapter, we will consider the case of an *ordinary* differential equations and in particular *initial value problems*. Here, the unknown is a function $f(t) : \mathbb{R} \to \mathbb{R}^n$, and we given an equation satisfied by f and its derivatives as well as $f(0)$; our goal is to predict $f(t)$ for $t > 0$. We will provide several examples of ODEs appearing in the computer science literature and then will proceed to describe common solution techniques.

As a note, we will use the notation f' to denote the derivative ${}^{df}/_{dt}$ of $f : [0, \infty) \to \mathbb{R}^n$. Our goal will be to find $f(t)$ given relationships between t, $f(t)$, $f'(t)$, $f''(t)$, and so on.

13.1 Motivation

ODEs appear in nearly any part of scientific example, and it is not difficult to encounter practical situations requiring their solution. For instance, the basic laws of physical motion are given by an ODE:

Example 13.1 (Newton's Second Law)**.** *Continuing from §5.1.2, recall that Newton's Second Law of Motion states that $F = ma$, that is, the total force on an object is equal to its mass times its acceleration. If we simulate n particles simultaneously, then we can think of combining all their positions into a vector $\vec{x} \in \mathbb{R}^{3n}$. Similarly, we can write a function $\vec{F}(t, \vec{x}, \vec{x}') \in \mathbb{R}^{3n}$ taking the time, positions of the particles, and their velocities and returning the total force on each particle. This function can take into account interrelationships between particles (e.g. gravitational forces or springs), external effects like wind resistance (which depends on $\vec{x}'$), external forces varying with time t, and so on.*

Then, to find the positions of all the particles as functions of time, we wish to solve the equation $\vec{x}'' = \vec{F}(t, \vec{x}, \vec{x}')/m$. We usually are given the positions and velocities of all the particles at time $t = 0$ as a starting condition.

Example 13.2 (Protein folding)**.** *On a smaller scale, the equations governing motions of molecules also are ordinary differential equations. One particularly challenging case is that of* protein folding, *in which the geometry structure of a protein is predicted by simulating intermolecular forces over time. These forces take many often nonlinear forms that continue to challenge researchers in computational biology.*

Example 13.3 (Gradient descent)**.** *Suppose we are wishing to minimize an energy function $E(\vec{x})$ over all $\vec{x}$. We learned in Chapter 8 that $-\nabla E(\vec{x})$ points in the direction E decreases the most at a given $\vec{x}$, so we did* line search *along that direction from $\vec{x}$ to minimize E locally. An alternative option popular in certain theories is to solve an ODE of the form $\vec{x}' = -\nabla E(\vec{x})$; in other words, think of $\vec{x}$ as a function of time $\vec{x}(t)$ that is attempting to decrease E by walking downhill.*

For example, suppose we wish to solve $A\vec{x} = \vec{b}$ for symmetric positive definite A. We know from §10.1.1 that this is equivalent to minimizing $E(\vec{x}) \equiv \frac{1}{2}\vec{x}^\top A\vec{x} - \vec{b}^\top \vec{x} + c$. Thus, we could attempt to solve the ODE $\vec{x}' = -\nabla f(\vec{x}) = \vec{b} - A\vec{x}$. As $t \to \infty$, we expect $\vec{x}(t)$ to better and better satisfy the linear system.

Example 13.4 (Crowd simulation)**.** *Suppose we are writing video game software requiring realistic simulation of virtual crowds of humans, animals, spaceships, and the like. One strategy for generating plausible motion, illustrated in Figure NUMBER, is to use differential equations. Here, the velocity of a member of the crowd is determined as a function of its environment; for example, in human crowds the proximity of other humans, distance to obstacles, and so on can affect the direction a given agent is moving. These rules can be simple, but in the aggregate their interaction is complex. Stable integrators for differential equations underlie this machinery, since we do not wish to have noticeably unrealistic or unphysical behavior.*

13.2 Theory of ODEs

A full treatment of the theory of ordinary differential equations is outside the scope of our discussion, and we refer the reader to CITE for more details. This aside, we mention some highlights here that will be relevant to our development in future sections.

13.2.1 Basic Notions

The most general ODE initial value problem takes the following form:

$$\begin{gathered}\text{Find } f(t) : \mathbb{R} \to \mathbb{R}^n \\ \text{Satisfying } F[t, f(t), f'(t), f''(t), \ldots, f^{(k)}(t)] = 0 \\ \text{Given } f(0), f'(0), f''(0), \ldots, f^{(k-1)}(0)\end{gathered}$$

Here, F is some relationship between f and all its derivatives; we use $f^{(\ell)}$ to denote the ℓ-th derivative of f. We can think of ODEs as determining *evolution* of f over time t; we know f and its derivatives at time zero and wish to predict it moving forward.

ODEs take many forms even in a single variable. For instance, denote $y = f(t)$ and suppose $y \in \mathbb{R}^1$. Then, examples of ODEs include:

- $y' = 1 + \cos t$: This ODE can be solved by integrating both sides e.g. using quadrature methods
- $y' = ay$: This ODE is linear in y
- $y' = ay + e^t$: This ODE is time and position-dependent
- $y'' + 3y' - y = t$: This ODE involves multiple derivatives of y
- $y'' \sin y = e^{ty'}$: This ODE is nonlinear in y and t.

Obviously the most general ODEs can be challenging to solve. We will restrict most of our discussion to the case of *explicit* ODEs, in which the highest-order derivative can be isolated:

Definition 13.1 (Explicit ODE). *An ODE is* explicit *if can be written in the form*

$$f^{(k)}(t) = F[t, f(t), f'(t), f''(t), \ldots, f^{(k-1)}(t)].$$

For example, an explicit form of Newton's second law is $\vec{x}''(t) = \frac{1}{m}\vec{a}(t, \vec{x}(t), \vec{x}'(t))$.

Surprisingly, generalizing the trick in §5.1.2, in fact any explicit ODE can be converted to a first-order equation $f'(t) = F[t, f(t)]$, where f has multidimensional output. This observation implies that we need no more than one derivative in our treatment of ODE algorithms. To see this relationship, we simply recall that ${}^{d^2y}/_{dt^2} = {}^{d}/_{dt}({}^{dy}/_{dt})$. Thus, we can define an intermediate variable $z \equiv {}^{dy}/_{dt}$, and understand ${}^{d^2y}/_{dt^2}$ as ${}^{dz}/_{dt}$ with the constraint $z = {}^{dy}/_{dt}$. More generally, if we wish to solve the explicit problem

$$f^{(k)}(t) = F[t, f(t), f'(t), f''(t), \ldots, f^{(k-1)}(t)],$$

where $f : \mathbb{R} \to \mathbb{R}^n$, then we define $g(t) : \mathbb{R} \to \mathbb{R}^{kn}$ using the first-order ODE:

$$\frac{d}{dt}\begin{pmatrix} g_1(t) \\ g_2(t) \\ \vdots \\ g_{k-1}(t) \\ g_k(t) \end{pmatrix} = \begin{pmatrix} g_2(t) \\ g_3(t) \\ \vdots \\ g_k(t) \\ F[t, g_1(t), g_2(t), \ldots, g_{k-1}(t)] \end{pmatrix}$$

Here, we denote $g_i(t) : \mathbb{R} \to \mathbb{R}^n$ to contain n components of g. Then, $g_1(t)$ satisfies the original ODE. To see so, we just check that our equation above implies $g_2(t) = g_1'(t), g_3(t) = g_2'(t) = g_1''(t)$, and so on. Thus, making these substitutions shows that the final row encodes the original ODE.

The trick above will simplify our notation, but some care should be taken to understand that this approach does not trivialize computations. In particular, in may cases our function $f(t)$ will only have a single output, but the ODE will be in several derivatives. We replace this case with one derivative and several outputs.

Example 13.5 (ODE expansion). *Suppose we wish to solve $y''' = 3y'' - 2y' + y$ where $y(t) : \mathbb{R} \to \mathbb{R}$. This equation is equivalent to:*

$$\frac{d}{dt}\begin{pmatrix} y \\ z \\ w \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & -2 & 3 \end{pmatrix}\begin{pmatrix} y \\ z \\ w \end{pmatrix}$$

Just as our trick above allows us to consider only first-order ODEs, we can restrict our notation even more to *autonomous* ODEs. These equations are of the form $f'(t) = F[f(t)]$, that is, F no longer depends on t. To do so, we could define

$$g(t) \equiv \begin{pmatrix} f(t) \\ \bar{g}(t) \end{pmatrix}.$$

Then, we can solve the following ODE for g instead:

$$g'(t) = \begin{pmatrix} f'(t) \\ \bar{g}'(t) \end{pmatrix} = \begin{pmatrix} F[f(t), \bar{g}(t)] \\ 1 \end{pmatrix}.$$

In particular, $\bar{g}(t) = t$ assuming we take $\bar{g}(0) = 0$.

It is possible to visualize the behavior of ODEs in many ways, illustrated in Figure NUMBER. For instance, if the unknown $f(t)$ is a function of a single variable, then we can think of $F[f(t)]$ as providing the slope of $f(t)$, as shown in Figure NUMBER. Alternatively, if $f(t)$ has output in $\mathbb{R}^2$, we no longer can visualize the dependence on time t, but we can draw *phase space*, which shows the tangent of $f(t)$ at each $(x, y) \in \mathbb{R}^2$.

13.2.2 Existence and Uniqueness

Before we can proceed to discretizations of the initial value problem, we should briefly acknowledge that not all differential equation problems are solvable. Furthermore, some differential equations admit multiple solutions.

Example 13.6 (Unsolvable ODE). *Consider the equation $y' = 2y/t$, with $y(0) \neq 0$ given; notice we are not dividing by zero because $y(0)$ is prescribed. Rewriting as*

$$\frac{1}{y}\frac{dy}{dt} = \frac{2}{t}$$

and integrating with respect to t on both sides shows:

$$\ln|y| = 2\ln t + c$$

Or equivalently, $y = Ct^2$ for some $C \in \mathbb{R}$. Notice that $y(0) = 0$ in this expression, contradicting our initial conditions. Thus, this ODE has no solution with the given initial conditions.

