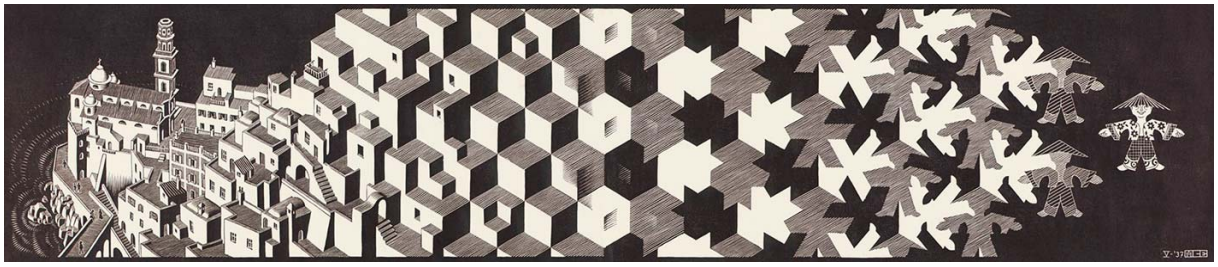


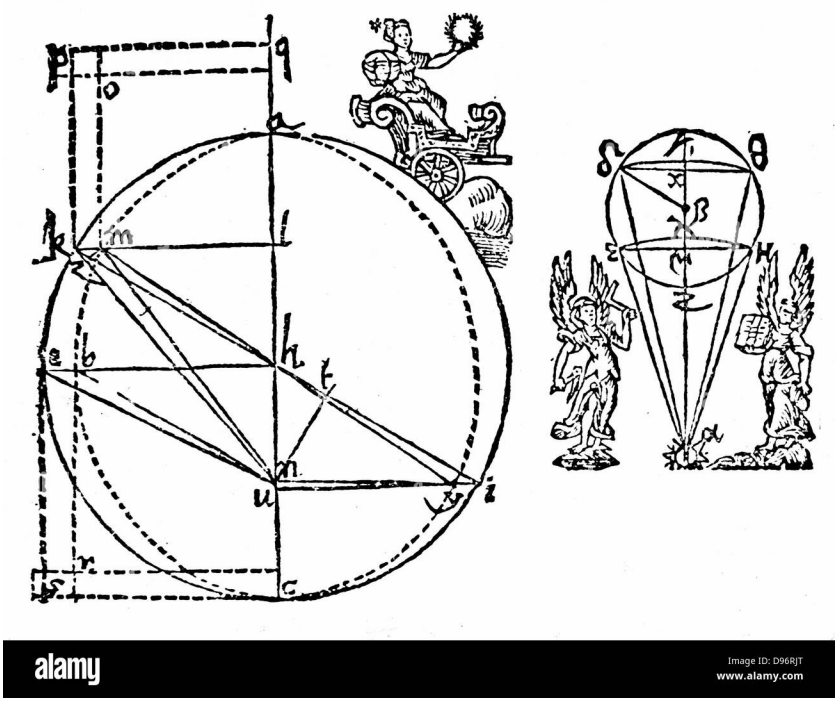
Numerical Analysis

Math 327 Lecture notes

Subhadip Chowdhury, PhD

THE COLLEGE OF WOOSTER

Spring 2022



Kepler's illustration to explain his discovery of the elliptical orbit of Mars.

From Johannes Kepler's 'Astronomia Nova', 1609, (ETH Bibliothek). In this text Kepler derives his famous equation that solves two-body orbital motion,

$$M = E - e\sin E,$$

where M (the mean anomaly) and e (the eccentricity) are known, and one solves for E (the eccentric anomaly). This vital problem spurred the development of algorithms for solving nonlinear equations.

Contents

Acknowledgement

Introduction

[*] We model our world with continuous mathematics. Whether our interest is natural science, engineering, even finance and economics, the models we most often employ are functions of real variables. The equations can be linear or nonlinear, involve derivatives, integrals, combinations of these and beyond. The tricks and techniques one learns in algebra and calculus for solving such systems *exactly* cannot tackle the complexities that arise in serious applications. Exact solution may require an intractable amount of work; worse, for many problems, it is impossible to write an exact solution using elementary functions like polynomials, roots, trig functions, and logarithms.