Example 13.7 (Nonunique solutions)**.** *Now, consider the same ODE with $y(0) = 0$. Consider $y(t)$ given by $y(t) = Ct^2$ for* any $C \in \mathbb{R}$. *Then, $y'(t) = 2Ct$. Thus,*

$$\frac{2y}{t} = \frac{2Ct^2}{t} = 2Ct = y'(t),$$

showing that the ODE is solved by this function regardless of C. Thus, solutions of this problem are nonunique.

Thankfully, there is a rich theory characterizing behavior and stability of solutions to differential equations. Our development in the next chapter will have a stronger set of conditions needed for existence of a solution, but in fact under weak conditions on f it is possible to show that an ODE $f'(t) = F[f(t)]$ has a solution. For instance, one such theorem guarantees local existence of a solution:

Theorem 13.1 (Local existence and uniqueness)**.** *Suppose F is continuous and Lipschitz, that is, $\|F[\vec{y}] - F[\vec{x}]\|_2 \leq L\|\vec{y} - \vec{x}\|_2$ for some L. Then, the ODE $f'(t) = F[f(t)]$ admits exactly one solution for all $t \geq 0$ regardless of initial conditions.*

In our subsequent development, we will assume that the ODE we are attempting to solve satisfies the conditions of such a theorem; this assumption is fairly realistic in that at least locally there would have to be fairly degenerate behavior to break such weak assumptions.

13.2.3 Model Equations

One way to gain intuition for the behavior of ODEs is to examine behavior of solutions to some simple *model equations* that can be solved in closed form. These equations represent linearizations of more practical equations, and thus locally they model the type of behavior we can expect.

We start with ODEs in a single variable. Given our simplifications in §13.2.1, the simplest equation we might expect to work with would be $y' = F[y]$, where $y(t) : \mathbb{R} \to \mathbb{R}$. Taking a linear approximation would yield equations of type $y' = ay + b$. Substituting $\bar{y} \equiv y + {}^{b}/_{a}$ shows: $\bar{y}' = y' = ay + b = a(\bar{y} - {}^{b}/_{a}) + b = a\bar{y}$. Thus, in our model equations the constant b simply induces a shift, and for our phenomenological study in this section we can assume $b = 0$.

By the argument above, we locally can understand behavior of $y' = F[y]$ by studying the linear equation $y' = ay$. In fact, applying standard arguments from calculus shows that

$$y(t) = Ce^{at}.$$

Obviously, there are three cases, illustrated in Figure NUMBER:

1. $a > 0$: In this case solutions get larger and larger; in fact, if $y(t)$ and $\hat{y}(t)$ both satisfy the ODE with slightly different starting conditions, as $t \to \infty$ they diverge.

2. $a = 0$: The system in this case is solved by constants; solutions with different starting points stay the same distance apart.

3. $a < 0$: Then, all solutions of the ODE approach 0 as $t \to \infty$.

We say cases 2 and 3 are *stable*, in the sense that perturbing $y(0)$ slightly yields solutions that get close and close over time; case 1 is *unstable*, since a small mistake in specifying the input parameter $y(0)$ will be amplified as time t advances. Unstable ODEs generate ill-posed computational problems; without careful consideration we cannot expect numerical methods to generate usable solutions in this case, since even theoretical outputs are so sensitive to perturbations of the input. On the other hand, stable problems are well-posed since small mistakes in $y(0)$ get diminished over time.

Advancing to multiple dimensions, we could study the linearized equation

$$\vec{y}' = A\vec{y}.$$

As explained in §5.1.2, if $\vec{y}_1, \cdots, \vec{y}_k$ are eigenvectors of A with eigenvalues $\lambda_1, \ldots, \lambda_k$ and $\vec{y}(0) = c_1\vec{y}_1 + \cdots + c_k\vec{y}_k$, then

$$\vec{y}(t) = c_1 e^{\lambda_1 t}\vec{y}_1 + \cdots + c_k e^{\lambda_k t}\vec{y}_k.$$

In other words, the eigenvalues of A take the place of a in our one-dimensional example. From this result, it is not hard to intuit that a multivariable system is stable exactly when its spectral radius is less than one.

In reality we wish to solve $\vec{y}' = F[\vec{y}]$ for general functions F. Assuming F is differentiable, we can write $F[\vec{y}] \approx F[\vec{y}_0] + J_F(\vec{y}_0)(\vec{y} - \vec{y}_0)$, yielding the model equation above after a shift. Thus, for short periods of time we expect behavior similar to the model equation. Additionally, the conditions in Theorem 13.1 can be viewed as a bound on the behavior of J_F, providing a connection to less localized theories of ODE.

13.3 Time-Stepping Schemes

We now proceed to describe several methods for solving the nonlinear ODE $\vec{y}' = F[\vec{y}]$ for potentially nonlinear functions F. In general, given a "time step" h, our methods will be used to generate estimates of $\vec{y}(t+h)$ given $\vec{y}(t)$. Applying these methods iteratively generates estimates of $\vec{y}_0 \equiv \vec{y}(t)$, $\vec{y}_1 \equiv \vec{y}(t+h)$, $\vec{y}_2 \equiv \vec{y}(t+2h)$, $\vec{y}_3 \equiv \vec{y}(t+3h)$, and so on. Notice that since F has no t dependence the mechanism for generating each additional step is the same as the first, so for the most part we will only need to describe a single step of these methods. We call methods for generating approximations of $\vec{y}(t)$ *integrators*, reflecting the fact that they are integrating out the derivatives in the input equation.

Of key importance to our consideration is the idea of *stability*. Just as ODEs can be stable or unstable, so can discretizations. For example, if h is too large, some schemes will accumulate error at an exponential rate; contrastingly, other methods are stable in that even if h is large the solutions will remain bounded. Stability, however, can compete with *accuracy*; often time stable schemes are bad approximations of $\vec{y}(t)$, even if they are guaranteed not to have wild behavior.

13.3.1 Forward Euler

Our first ODE strategy comes from our construction of the forward differencing scheme in §12.3.2:

$$F[\vec{y}_k] = \vec{y}'(t) = \frac{\vec{y}_{k+1} - \vec{y}_k}{h} + O(h)$$

Solving this relationship for $\vec{y}_{k+1}$ shows

$$\vec{y}_{k+1} = \vec{y}_k + hF[\vec{y}_k] + O(h^2) \approx \vec{y}_k + hF[\vec{y}_k].$$

Thus, the *forward Euler* scheme applies the formula on the right to estimate $\vec{y}_{k+1}$. It is one of the most efficient strategies for time-stepping, since it simply evaluates F and adds a multiple of the result to $\vec{y}_k$. For this reason, we call it an *explicit* method, that is, there is an explicit formula for $\vec{y}_{k+1}$ in terms of $\vec{y}_k$ and F.

Analyzing the accuracy of this method is fairly straightforward. Notice that our approximation of $\vec{y}_{k+1}$ is $O(h^2)$, so each step induces quadratic error. We call this error *localized truncation error* because it is the error induced by a single step; the word "truncation" refers to the fact that we truncated a Taylor series to obtain this formula. Of course, our iterate $\vec{y}_k$ already may be inaccurate thanks to accumulated truncation errors from previous iterations. If we integrate from t_0 to t with $O(1/h)$ steps, then our total error looks like $O(h)$; this estimate represents *global truncation error*, and thus we usually write that the forward Euler scheme is "first-order accurate."

The stability of this method requires somewhat more consideration. In our discussion, we will work out the stability of methods in the one-variable case $y' = ay$, with the intuition that similar statements carry over to multidimensional equations by replacing a with the spectral radius. In this case, we know

$$y_{k+1} = y_k + ahy_k = (1 + ah)y_k.$$

In other words, $y_k = (1 + ah)^k y_0$. Thus, the integrator is stable when $|1 + ah| \leq 1$, since otherwise $|y_k| \to \infty$ exponentially. Assuming $a < 0$ (otherwise the problem is ill-posed), we can simplify:

$$\begin{aligned} |1 + ah| \leq 1 &\iff -1 \leq 1 + ah \leq 1 \\ &\iff -2 \leq ah \leq 0 \\ &\iff 0 \leq h \leq \frac{2}{|a|}, \text{ since } a < 0 \end{aligned}$$

Thus, forward Euler admits a *time step restriction* for stability given by our final condition on h. In other words, the output of forward Euler can explode even when $y' = ay$ is stable if h is not small enough. Figure NUMBER illustrates what happens when this condition is obeyed or violated. In multiple dimensions, we can replace this restriction with an analogous one using the spectral radius of A. For nonlinear ODEs this formula gives a guide for stability at least locally in time; globally h may have to be adjusted if the Jacobian of F becomes worse conditioned.

13.3.2 Backward Euler

Similarly, we could have applied the *backward* differencing scheme at $\vec{y}_{k+1}$ to design an ODE integrator:

$$F[\vec{y}_{k+1}] = \vec{y}'(t) = \frac{\vec{y}_{k+1} - \vec{y}_k}{h} + O(h)$$

Thus, we solve the following potentially nonlinear system of equations for $\vec{y}_{k+1}$:

$$\vec{y}_k = \vec{y}_{k+1} - hF[\vec{y}_{k+1}].$$

Because we have to solve this equation for $\vec{y}_{k+1}$, backward Euler is an *implicit* integrator.

This method is first-order accurate like forward Euler by an identical proof. The stability of this method, however, contrasts considerably with forward Euler. Once again considering the model equation $y' = ay$, we write:

$$y_k = y_{k+1} - hay_{k+1} \implies y_{k+1} = \frac{y_k}{1-ha}.$$

Paralleling our previous argument, backward Euler is stable under the following condition:

$$\begin{aligned}\frac{1}{|1-ha|} \leq 1 &\iff |1-ha| \geq 1\\ &\iff 1-ha \leq -1 \text{ or } 1-ha \geq 1\\ &\iff h \leq \frac{2}{a} \text{ or } h \geq 0, \text{ for } a < 0\end{aligned}$$

Obviously we always take $h \geq 0$, so backward Euler is unconditionally stable.

Of course, even if backward Euler is stable it is not necessarily accurate. If h is too large, $\vec{y}_k$ will approach zero far too quickly. When simulating cloth and other physical materials that require lots of high-frequency detail to be realistic, backward Euler may not be an effective choice. Furthermore, we have to invert $F[\cdot]$ to solve for $\vec{y}_{k+1}$.

Example 13.8 (Backward Euler)**.** *Suppose we wish to solve $\vec{y}' = A\vec{y}$ for $A \in \mathbb{R}^{n\times n}$. Then, to find $\vec{y}_{k+1}$ we solve the following system:*

$$\vec{y}_k = \vec{y}_{k+1} - hA\vec{y}_{k+1} \implies \vec{y}_{k+1} = (I_{n\times n} - hA)^{-1}\vec{y}_k.$$

13.3.3 Trapezoidal Method

Suppose $\vec{y}_k$ is known at time t_k and $\vec{y}_{k+1}$ represents the value at time $t_{k+1} = t_k + h$. Suppose we also know $\vec{y}_{k+1/2}$ halfway in between these two steps. Then, by our derivation of the centered differencing we know:

$$\vec{y}_{k+1} = \vec{y}_k + hF[\vec{y}_{k+1/2}] + O(h^3)$$

From our derivation of the trapezoidal rule:

$$\frac{F[\vec{y}_{k+1}] + F[\vec{y}_k]}{2} = F[\vec{y}_{k+1/2}] + O(h^2)$$

Substituting this relationship yields our first second-order integration scheme, the *trapezoid method* for integrating ODEs:

$$\vec{y}_{k+1} = \vec{y}_k + h\frac{F[\vec{y}_{k+1}] + F[\vec{y}_k]}{2}$$

Like backward Euler, this method is implicit since we must solve this equation for $\vec{y}_{k+1}$.

Once again carrying out stability analysis on $y' = ay$, we find in this case time steps of the trapezoidal method solve

$$y_{k+1} = y_k + \frac{1}{2}ha(y_{k+1} + y_k)$$

In other words,

$$y_k = \left(\frac{1+\frac{1}{2}ha}{1-\frac{1}{2}ha}\right)^k y_0.$$

The method is thus stable when

$$\left|\frac{1+\frac{1}{2}ha}{1-\frac{1}{2}ha}\right| < 1.$$

It is easy to see that this inequality holds whenever $a < 0$ and $h > 0$, showing that the trapezoid method is unconditionally stable.

Despite its higher order of accuracy with maintained stability, the trapezoid method, however, has some drawbacks that make it less popular than backward Euler. In particular, consider the ratio

$$R \equiv \frac{y_{k+1}}{y_k} = \frac{1+\frac{1}{2}ha}{1-\frac{1}{2}ha}$$

When $a < 0$, for large enough h this ratio eventually becomes negative; in fact, as $h \to \infty$, we have $R \to -1$. Thus, as illustrated in Figure NUMBER, if time steps are too large, the trapezoidal method of integration tends to exhibit undesirable oscillatory behavior that is not at all like what we might expect for solutions of $y' = ay$.

13.3.4 Runge-Kutta Methods

A class of integrators can be derived by making the following observation:

$$\begin{aligned}\vec{y}_{k+1} &= \vec{y}_k + \int_{t_k}^{t_k+\Delta t} \vec{y}\,'(t)\,dt \text{ by the Fundamental Theorem of Calculus}\\ &= \vec{y}_k + \int_{t_k}^{t_k+\Delta t} F[\vec{y}(t)]\,dt\end{aligned}$$

Of course, using this formula outright does not work for formulating a method for time-stepping since we do not know $\vec{y}(t)$, but careful application of our quadrature formulae from the previous chapter can generate feasible strategies.

For example, suppose we apply the trapezoidal method for integration. Then, we find:

$$\vec{y}_{k+1} = \vec{y}_k + \frac{h}{2}(F[\vec{y}_k] + F[\vec{y}_{k+1}]) + O(h^3)$$

This is the formula we wrote for the trapezoidal method in §13.3.3.

If we do not wish to solve for $\vec{y}_{k+1}$ implicitly, however, we must find an expression to approximate $F[\vec{y}_{k+1}]$. Using Euler's method, however, we know that $\vec{y}_{k+1} = \vec{y}_k + hF[\vec{y}_k] + O(h^2)$. Making this substitution for $\vec{y}_{k+1}$ does not affect the order of approximation of the trapezoidal time step above, so we can write:

$$\vec{y}_{k+1} = \vec{y}_k + \frac{h}{2}(F[\vec{y}_k] + F[\vec{y}_k + hF[\vec{y}_k]]) + O(h^3)$$

Ignoring the $O(h^3)$ terms yields a new integration strategy known as *Heun's method*, which is second-order accurate and explicit.

If we study stability behavior of Heun's method for $y' = ay$ for $a < 0$, we know:

$$\begin{aligned} y_{k+1} &= y_k + \frac{h}{2}(ay_k + a(y_k + hay_k)) \\ &= \left(1 + \frac{h}{2}a(2 + ha)\right) y_k \\ &= \left(1 + ha + \frac{1}{2}h^2a^2\right) y_k \end{aligned}$$

Thus, the method is stable when

$$\begin{aligned} & -1 \leq 1 + ha + \frac{1}{2}h^2a^2 \leq 1 \\ \iff & -4 \leq 2ha + h^2a^2 \leq 0 \end{aligned}$$

The inequality on the right shows $h \leq \frac{2}{|a|}$, and the one on the left is always true for $h > 0$ and $a < 0$, so the stability condition is $h \leq \frac{2}{|a|}$.

Heun's method is an example of a *Runge-Kutta* method derived by applying quadrature methods to the integral above and substituting Euler steps into $F[\cdot]$. Forward Euler is a first-order accurate Runge-Kutta method, and Heun's method is second-order. A popular fourth-order Runge-Kutta method (abbreviated "RK4") is given by:

$$\begin{aligned} \vec{y}_{k+1} &= \vec{y}_k + \frac{h}{6}(\vec{k}_1 + 2\vec{k}_2 + 2\vec{k}_3 + \vec{k}_4) \\ \text{where } \vec{k}_1 &= F\left[\vec{y}_k\right] \\ \vec{k}_2 &= F\left[\vec{y}_k + \frac{1}{2}h\vec{k}_1\right] \\ \vec{k}_3 &= F\left[\vec{y}_k + \frac{1}{2}h\vec{k}_2\right] \\ \vec{k}_4 &= F\left[\vec{y}_k + h\vec{k}_3\right] \end{aligned}$$

This method can be derived by applying Simpson's rule for quadrature.

Runge-Kutta methods are popular because they are explicit and thus easy to evaluate while providing high degrees of accuracy. The cost of this accuracy, however, is that $F[\cdot]$ must be evaluated more times. Furthermore, Runge-Kutta strategies can be extended to implicit methods that can solve stiff equations.

13.3.5 Exponential Integrators

One class of integrators that achieves strong accuracy when $F[\cdot]$ is approximately linear is to use our solution to the model equation explicitly. In particular, if we were solving the ODE $\vec{y}' = A\vec{y}$, using eigenvectors of A (or any other method) we could find an explicit solution $\vec{y}(t)$ as explained in §13.2.3. We usually write $\vec{y}_{k+1} = e^{Ah}\vec{y}_k$, where e^{Ah} encodes our exponentiation of the eigenvalues (in fact we can find a matrix e^{Ah} from this expression that solves the ODE to time h).

Now, if we write

$$\vec{y}' = A\vec{y} + G[\vec{y}],$$

where G is a nonlinear but small function, we can achieve fairly high accuracy by integrating the A part explicitly and then approximating the nonlinear G part separately. For example, the *first-order exponential integrator* applies forward Euler to the nonlinear G term:

$$\vec{y}_{k+1} = e^{Ah}\vec{y}_k - A^{-1}(1 - e^{Ah})G[\vec{y}_k]$$

Analysis revealing the advantages of this method is more complex than what we have written, but intuitively it is clear that these methods will behave particularly well when G is small.

13.4 Multivalue Methods

The transformations in §13.2.1 enabled us to simplify notation in the previous section considerably by reducing all explicit ODEs to the form $\vec{y}' = F[\vec{y}]$. In fact, while all explicit ODEs *can* be written this way, it is not clear that they always *should*.

In particular, when we reduced k-th order ODEs to first-order ODEs, we introduced a number of variables representing the first through $k-1$-st derivatives of the desired output. In fact, in our final solution we only care about the zeroth derivative, that is, the function itself, so orders of accuracy on the temporary variables are less important.

From this perspective, consider the Taylor series

$$\vec{y}(t_k + h) = \vec{y}(t_k) + h\vec{y}'(t_k) + \frac{h^2}{2}\vec{y}''(t_k) + O(h^3)$$

If we only know $\vec{y}'$ up to $O(h^2)$, this does not affect our approximation, since $\vec{y}'$ gets multiplied by h. Similarly, if we only know $\vec{y}''$ up to $O(h)$, this approximation will not affect the Taylor series terms above because it will get multiplied by $h^2/2$. Thus, we now consider "multivalue" methods, designed to integrate $\vec{y}^{(k)}(t) = F[t, \vec{y}'(t), \vec{y}''(t), \ldots, \vec{y}^{(k-1)}(t)]$ with different-order accuracy for different derivatives of the function $\vec{y}$.

Given the importance of Newton's second law $F = ma$, we will restrict to the case $\vec{y}'' = F[t, \vec{y}, \vec{y}']$; many extensions exist for the less common k-th order case. We introduce a "velocity" vector $\vec{v}(t) = \vec{y}'(t)$ and an "acceleration" vector $\vec{a}$. By our previous reduction, we wish to solve the following first-order system:

$$\begin{aligned}
\vec{y}'(t) &= \vec{v}(t) \\
\vec{v}'(t) &= \vec{a}(t) \\
\vec{a}(t) &= F[t, \vec{y}(t), \vec{v}(t)]
\end{aligned}$$

Our goal is to derive an integrator specifically tailored to this system.