This course tells a marvelous success story. Through the use of clever algorithms, careful analysis, and speedy computers, we can construct *approximate* solutions to these otherwise intractable problems with remarkable speed. Trefethen[†] defines numerical analysis to be

'the study of algorithms for the problems of continuous mathematics'.

This course takes a tour through many such algorithms, sampling a variety of techniques suitable across many applications. We aim to assess alternative methods based on both accuracy and efficiency, to discern well-posed problems from ill-posed ones, and to see these methods in action through computer implementation.

Perhaps the importance of numerical analysis can be best appreciated by realizing the impact its disappearance would have on our world. The space program would evaporate; aircraft design would be hobbled; weather forecasting would again become the stuff of soothsaying and almanacs. The ultrasound technology that uncovers cancer and illuminates the womb would vanish. Google couldn't rank web pages. Even the letters you are reading, whose shapes are specified by polynomial curves, would suffer. (Several important exceptions involve discrete, not continuous, mathematics: combinatorial optimization, cryptography and gene sequencing.)

On one hand, we are interested in *complexity*: we want algorithms that minimize the number of calculations required to compute a solution. But we are also interested in the *quality* of approximation: since we do not obtain exact solutions, we must understand the accuracy of our answers. Discrepancies arise from approximating a complicated function by a polynomial, a continuum by a discrete grid of points, or the real numbers by a finite set of floating point numbers. Different algorithms for the same problem will differ in the quality of their answers and the labor required to obtain those answers; we will learn how to evaluate algorithms according to these criteria.

Numerical analysis forms the heart of 'scientific computing' or 'computational science and engineering,' fields that also encompass the high-performance computing technology that makes our algorithms practical for problems with millions of variables, visualization techniques that illuminate the data sets that emerge from these computations, and the applications that motivate them.

Though numerical analysis has flourished in the past seventy years, its roots go back centuries, where

[*] This section is adapted from Lecture Notes by Prof. Mark Embree at Virginia Tech.

[†] Lloyd N. Trefethen. ''The Definition of Numerical Analysis''. In: **Bulletin of the Institute for Mathematics and Applications** (1993). URL: https://webs.um.es/eliseo/um/uploads/Main/TrefethendefNA.pdf.

approximations were necessary in celestial mechanics and, more generally, 'natural philosophy'. Science, commerce, and warfare magnified the need for numerical analysis, so much so that the early twentieth century spawned the profession of 'computers', people who conducted computations with hand-crank desk calculators. But numerical analysis has always been more than mere number-crunching, as observed by Alston Householder in the introduction to his *Principles of Numerical Analysis*, published in 1953, the end of the human computer era:

> The material was assembled with high-speed digital computation always in mind, though many techniques appropriate only to ''hand'' computation are discussed. … How otherwise the continued use of these machines will transform the computer's art remains to be seen. But this much can surely be said, that their effective use demands a more profound understanding of the mathematics of the problem, and a more detailed acquaintance with the potential sources of error, than is ever required by a computation whose development can be watched, step by step, as it proceeds.

Thus the *analysis* component of 'numerical analysis' is essential. We rely on tools of classical real analysis, such as continuity, differentiability, Taylor expansion, and convergence of sequences and series.

Matrix computations play a fundamental role in numerical analysis. Discretization of continuous variables turns calculus into algebra. Algorithms for the fundamental problems in linear algebra are covered in MATH 211. This course is a prerequisite for Math 327; when the methods we discuss this semester connect to matrix techniques, we will provide pointers.

Chapter 1 | Interpolation

The idea behind interpolation is something you've practiced since Kindergarten: Connecting the dots. Now that we're all grown up, we call our "dots" data points in the $xy-$plane, and we connect them with very specific kinds of curves.