13.4.1 Newmark Schemes

We begin by deriving the famous class of *Newmark* integrators.[1] Denote $\vec{y}_k$, $\vec{v}_k$, and $\vec{a}_k$ as the position, velocity, and acceleration vectors at time t_k; our goal is to advance to time $t_{k+1} \equiv t_k + h$.

[1]We follow the development in `http://www.stanford.edu/group/frg/course_work/AA242B/CA-AA242B-Ch7.pdf`.

Use $\vec{y}(t)$, $\vec{v}(t)$, and $\vec{a}(t)$ to denote the functions of time assuming we start at t_k. Then, obviously we can write

$$\vec{v}_{k+1} = \vec{v}_k + \int_{t_k}^{t_{k+1}} \vec{a}(t)\,dt$$

We also can write $\vec{y}_{k+1}$ as an integral involving $\vec{a}(t)$, by following a few steps:

$$\begin{aligned}
\vec{y}_{k+1} &= \vec{y}_k + \int_{t_k}^{t_{k+1}} \vec{v}(t)\,dt \\
&= \vec{y}_k + [t\vec{v}(t)]_{t_k}^{t_{k+1}} - \int_{t_k}^{t_{k+1}} t\vec{a}(t)\,dt \text{ after integration by parts} \\
&= \vec{y}_k + t_{k+1}\vec{v}_{k+1} - t_k\vec{v}_k - \int_{t_k}^{t_{k+1}} t\vec{a}(t)\,dt \text{ by expanding the difference term} \\
&= \vec{y}_k + h\vec{v}_k + t_{k+1}\vec{v}_{k+1} - t_{k+1}\vec{v}_k - \int_{t_k}^{t_{k+1}} t\vec{a}(t)\,dt \text{ by adding and subtracting } h\vec{v}_k \\
&= \vec{y}_k + h\vec{v}_k + t_{k+1}(\vec{v}_{k+1} - \vec{v}_k) - \int_{t_k}^{t_{k+1}} t\vec{a}(t)\,dt \text{ after factoring} \\
&= \vec{y}_k + h\vec{v}_k + t_{k+1}\int_{t_k}^{t_{k+1}} \vec{a}(t)\,dt - \int_{t_k}^{t_{k+1}} t\vec{a}(t)\,dt \text{ since } \vec{v}'(t) = \vec{a}(t) \\
&= \vec{y}_k + h\vec{v}_k + \int_{t_k}^{t_{k+1}} (t_{k+1} - t)\vec{a}(t)\,dt
\end{aligned}$$

Suppose we choose $\tau \in [t_k, t_{k+1}]$. Then, we can write expressions for $\vec{a}_k$ and $\vec{a}_{k+1}$ using the Taylor series about τ:

$$\begin{aligned}
\vec{a}_k &= \vec{a}(\tau) + \vec{a}'(\tau)(t_k - \tau) + O(h^2) \\
\vec{a}_{k+1} &= \vec{a}(\tau) + \vec{a}'(\tau)(t_{k+1} - \tau) + O(h^2)
\end{aligned}$$

For any constant $\gamma \in \mathbb{R}$, if we scale the first equation by $1 - \gamma$ and the second by γ and sum the results, we find:

$$\begin{aligned}
\vec{a}(\tau) &= (1-\gamma)\vec{a}_k + \gamma\vec{a}_{k+1} + \vec{a}'(\tau)((\gamma - 1)(t_k - \tau) - \gamma(t_{k+1} - \tau)) + O(h^2) \\
&= (1-\gamma)\vec{a}_k + \gamma\vec{a}_{k+1} + \vec{a}'(\tau)(\tau - h\gamma - t_k) + O(h^2) \text{ after substituting } t_{k+1} = t_k + h
\end{aligned}$$

In the end, we wish to integrate $\vec{a}$ from t_k to t_{k+1} to get the change in velocity. Thus, we compute:

$$\begin{aligned}
\int_{t_k}^{t_{k+1}} \vec{a}(\tau)\,d\tau &= (1-\gamma)h\vec{a}_k + \gamma h\vec{a}_{k+1} + \int_{t_k}^{t_{k+1}} \vec{a}'(\tau)(\tau - h\gamma - t_k)\,d\tau + O(h^3) \\
&= (1-\gamma)h\vec{a}_k + \gamma h\vec{a}_{k+1} + O(h^2),
\end{aligned}$$

where the second step holds because the integrand is $O(h)$ and the interval of integration is of width h. In other words, we now know:

$$\vec{v}_{k+1} = \vec{v}_k + (1-\gamma)h\vec{a}_k + \gamma h\vec{a}_{k+1} + O(h^2)$$

To make a similar approximation for $\vec{y}_{k+1}$, we can write

$$\begin{aligned}
\int_{t_k}^{t_{k+1}} (t_{k+1} - t)\vec{a}(t)\,dt &= \int_{t_k}^{t_{k+1}} (t_{k+1} - \tau)((1-\gamma)\vec{a}_k + \gamma\vec{a}_{k+1} + \vec{a}'(\tau)(\tau - h\gamma - t_k))\,d\tau + O(h^3) \\
&= \frac{1}{2}(1-\gamma)h^2\vec{a}_k + \frac{1}{2}\gamma h^2\vec{a}_{k+1} + O(h^2) \text{ by a similar argument.}
\end{aligned}$$

Thus, we can use our earlier relationship to show:

$$\begin{aligned}\vec{y}_{k+1} &= \vec{y}_k + h\vec{v}_k + \int_{t_k}^{t_{k+1}} (t_{k+1} - t)\vec{a}(t)\,dt \text{ from before} \\ &= \vec{y}_k + h\vec{v}_k + \left(\frac{1}{2} - \beta\right) h^2\vec{a}_k + \beta h^2\vec{a}_{k+1} + O(h^2)\end{aligned}$$

Here, we use β instead of γ (and absorb a factor of two in the process) because the γ we choose for approximating $\vec{y}_{k+1}$ does not have to be the same as the one we choose for approximating $\vec{v}_{k+1}$.

After all this integration, we have derived the class of Newmark schemes, with two input parameters γ and β, which has first-order accuracy by the proof above:

$$\begin{aligned}\vec{y}_{k+1} &= \vec{y}_k + h\vec{v}_k + \left(\frac{1}{2} - \beta\right) h^2\vec{a}_k + \beta h^2\vec{a}_{k+1} \\ \vec{v}_{k+1} &= \vec{v}_k + (1-\gamma)h\vec{a}_k + \gamma h\vec{a}_{k+1} \\ \vec{a}_k &= F[t_k, \vec{y}_k, \vec{v}_k]\end{aligned}$$

Different choices of β and γ lead to different schemes. For instance, consider the following examples:

- $\beta = \gamma = 0$ gives the *constant acceleration* integrator:

  $$\begin{aligned}\vec{y}_{k+1} &= \vec{y}_k + h\vec{v}_k + \frac{1}{2}h^2\vec{a}_k \\ \vec{v}_{k+1} &= \vec{v}_k + h\vec{a}_k\end{aligned}$$

  This integrator is explicit and holds exactly when the acceleration is a constant function.

- $\beta = 1/2, \gamma = 1$ gives the *constant implicit acceleration* integrator:

  $$\begin{aligned}\vec{y}_{k+1} &= \vec{y}_k + h\vec{v}_k + \frac{1}{2}h^2\vec{a}_{k+1} \\ \vec{v}_{k+1} &= \vec{v}_k + h\vec{a}_{k+1}\end{aligned}$$

  Here, the velocity is stepped implicitly using backward Euler, giving first-order accuracy. The update of $\vec{y}$, however, can be written

  $$\vec{y}_{k+1} = \vec{y}_k + \frac{1}{2}h(\vec{v}_k + \vec{v}_{k+1}),$$

  showing that it locally is updated by the midpoint rule; this is our first example of a scheme where the velocity and position updates have different orders of accuracy. Even so, it is possible to show that this technique, however, is globally first-order accurate in $\vec{y}$.

- $\beta = 1/4, \gamma = 1/2$ gives the following second-order trapezoidal scheme after some algebra:

  $$\begin{aligned}\vec{x}_{k+1} &= \vec{x}_k + \frac{1}{2}h(\vec{v}_k + \vec{v}_{k+1}) \\ \vec{v}_{k+1} &= \vec{v}_k + \frac{1}{2}h(\vec{a}_k + \vec{a}_{k+1})\end{aligned}$$

- $\beta = 0, \gamma = 1/2$ gives a second-order accurate *central differencing* scheme. In the canonical form, we have

  $$\begin{aligned}\vec{x}_{k+1} &= \vec{x}_k + h\vec{v}_k + \frac{1}{2}h^2\vec{a}_k \\ \vec{v}_{k+1} &= \vec{v}_k + \frac{1}{2}h(\vec{a}_k + \vec{a}_{k+1})\end{aligned}$$

  The method earns its name because simplifying the equations above leads to the alternative form:

  $$\begin{aligned}\vec{v}_{k+1} &= \frac{\vec{y}_{k+2} - \vec{y}_k}{2h} \\ \vec{a}_{k+1} &= \frac{\vec{y}_{k+1} - 2\vec{y}_{k+1} + \vec{y}_k}{h^2}\end{aligned}$$

It is possible to show that Newmark's methods are unconditionally stable when $4\beta > 2\gamma > 1$ and that second-order accuracy occurs exactly when $\gamma = 1/2$ (CHECK).

13.4.2 Staggered Grid

A different way to achieve second-order accuracy in $\vec{y}$ is to use centered differences about the time $t_{k+1/2} \equiv t_k + h/2$:

$$\vec{y}_{k+1} = \vec{y}_k + h\vec{v}_{k+1/2}$$

Rather than use Taylor arguments to try to move $\vec{v}_{k+1/2}$, we can simply store velocities $\vec{v}$ at *half* points on the grid of time steps.

Then, we can use a similar update to step forward the velocities:

$$\vec{v}_{k+3/2} = \vec{v}_{k+1/2} + h\vec{a}_{k+1}.$$

Notice that this update actually is second-order accurate for $\vec{x}$ as well, since if we substitute our expressions for $\vec{v}_{k+1/2}$ and $\vec{v}_{k+3/2}$ we can write:

$$\vec{a}_{k+1} = \frac{1}{h^2}(\vec{y}_{k+2} - 2\vec{y}_{k+1} + \vec{y}_k)$$

Finally, a simple approximation suffices for the acceleration term since it is a higher-order term:

$$\vec{a}_{k+1} = F\left[t_{k+1}, \vec{x}_{k+1}, \frac{1}{2}(\vec{v}_{k+1/2} + \vec{v}_{k+3/2})\right]$$

This expression can be substituted into the expression for $\vec{v}_{k+3/2}$.

When $F[\cdot]$ has no dependence on $\vec{v}$, the method is fully explicit:

$$\begin{aligned}\vec{y}_{k+1} &= \vec{y}_k + h\vec{v}_{k+1/2} \\ \vec{a}_{k+1} &= F[t_{k+1}, \vec{y}_{k+1}] \\ \vec{v}_{k+3/2} &= \vec{v}_{k+1/2} + h\vec{a}_{k+1}\end{aligned}$$

This is known as the *leapfrog* method of integration, thanks to the staggered grid of times and the fact that each midpoint is used to update the next velocity or position.

Otherwise, if the velocity update has dependence on $\vec{v}$ then the method becomes implicit. Often times, dependence on velocity is symmetric; for instance, wind resistance simply changes sign if you reverse the direction you are moving. This property can lead to symmetric matrices in the implicit step for updating velocities, making it possible to use conjugate gradients and related fast iterative methods to solve.

13.5 To Do

- Define stiff ODE
- Give table of time stepping methods for $F[t;\vec{y}]$
- Use $\vec{y}$ notation more consistently

13.6 Problems

- TVD RK
- Multistep/multivalue methods a la Heath
- Verlet integration
- Symplectic integrators

Chapter 14

Partial Differential Equations

Our intuition for ordinary differential equations generally stems from the time evolution of physical systems. Equations like Newton's second law determining the motion of physical objects over time dominate the literature on such initial value problems; additional examples come from chemical concentrations reacting over time, populations of predators and prey interacting from season to season, and so on. In each case, the initial configuration—e.g. the positions and velocities of particles in a system at time zero—are known, and the task is to predict behavior as time progresses.

In this chapter, however, we entertain the possibility of *coupling* relationships between different derivatives of a function. It is not difficult to find examples where this coupling is necessary. For instance, when simulating smoke or gases quantities like "pressure gradients," the derivative of the pressure of a gas in *space*, figure into how the gas moves over *time*; this structure is reasonable since gas naturally diffuses from high-pressure regions to low-pressure regions. In image processing, derivatives couple even more naturally, since measurements about images tend to occur in the x and y directions simultaneously.

Equations coupling together derivatives of functions are known as *partial differential equations*. They are the subject of a rich but strongly nuanced theory worthy of larger-scale treatment, so our goal here will be to summarize key ideas and provide sufficient material to solve problems commonly appearing in practice.

14.1 Motivation

Partial differential equations (PDEs) arise when the unknown is some function $f : \mathbb{R}^n \to \mathbb{R}^m$. We are given one or more relationship between the partial derivatives of f, and the goal is to find an f that satisfies the criteria. PDEs appear in nearly any branch of applied mathematics, and we list just a few below.

As an aside, before introducing specific PDEs we should introduce some notation. In particular, there are a few combinations of partial derivatives that appear often in the world of PDEs. If $f : \mathbb{R}^3 \to \mathbb{R}$ and $\vec{v} : \mathbb{R}^3 \to \mathbb{R}^3$, then the following operators are worth remembering:

$$\text{Gradient: } \nabla f \equiv \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial x_3}\right)$$

$$\begin{aligned}
\text{Divergence: } \nabla\cdot\vec{v} &\equiv \frac{\partial v_1}{\partial x_1}+\frac{\partial v_2}{\partial x_2}+\frac{\partial v_3}{\partial x_3}\\
\text{Curl: } \nabla\times\vec{v} &\equiv \left(\frac{\partial v_3}{\partial x_2}-\frac{\partial v_2}{\partial x_3},\frac{\partial v_1}{\partial x_3}-\frac{\partial v_3}{\partial x_1},\frac{\partial v_2}{\partial x_1}-\frac{\partial v_1}{\partial x_2}\right)\\
\text{Laplacian: } \nabla^2 f &\equiv \frac{\partial^2 f}{\partial x_1^2}+\frac{\partial^2 f}{\partial x_2^2}+\frac{\partial^2 f}{\partial x_3^2}
\end{aligned}$$

Example 14.1 (Fluid simulation). *The flow of fluids like water and smoke is governed by the* Navier-Stokes equations, *a system of PDEs in many variables. In particular, suppose a fluid is moving in some region $\Omega \subseteq \mathbb{R}^3$. We define the following variables, illustrated in Figure NUMBER:*

- $t \in [0,\infty)$: *Time*
- $\vec{v}(t):\Omega\to\mathbb{R}^3$: *The velocity of the fluid*
- $\rho(t):\Omega\to\mathbb{R}$: *The density of the fluid*
- $p(t):\Omega\to\mathbb{R}$: *The pressure of the fluid*
- $\vec{f}(t):\Omega\to\mathbb{R}^3$: *External forces like gravity on the fluid*

If the fluid has viscosity μ, then if we assume it is incompressible *the Navier-Stokes equations state:*

$$\rho\left(\frac{\partial\vec{v}}{\partial t}+\vec{v}\cdot\nabla\vec{v}\right)=-\nabla p+\mu\nabla^2\vec{v}+\vec{f}$$

Here, $\nabla^2\vec{v}={}^{\partial^2 v_1}/_{\partial x_1^2}+{}^{\partial^2 v_2}/_{\partial x_2^2}+{}^{\partial^2 v_3}/_{\partial x_3^2}$; we think of the gradient ∇ as a gradient in space rather than time, i.e. $\nabla f=(\frac{\partial f}{\partial x_1},\frac{\partial f}{\partial x_2},\frac{\partial f}{\partial x_3})$. This system of equations determines the time dynamics of fluid motion and actually can be constructed by applying Newton's second law to tracking "particles" of fluid. Its statement, however, involves not only derivatives in time $\frac{\partial}{\partial t}$ but also derivatives in space ∇, making it a PDE.

Example 14.2 (Maxwell's equations). *Maxwell's equations determine the interaction of electric fields $\vec{E}$ and magnetic fields $\vec{B}$ over time. As with the Navier-Stokes equations, we think of the gradient, divergence, and curl as taking partial derivatives in space (and not time t). Then, Maxwell's system (in "strong" form) can be written:*

$$\begin{aligned}
\textit{Gauss's law for electric fields: } \nabla\cdot\vec{E} &= \frac{\rho}{\varepsilon_0}\\
\textit{Gauss's law for magnetism: } \nabla\cdot\vec{B} &= 0\\
\textit{Faraday's law: } \nabla\times\vec{E} &= -\frac{\partial\vec{B}}{\partial t}\\
\textit{Ampère's law: } \nabla\times\vec{B} &= \mu_0\left(\vec{J}+\varepsilon_0\frac{\partial\vec{E}}{\partial t}\right)
\end{aligned}$$

Here, ε_0 and μ_0 are physical constants and $\vec{J}$ encodes the density of electrical current. Just like the Navier-Stokes equations, Maxwell's equations related derivatives of physical quantities in time t to their derivatives in space (given by curl and divergence terms).

Example 14.3 (Laplace's equation)**.** *Suppose Ω is a domain in $\mathbb{R}^2$ with boundary $\partial\Omega$ and that we are given a function $g : \partial\Omega \to \mathbb{R}$, illustrated in Figure NUMBER. We may wish to interpolate g to the interior of Ω. When Ω is an irregular shape, however, our strategies for interpolation from Chapter 11 can break down.*

Suppose we define $f(\vec{x}) : \Omega \to \mathbb{R}$ to be the interpolating function. Then, one strategy inspired by our approach to least-squares is to define an energy functional:

$$E[f] = \int_\Omega \|\nabla f(\vec{x})\|_2^2 \, d\vec{x}$$

That is, $E[f]$ measures the "total derivative" of f measured by taking the norm of its gradient and integrating this quantity over all of Ω. Wildly fluctuating functions f will have high values of $E[f]$ since the slope ∇f will be large in many places; smooth and low-frequency functions f, on the other hand, will have small $E[f]$ since their slope will be small everywhere.[1] *Then, we could ask that f interpolates g while being as smooth as possible in the interior of Ω using the following optimization:*

$$\begin{aligned} &\text{minimize}_f \; E[f] \\ &\text{such that } f(\vec{x}) = g(\vec{x}) \, \forall x \in \partial\Omega \end{aligned}$$

This setup looks *like optimizations we have solved in previous examples, but now our unknown is a function f rather than a point in $\mathbb{R}^n$!*

If f minimizes E, then $E[f+h] \geq E[f]$ for all functions $h(\vec{x})$. This statement is true even for small perturbations $E[f+\varepsilon h]$ as $\varepsilon \to 0$. Dividing by ε and taking the limit as $\varepsilon \to 0$, we must have $\frac{d}{d\varepsilon}E[f+\varepsilon h]|_{\varepsilon=0} = 0$; this is just like setting directional derivatives of a function equal to zero to find its minima. We can simplify:

$$\begin{aligned} E[f+\varepsilon h] &= \int_\Omega \|\nabla f(\vec{x}) + \varepsilon \nabla h(\vec{x})\|_2^2 \, d\vec{x} \\ &= \int_\Omega (\|\nabla f(\vec{x})\|_2^2 + 2\varepsilon \nabla f(\vec{x}) \cdot \nabla h(\vec{x}) + \varepsilon^2 \|\nabla h(\vec{x})\|_2^2) \, d\vec{x} \end{aligned}$$