In the late 1950's, engineers at two rival French automakers (Paul de Casteljau at Citroen and Pierre Bézier at Renault) used a new way of interpolating data points to generate curves that could be manipulated easily to design automobiles. Their method was a trade secret until Bézier publicized the concept in 1962, which is why we now call the resulting objects "Bézier curves". The same methods are now commonly used not only in industrial design, but in computer graphics. In particular, most printers use these methods to generate fonts. Here's a schematic* that illustrates how the outline of an "G" might be drawn by interpolating 23 data points.

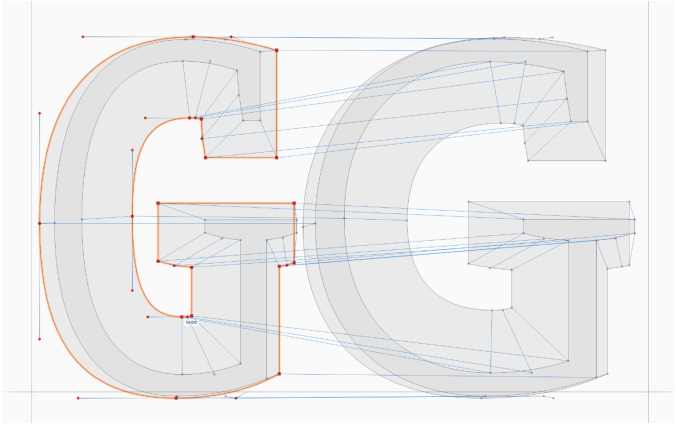


Figure 1.1: Font Curve Mathematics

Notice that the sides are all straight lines and ellipses in this case. Linear functions are first-degree polynomials, and conic sections are second degree. The reason that auto designs and printer fonts look much better than the above is because they use higher-degree polynomials in the interpolation.

§1.1 Polynomial Interpolation: definitions and notations

Definition 1.1

The set of continuous functions that map from $[a,b] \subseteq \mathbb{R}$ to $\mathbb{R}$ is denoted by $C[a,b]$. The set of continuous functions whose first r derivatives are also continuous on $[a,b]$ is denoted by $C^r[a,b]$. (Note that $C^0[a,b] \equiv C[a,b]$.)

Definition 1.2

The set of polynomials of degree n or less is denoted by $\mathcal{P}_n$.

*Taken from https://www.lucasfonts.com/learn/interpolation-curve-technicalities

Note that $C[a,b]$, $C^r[a,b]$ (for any $a < b, r \geq 0$) and $\mathcal{P}_n$ are **vector spaces** of functions (since linear combinations of such functions maintain continuity and polynomial degree). Furthermore, note that every element p_n in $\mathcal{P}_n$ can be uniquely determined by $(n+1)$ constants $c_i \in \mathbb{R}$ for $(0 \leq i \leq n)$, as we can write

$$p_n(x) = c_0 + c_1 x + c_2 x^2 + \ldots + c_n x^n \tag{1.1}$$

Hence $\mathcal{P}_n$ is an $n+1$ dimensional subspace of $C[a,b]$.

Working with polynomials is especially convenient in a computer as we only need to store the $n+1$ coefficients. Operations such as taking the derivative or integrating f are also easier. The idea in this chapter is to then find a polynomial that approximates a general function $f \in C[a,b]$. The following theorem due to Weierstrass says that this is always doable!

Theorem 1.3: Weierstrass Approximation Theorem

For any $f \in C[a,b]$ and any $\varepsilon > 0$, there exists a polynomial $p_n(x)$ of finite degree such that

$$\max_{x \in [a,b]} |f(x) - p_n(x)| \leq \varepsilon.$$

In this chapter, we look for a suitable polynomial p_n by using the technique of **interpolation**.

1.1.1 The Polynomial Interpolation Problem

The polynomial interpolation problem can be stated as:

Given $f \in C[a,b]$ and $n+1$ points $\{x_j\}_{j=0}^n$ satisfying

$$a \leq x_0 < x_1 < \cdots < x_n \leq b,$$

determine some $p_n \in \mathcal{P}_n$ such that

$$p_n(x_j) = f(x_j) \quad \text{for } j = 0, \ldots, n.$$

The points $f(x_j)$ are usually called *nodes* and the polynomial p_n is called a *f-interpolating polynomial*.