Differentiating shows:

$$\begin{aligned} \frac{d}{d\varepsilon}E[f+\varepsilon h] &= \int_\Omega (2\nabla f(\vec{x}) \cdot \nabla h(\vec{x}) + 2\varepsilon \|\nabla h(\vec{x})\|_2^2) \, d\vec{x} \\ \frac{d}{d\varepsilon}E[f+\varepsilon h]|_{\varepsilon=0} &= 2\int_\Omega [\nabla f(\vec{x}) \cdot \nabla h(\vec{x})] \, d\vec{x} \end{aligned}$$

This derivative must equal zero for all h, so in particular we can choose $h(\vec{x}) = 0$ for all $\vec{x} \in \partial\Omega$. Then, applying integration by parts, we have:

$$\frac{d}{d\varepsilon}E[f+\varepsilon h]|_{\varepsilon=0} = -2\int_\Omega h(\vec{x}) \nabla^2 f(\vec{x}) \, d\vec{x}$$

This expression must equal zero for all *(all!) perturbations h, so we must have $\nabla^2 f(\vec{x}) = 0$ for all $\vec{x} \in \Omega \backslash \partial\Omega$ (a formal proof is outside of the scope of our discussion). That is, the interpolation problem above*

[1]The notation $E[\cdot]$ used here does not stand for "expectation" as it might in probability theory, but rather simply is an "energy" functional; it is standard notation in areas of functional analysis.

can be solved using the following PDE:

$$\begin{aligned}\nabla^2 f(\vec{x}) &= 0\\ f(\vec{x}) &= g(\vec{x})\,\forall\vec{x}\in\partial\Omega\end{aligned}$$

This equation is known as Laplace's equation*, and it can be solved using sparse positive definite linear methods like what we covered in Chapter 10. As we have seen, it can be applied to interpolation problems for irregular domains Ω; furthermore, $E[f]$ can be augmented to measure other properties of f, e.g. how well f approximates some noisy function f_0, to derive related PDEs by paralleling the argument above.*

Example 14.4 (Eikonal equation)**.** *Suppose $\Omega\subseteq\mathbb{R}^n$ is some closed region of space. Then, we could take $d(\vec{x})$ to be a function measuring the distance from some point $\vec{x}_0$ to $\vec{x}$ completely within Ω. When Ω is convex, we can write d in closed form:*

$$d(\vec{x}) = \|\vec{x}-\vec{x}_0\|_2.$$

As illustrated in Figure NUMBER, however, if Ω is non-convex or a more complicated domain like a surface, distances become more complicated to compute. In this case, distance functions d satisfy the localized condition known as the eikonal equation*:*

$$\|\nabla d\|_2 = 1.$$

If we can compute it, d can be used for tasks like planning paths of robots by minimizing the distance they have to travel with the constraint that they only can move in Ω.

Specialized algorithms known as fast marching methods *are used to find estimates of d given $\vec{x}_0$ and Ω by integrating the eikonal equation. This equation is nonlinear in the derivative ∇d, so integration methods for this equation are somewhat specialized, and proof of their effectiveness is complex. Interestingly but unsurprisingly, many algorithms for solving the eikonal equation have structure similar to Dijkstra's algorithm for computing shortest paths along graphs.*

Example 14.5 (Harmonic analysis)**.** *Different objects respond differently to vibrations, and in large part these responses are functions of the* geometry *of the objects. For example, cellos and pianos can play the same note, but even an inexperienced musician easily can distinguish between the sounds they make. From a mathematical standpoint, we can take $\Omega\subseteq\mathbb{R}^3$ to be a shape represented either as a surface or a volume. If we clamp the edges of the shape, then its* frequency spectrum *is given by solutions of the following differential eigenvalue problem:*

$$\begin{aligned}\nabla^2 f &= \lambda f\\ f(x) &= 0\,\forall x\in\partial\Omega,\end{aligned}$$

where ∇^2 is the Laplacian of Ω and $\partial\Omega$ is the boundary of Ω. Figure NUMBER shows examples of these functions on different domains Ω.

It is easy to check that $\sin kx$ *solves this problem when Ω is the interval $[0,2\pi]$, for $k\in\mathbb{Z}$. In particular, the Laplacian in one dimension is $\partial^2/\partial x^2$, and thus we can check:*

$$\begin{aligned}\frac{\partial^2}{\partial x^2}\sin kx &= \frac{\partial}{\partial x}k\cos kx\\ &= -k^2\sin kx\\ \sin k\cdot 0 &= 0\\ \sin k\cdot 2\pi &= 0\end{aligned}$$

Thus, the eigenfunctions are $\sin kx$ *with eigenvalues $-k^2$.*

14.2 Basic definitions

Using the notation of CITE, we will assume that our unknown is some function $f : \mathbb{R}^n \to \mathbb{R}$. For equations of up to three variables, we will use subscript notation to denote partial derivatives:

$$
\begin{aligned}
f_x &\equiv \frac{\partial f}{\partial x},\\
f_y &\equiv \frac{\partial f}{\partial y},\\
f_{xy} &\equiv \frac{\partial^2 f}{\partial x \partial y},
\end{aligned}
$$

and so on.

Partial derivatives usually are stated as relationships between two or more derivatives of f, as in the following:

- Linear, homogeneous: $f_{xx} + f_{xy} - f_y = 0$
- Linear: $f_{xx} - y f_{yy} + f = xy^2$
- Nonlinear: $f_{xx}^2 = f_{xy}$

Generally, we really wish to find $f : \Omega \to \mathbb{R}$ for some $\Omega \subseteq \mathbb{R}^n$. Just as ODEs were stated as initial value problems, we will state most PDEs as *boundary value problems*. That is, our job will be to fill in f in the interior of Ω given values on its boundary. In fact, we can think of the ODE initial value problem this way: the domain is $\Omega = [0, \infty)$, with boundary $\partial\Omega = \{0\}$, which is where we provide input data. Figure NUMBER illustrates more complex examples. Boundary conditions for these problems take many forms:

- *Dirichlet conditions* simply specify the value of $f(\vec{x})$ on $\partial\Omega$
- *Neumann conditions* specify the derivatives of $f(\vec{x})$ on $\partial\Omega$
- *Mixed* or *Robin conditions* combine these two

14.3 Model Equations

Recall from the previous chapter that we were able to understand many properties of ODEs by examining a *model equation* $y' = ay$. We can attempt to pursue a similar approach for PDEs, although we will find that the story is more nuanced when derivatives are linked together.

As with the model equation for ODEs, we will study the single-variable *linear* case. We also will restrict ourselves to *second-order* systems, that is, systems containing at most the second derivative of u; the model ODE was first-order but here we need at least two orders to study how derivatives interact in a nontrivial way.

A linear second-order PDE has the following general form:

$$\sum_{ij} a_{ij} \frac{\partial f}{\partial x_i \partial x_j} + \sum_i b_i \frac{\partial f}{\partial x_i} + c = 0$$

Formally, we might define the "gradient operator" as:

$$\nabla \equiv \left(\frac{\partial}{\partial x_1}, \frac{\partial}{\partial x_2}, \ldots, \frac{\partial}{\partial x_n}\right).$$

You should check that this operator is reasonable notation in that expressions like ∇f, $\nabla \cdot \vec{v}$, and $\nabla \times \vec{v}$ all provide the proper expressions. In this notation, the PDE can be thought of as taking a matrix form:

$$(\nabla^\top A \nabla + \nabla \cdot \vec{b} + c) f = 0.$$

This form has much in common with our study of quadratic forms in conjugate gradients, and in fact we usually characterize PDEs by the structure of A:

- If A is *positive or negative definite*, system is *elliptic*.
- If A is *positive or negative semidefinite*, the system is *parabolic*.
- If A has only one eigenvalue of different sign from the rest, the system is *hyperbolic*.
- If A satisfies none of the criteria, the system is *ultrahyperbolic.*

These criteria are listed approximately in order of the difficulty level of solving each type of equation. We consider the first three cases below and provide examples of corresponding behavior; ultrahyperbolic equations do not appear as often in practice and require highly specialized techniques for their solution.

TODO: Reduction to canonical form via eigenstuff of A (not in 205A)

14.3.1 Elliptic PDEs

Just as positive definite matrices allow for specialized algorithms like Cholesky decomposition and conjugate gradients that simplify their inversion, elliptic PDEs have particularly strong structure that leads to effective solution techniques.

The model elliptic PDE is the *Laplace equation*, given by $\nabla^2 f = g$ for some given function g as in Example 14.3. For instance, in two variables the Laplace equation becomes

$$f_{xx} + f_{yy} = g.$$

Figure NUMBER illustrates some solutions of the Laplace equation for different choices of u and f on a two-dimensional domain.

Elliptic equations have many important properties. Of particular theoretical and practical importance is the idea of *elliptic regularity*, that solutions of elliptic PDEs automatically are smooth functions in $C^\infty(\Omega)$. This property is not immediately obvious: a second-order PDE in f only requires that f be twice-differentiable to make sense, but in fact under weak conditions they automatically are infinitely differentiable. This property lends to the physical intuition that elliptic equations represent stable physical equilbria like the rest pose of a stretched out rubber sheet. Second-order elliptic equations in the form above also are guaranteed to admit solutions, unlike PDEs in some other forms.

Example 14.6 (Poisson in one variable). *The Laplace equation with $g = 0$ is given the special name* Poisson's equation. *In one variable, it can be written $f''(x) = 0$, which trivially is solved by $f(x) = \alpha x + \beta$. This equation is sufficient to examine possible boundary conditions on $[a, b]$:*

- *Dirichlet conditions for this equation simply specify $f(a)$ and $f(b)$; there is obviously a unique line that goes through $(a, f(a))$ and $(b, f(b))$, which provides the solution to the equation.*

- *Neumann conditions would specify $f'(a)$ and $f'(b)$. But, $f'(a) = f'(b) = \alpha$ for $f(x) = \alpha x + \beta$. In this way, boundary values for Neumann problems can be subject to* compatibility conditions *needed to admit admit a solution. Furthermore, the choice of β does not affect the boundary conditions, so when they are satisfied the solution is not unique.*

14.3.2 Parabolic PDEs

Continuing to parallel the structure of linear algebra, positive *semi*definite systems of equations are only slightly more difficult to deal with than positive definite ones. In particular, positive semidefinite matrices admit a null space which must be dealt with carefully, but in the remaining directions the matrices behave the same as the definite case.

The heat equation is the model parabolic PDE. Suppose $f(0; x, y)$ is a distribution of heat in some region $\Omega \subseteq \mathbb{R}^2$ at time $t = 0$. Then, the heat equation determines how the heat diffuses over time t as a function $f(t; x, y)$:

$$\frac{\partial f}{\partial t} = \alpha \nabla^2 f,$$

where $\alpha > 0$ and we once again think of ∇^2 as the Laplacian in the *space variables* x and y, that is, $\nabla^2 = \partial^2/\partial x^2 + \partial^2/\partial y^2$. This equation must be parabolic, since there is the same coefficient α in front of f_{xx} and f_{yy}, but f_{tt} does not figure into the equation.

Figure NUMBER illustrates a phenomenological interpretation of the heat equation. We can think of $\nabla^2 f$ as measuring the convexity of f, as in Figure NUMBER(a). Thus, the heat equation increases u with time when its value is "cupped" upward, and decreases f otherwise. This negative feedback is stable and leads to equilibrium as $t \to \infty$.

There are two boundary conditions needed for the heat equation, both of which come with straightforward physical interpretations:

- The distribution of heat $f(0; x, y)$ at time $t = 0$ at all points $(x, y) \in \Omega$

- Behavior of f when $t > 0$ at points $(x, y) \in \partial\Omega$. These boundary conditions describe behavior at the boundary of the domain. Dirichlet conditions here provide $f(t; x, y)$ for all $t \geq 0$ and $(x, y) \in \partial\Omega$, corresponding to the situation in which an outside agent fixes the temperatures at the boundary of the domain. These conditions might occur if Ω is a piece of foil sitting next to a heat source whose temperature is not significantly affected by the foil like a large refrigerator or oven. Contrastingly, Neumann conditions specify the derivative of f in the direction normal to the boundary $\partial\Omega$, as in Figure NUMBER; they correspond to fixing the *flux* of heat out of Ω caused by different types of insulation.

14.3.3 Hyperbolic PDEs

The final model equation is the wave equation, corresponding to the indefinite matrix case:

$$\frac{\partial^2 f}{\partial t^2} - c^2 \nabla^2 f = 0$$

The wave equation is hyperbolic because the second derivative in time has opposite sign from the two spatial derivatives. This equation determines the motion of waves across an elastic medium like a rubber sheet; for example, it can be derived by applying Newton's second law to points on a piece of elastic, where x and y are positions on the sheet and $f(t;x,y)$ is the height of the piece of elastic at time t.

Figure NUMBER illustrates a one-dimensional solution of the wave equation. Wave behavior contrasts considerably with heat diffusion in that as $t \to \infty$ energy may not diffuse. In particular, waves can bounce back and forth across a domain indefinitely. For this reason, we will see that implicit integration strategies may not be appropriate for integrating hyperbolic PDEs because they tend to damp out motion.

Boundary conditions for the wave equation are similar to those of the heat equation, but now we must specify both $f(0;x,y)$ and $f_t(0;x,y)$ at time zero:

- The conditions at $t = 0$ specify the position and velocity of the wave at the initial time.
- Boundary conditions on Ω determine what happens at the ends of the material. Dirichlet conditions correspond to fixing the sides of the wave, e.g. plucking a cello string, which is held flat at its two ends on the instrument. Neumann conditions correspond to leaving the ends of the wave untouched like the end of a whip.

14.4 Derivatives as Operators

In PDEs and elsewhere, we can think of derivatives as operators acting on functions the same way that matrices act on vectors. Our choice of notation often reflects this parallel: The derivative ${}^{df}/_{dx}$ looks like the product of an operator ${}^{d}/_{dx}$ and a function f. In fact, differentiation is a *linear operator* just like matrix multiplication, since for all $f, g : \mathbb{R} \to \mathbb{R}$ and $a, b \in \mathbb{R}$

$$\frac{d}{dx}(af(x) + bg(x)) = a\frac{d}{dx}f(x) + b\frac{d}{dx}g(x).$$

In fact, when we discretize PDEs for numerical solution, we can carry this analogy out completely. For example, consider a function f on $[0,1]$ discretized using $n+1$ evenly-spaced samples, as in Figure NUMBER. Recall that the space between two samples is $h = {}^{1}/_{n}$. In Chapter 12, we developed an approximation for the second derivative $f''(x)$:

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

Suppose our n samples of $f(x)$ on $[0,1]$ are $y_0 \equiv f(0), y_1 \equiv f(h), y_2 \equiv f(2h), \ldots, y_n = f(nh)$. Then, applying our formula above gives a strategy for approximating f'' at each grid point:

$$y_k'' \equiv \frac{y_{k+1} - 2y_k + y_{k-1}}{h^2}$$

That is, the second derivative of a function on a grid of points can be computed using the 1— − 2—1 stencil illustrated in Figure NUMBER(a).

One subtlety we did not address is what happens at y_0'' and y_n'', since the formula above would require y_{-1} and y_{n+1}. In fact, this decision encodes the *boundary conditions* introduced in §14.2. Keeping in mind that $y_0 = f(0)$ and $y_n = f(1)$, examples of possible boundary conditions for f' include:

- Dirichlet boundary conditions: $y_{-1} = y_{n+1} = 0$, that is, simply fix the value of y beyond the endpoints
- Neumann boundary conditions: $y_{-1} = y_0$ and $y_{n+1} = y_n$, encoding the boundary condition $f'(0) = f'(1) = 0$.
- Periodic boundary conditions: $y_{-1} = y_n$ and $y_{n+1} = y_0$, making the identification $f(0) = f(1)$

Suppose we stack the samples y_k into a vector $\vec{y} \in \mathbb{R}^{n+1}$ and the samples y''_k into a second vector $\vec{w} \in \mathbb{R}^{n+1}$. Then, our construction above it is easy to see that $h^2\vec{w} = L_1\vec{y}$, where L_1 is one of the choices below:

$$\underset{\textbf{Dirichlet}}{\begin{pmatrix} -2 & 1 & & & & \\ 1 & -2 & 1 & & & \\ & 1 & -2 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & -2 & 1 \\ & & & & 1 & -2 \end{pmatrix}} \quad \underset{\textbf{Neumann}}{\begin{pmatrix} -1 & 1 & & & & \\ 1 & -2 & 1 & & & \\ & 1 & -2 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & -2 & 1 \\ & & & & 1 & -1 \end{pmatrix}} \quad \underset{\textbf{Periodic}}{\begin{pmatrix} -2 & 1 & & & & 1 \\ 1 & -2 & 1 & & & \\ & 1 & -2 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & -2 & 1 \\ 1 & & & & 1 & -2 \end{pmatrix}}$$

That is, the matrix L can be thought of as a discretized version of the operator $\frac{d^2}{dx^2}$ acting on $\vec{y} \in \mathbb{R}^{n+1}$ rather than functions $f : [0,1] \to \mathbb{R}$.

We can write a similar approximation for $\nabla^2 f$ when we sample $f : [0,1] \times [0,1] \to \mathbb{R}$ with a grid of values, as in Figure NUMBER. In particular, recall that in this case $\nabla^2 f = f_{xx} + f_{yy}$, so in particular we can sum up x and y second derivatives as we did in the one-dimensional example above. This leads to a doubled-over 1— $-$ 2—1 stencil, as in Figure NUMBER. If we number our samples as $y_{k,\ell} \equiv f(kh, \ell h)$, then our formula for the Laplacian of f is in this case:

$$(\nabla^2 y)_{k,\ell} \equiv \frac{1}{h^2}(y_{(k-1),\ell} + y_{k,(\ell-1)} + y_{(k+1),\ell} + y_{k,(\ell+1)} - 4y_{k,\ell})$$

If we once again combine our samples of y and y'' into $\vec{y}$ and $\vec{w}$, then using a similar construction and choice of boundary conditions we can once again write $h^2\vec{w} = L_2\vec{y}$. This two-dimensional grid Laplacian L_2 appears in many image processing applications, where (k, ℓ) is used to index pixels on an image.

A natural question to ask after the discussion above is why we jumped to the second derivative Laplacian in our discussion above rather than discretizing the first derivative $f'(x)$. In principle, there is no reason why we could not make similar matrices D implementing forward, backward, or symmetric difference approximations of f'. A few technicalities, however, make this task somewhat more difficult, as detailed below.

Most importantly, we must decide which first derivative approximation to use. If we write y'_k as the forward difference $\frac{1}{h}(y_{k+1} - y_k)$, for example, then we will be in the unnaturally asymmetric position of needing a boundary condition at y_n but not at y_0. Contrastingly, we could use the symmetric difference $\frac{1}{2h}(y_{k+1} - y_{k-1})$, but this discretization suffers from a more subtle *fencepost problem* illustrated in Figure NUMBER. In particular, this version of y'_k ignores the value of y_k and only looks at its neighbors y_{k-1} and y_{k+1}, which can create artifacts since each row of D only involves y_k for either even or odd k but not both.

If we use forward or backward derivatives to avoid the fencepost problems, we lose an order of accuracy and also suffer from the asymmetries described above. As with the leapfrog integration algorithm in §13.4.2, one way to avoid these issues is to think of the derivatives as living on *half* gridpoints, illustrated in Figure NUMBER. In the one-dimensional case, this change corresponds to labeling the difference $\frac{1}{h}(y_{k+1} - y_k)$ as $y_{k+1/2}$. This technique of placing different derivatives on vertices, edges, and centers of grid cells is particularly common in fluid simulation, which maintains pressures, fluid velocities, and so on at locations that simplify calculations.

These subtleties aside, our main conclusion from this discussion is that if we discretize a function $f(\vec{x})$ by keeping track of samples (x_i, y_i) then most reasonable approximations of derivatives of f will be computable as a product $L\vec{x}$ for some matrix L. This observation completes the analogy: "Derivatives act on functions as matrices act on vectors." Or in standardized exam notation:

Derivatives : Functions :: Matrices : Vectors

14.5 Solving PDEs Numerically

Much remains to be said about the theory of PDEs. Questions of existence and uniqueness as well as the possibility of characterizing solutions to assorted PDEs leads to nuanced discussions using advanced aspects of real analysis. While a complete understanding of these properties is needed to prove effectiveness of PDE discretizations rigorously, we already have enough to suggest a few techniques that are used in practice.

14.5.1 Solving Elliptic Equations

We already have done most of the work for solving elliptic PDEs in §14.4. In particular, suppose we wish to solve a linear elliptic PDE of the form $\mathcal{L}f = g$. Here $\mathcal{L}$ is a differential operator; for example, to solve the Laplace's equation we would take $\mathcal{L} \equiv \nabla^2$, the Laplacian. Then, in §14.4 we showed that if we discretize f by taking a set of samples in a vector $\vec{y}$ with $y_i = f(x_i)$, then a corresponding approximation of $\mathcal{L}f$ can be written $L\vec{y}$ for some matrix L. If we also discretize g using samples in a vector $\vec{b}$, then solving the elliptic PDE $\mathcal{L}f = g$ is approximated by solving the linear system $L\vec{y} = \vec{b}$.

Example 14.7 (Elliptic PDE discretization). *Suppose we wish to approximate solutions to $f''(x) = g(x)$ on $[0, 1]$ with boundary conditions $f(0) = f(1) = 0$. We will approximate $f(x)$ with a vector $\vec{y} \in \mathbb{R}^n$ sampling f as follows:*

$$\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} \approx \begin{pmatrix} f(h) \\ f(2h) \\ \vdots \\ f(nh) \end{pmatrix}$$

where $h = 1/n+1$. We do not add samples at $x = 0$ or $x = 1$ since the boundary conditions determine values there. We will use $\vec{b}$ to hold an analogous set of values for $g(x)$.

Given our boundary conditions, we discretize $f''(x)$ as $\frac{1}{h^2}L\vec{y}$, where L is given by:

$$L \equiv \begin{pmatrix} -2 & 1 & & & & \\ 1 & -2 & 1 & & & \\ & 1 & -2 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & -2 & 1 \\ & & & & 1 & -2 \end{pmatrix}$$

Thus, our approximate solution to the PDE is given by $\vec{y} = h^2L^{-1}\vec{b}$.

Just as elliptic PDEs are the most straightforward PDEs to solve, their discretizations using matrices as in the above example are the most straightforward to solve. In fact, generally the discretization of an elliptic operator $\mathcal{L}$ is a *positive definite* and *sparse* matrix perfectly suited for the solution techniques derived in Chapter 10.

Example 14.8 (Elliptic operators as matrices)**.** *Consider the matrix L from Example 14.7. We can show L is negative definite (and hence the* positive *definite system $-L\vec{y} = -h^2\vec{b}$ can be solved using conjugate gradients) by noticing that $-L = D^\top D$ for the matrix $D \in \mathbb{R}^{(n+1)\times n}$ given by:*

$$D = \begin{pmatrix} 1 & & & & \\ -1 & 1 & & & \\ & -1 & 1 & & \\ & & \ddots & \ddots & \\ & & & -1 & 1 \\ & & & & -1 \end{pmatrix}$$

This matrix is nothing more than the finite-differenced first *derivative, so this observation parallels the fact that $d^2f/dx^2 = d/dx(df/dx)$. Thus, $\vec{x}^\top L\vec{x} = -\vec{x}^\top D^\top D\vec{x} = -\|D\vec{x}\|_2^2 \leq 0$, showing L is negative semidefinite. It is easy to see $D\vec{x} = 0$ exactly when $\vec{x} = 0$, completing the proof that L is in fact negative definite.*

14.5.2 Solving Parabolic and Hyperbolic Equations

Parabolic and hyperbolic equations generally introduce a *time* variable into the formulation, which also is differentiated but potentially to lower order. Since solutions to parabolic equations admit many stability properties, numerical techniques for dealing with this time variable often are stable and well-conditioned; contrastingly, more care must be taken to treat hyperbolic behavior and prevent dampening of motion over time.

Semidiscrete Methods Probably the most straightforward approach to solving simple time-dependent equations is to use a *semidiscrete* method. Here, we discretize the domain but not the time variable, leading to an ODE that can be solved using the methods of Chapter 13.

Example 14.9 (Semidiscrete heat equation)**.** *Consider the heat equation in one variable, given by $f_t = f_{xx}$, where $f(t;x)$ represents the heat at position x and time t. As boundary data, the user provides a* *function $f_0(x)$ such that $f(0;x) \equiv f_0(x)$; we also attach the boundary $x \in \{0,1\}$ to a refrigerator and thus enforce $f(t;0) = f(t;1) = 0$.*

Suppose we discretize the x variable by defining:

$$\begin{aligned}\bar{f}_1(t) &\equiv f(h;t)\\ \bar{f}_2(t) &\equiv f(2h;t)\\ &\vdots \quad \vdots\\ \bar{f}_n(t) &\equiv f(nh;t),\end{aligned}$$

where as in Example 14.7 we take $h = 1/n+1$ and omit samples at $x \in \{0,1\}$ since they are provided by the boundary conditions.

Combining these $\bar{f}_i$'s, we can define $\bar{f}(t) : \mathbb{R} \to \mathbb{R}^n$ to be the semidiscrete *version of f where we have sampled in space but not time. By our construction, the semidiscrete PDE approximation is the ODE given by $\bar{f}'(t) = L\bar{f}(t)$.*

The previous example shows an instance of a very general pattern for parabolic equations. When we simulate *continuous* phenomena like heat moving across a domain or chemicals diffusing through a membrane, there is usually one time variable and then several spatial variables that are differentiated in an elliptic way. When we discretize this system semidiscretely, we can then use ODE integration strategies for their solution. In fact, in the same way that the matrix used to solve a linear elliptic equation as in §14.5.1 generally is positive or negative definite, when we write a semidiscrete parabolic PDE $\bar{f}' = L\bar{f}$, the matrix L usually is negative definite. This observation implies that $\bar{f}$ solving this continuous ODE is unconditionally stable, since negative eigenvalues are damped out over time.

As outlined in the previous chapter, we have many choices for solving the ODE in time resulting from a spatial discretization. If time steps are small and limited, explicit methods may be acceptable. Implicit solvers often are applied to solving parabolic PDEs; diffusive behavior of implicit Euler may generate inaccuracy but behaviorally speaking appears similar to diffusion provided by the heat equation and may be acceptable even with fairly large time steps. Hyperbolic PDEs may require implicit steps for stability, but advanced integrators such as "symplectic integrators" can prevent oversmoothing caused by these types of steps.

One contrasting approach is to write solutions of semidiscrete systems $\bar{f}' = L\bar{f}$ in terms of eigenvectors of L. Suppose $\vec{v}_1, \ldots, \vec{v}_n$ are eigenvectors of L with eigenvalues $\lambda_1, \ldots, \lambda_n$ and that we know $\bar{f}(0) = c_1\vec{v}_1 + \cdots + c_n\vec{v}_n$. Then, recall that the solution of $\bar{f}' = L\bar{f}$ is given by:

$$\bar{f}(t) = \sum_i c_i e^{\lambda_i t}\vec{v}_i$$

This formula is nothing new beyond §5.1.2, which we introduced during our discussion of eigenvectors and eigenvalues. The eigenvectors of L, however, may have physical meaning in the case of a semidiscrete PDE, as in Example 14.5, which showed that eigenvectors of Laplacians L correspond to different resonant vibrations of the domain. Thus, this eigenvector approach can be applied to develop, for example, "low-frequency approximations" of the initial value data by truncating the sum above over i, with the advantage that t dependence is known exactly without time stepping.

Example 14.10 (Eigenfunctions of the Laplacian)**.** *Figure NUMBER shows eigenvectors of the matrix L from Example 14.7. Eigenvectors with low eigenvalues correspond to* low-frequency *functions on* $[0, 1]$ *with values fixed on the endpoints and can be good approximations of* $f(x)$ *when it is relatively smooth.*

Fully Discrete Methods Alternatively, we might treat the space and time variables more democratically and discretize them both simultaneously. This strategy yields a system of equations to solve more like §14.5.1. This method is easy to formulate by paralleling the elliptic case, but the resulting linear systems of equations can be large if dependence between time steps has a global reach.

Example 14.11 (Fully-discrete heat diffusion)**.** *Explicit, implicit, Crank-Nicolson. Not covered in CS 205A.*

It is important to note that in the end, even semidiscrete methods can be considered fully discrete in that the time-stepping ODE method still discretizes the t variable; the difference is mostly for classification of how methods were derived. One advantage of semidiscrete techniques, however, is that they can adjust the time step for t depending on the current iterate, e.g. if objects are moving quickly in a physical simulation it might make sense to take more time steps and resolve this motion. Some methods even adjust the discretization of the domain of x values in case more resolution is needed near local discontinuities or other artifacts.

14.6 Method of Finite Elements

Not covered in 205A.

14.7 Examples in Practice

In lieu of a rigorous treatment of all commonly-used PDE techniques, in this section we provide examples of where they appear in practice in computer science.

14.7.1 Gradient Domain Image Processing

14.7.2 Edge-Preserving Filtering

14.7.3 Grid-Based Fluids

14.8 To Do

- More on existence/uniqueness
- CFL conditions
- Lax equivalence theorem
- Consistency, stability, and friends

14.9 Problems

- Show 1d Laplacian can be factored as $D^\top D$ for first derivative matrix D
- Solve first-order PDE

Acknowledgments

[Discussion goes here]

I owe many thanks to the students of Stanford's CS 205A, Fall 2013 for catching numerous typos and mistakes in the development of this book. The following is a no-doubt incomplete list of students who contributed to this effort: Tao Du, Lennart Jansson, Miles Johnson, Luke Knepper, Minjae Lee, Nisha Masharani, John Reyna, William Song, Ben-Han Sung, Martina Troesch, Ozhan Turgut, Patrick Ward, Joongyeub Yeo, and Yang Zhao.

Special thanks to Jan Heiland and Tao Du for helping clarify the derivation of the BFGS algorithm.

[More discussion here]