It shall become clear why we require $n+1$ points $x_0, \ldots, x_n$, and no more, to determine a degree-n polynomial p_n. (You know the $n = 1$ case well: two points determine a unique line.) If the number of data points were smaller, we could construct infinitely many degree-n interpolating polynomials. Were it larger, there would in general be no degree-n interpolant.

As numerical analysts, we seek answers to the following questions:

- Does such a polynomial $p_n \in \mathcal{P}_n$ exist?
- If so, is it unique?
- Does $p_n \in \mathcal{P}_n$ behave like $f \in C[a,b]$ at points $x \in [a,b]$ when $x \neq x_j$ for $j = 0, \ldots, n$?
- How can we compute $p_n \in \mathcal{P}_n$ efficiently on a computer?
- If we want to add a new interpolation point x_{n+1}, can we easily adjust p_n to give an interpolating polynomial p_{n+1} of one higher degree?

- How should the interpolation points $\{x_j\}$ be chosen?

Regarding this last question, we should note that, in practice, we are not always able to choose the interpolation points as freely as we might like. For example, our 'continuous function $f \in C[a,b]$ ' could actually be a discrete list of previously collected experimental data, and we are stuck with the values $\{x_j\}_{j=0}^n$ at which the data was measured. In **??** we will see an alternative approach, appropriate for noisy data, where the overall error $|f(x) - p_n(x)|$ is minimised, without requiring p_n to match f at specific points.

§I.2 Constructing Interpolating Polynomials in Monomial Basis

Every polynomial $p_n(x)$ in the vector space $\mathcal{P}_n$ can be written as a linear combination of the basis functions $1, x, x^2, \ldots, x^n$; these basis functions are called *monomials*.

To construct the polynomial interpolant to f , we merely need to determine the proper values for the coefficients $c_0, c_1, \ldots, c_n$ in eq. (1.1). This reduces to solving the linear system

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ 1 & x_2 & x_2^2 & \cdots & x_2^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} f(x_0) \\ f(x_1) \\ f(x_2) \\ \vdots \\ f(x_n) \end{bmatrix}, \tag{1.2}$$

which we denote as $\mathrm{A}\vec{c} = \vec{f}$. Matrices of this form, called Vandermonde matrices, arise in a wide range of applications. Provided all the interpolation points $\{x_j\}$ are distinct, one can show that this matrix is invertible.[†]

Hence, fundamental properties of linear algebra allow us to confirm that there is **exactly one** degree-n polynomial (whose coefficients are given by $\mathrm{A}^{-1}\vec{f}$), that interpolates f at the given $n+1$ distinct interpolation points.

Theorem 2.4

For any set of $n+1$ different points $\{a \le x_0, x_1, \ldots, x_n \le b\}$ and any function $f \in \mathrm{C}[a,b]$, there exists a unique interpolating polynomial of degree less than or equal to n.

We may also prove uniqueness by the following straightforward argument.

Proof of theorem 4.

[Uniqueness Part.] Suppose that in addition to p_n there is another interpolating polynomial q_n. Then the degree-n polynomial $r_n = p_n - q_n$ has $n+1$ roots x_i for $0 \le i \le n$. From the Fundamental theorem of Algebra, this is possible only if $r_n(x) = 0$. ■

Note: The unique polynomial through $n+1$ points may have degree $< n$. This happens when $c_n = 0$ in the solution eq. (1.2).

One way to solve the above linear system in eq. (1.2) is Gaussian elimination, which we will come back to in section 2.2. However, this is computationally inefficient (taking $\mathcal{O}(n^3)$ operations). In practice, we choose a different basis for p_n. There are two common choices, due to Lagrange and Newton.

[†] In fact, the determinant is given by $\prod_{0 \le i < j \le n} (x_i - x_j)$.

§I.3 Constructing interpolants in the Lagrange basis

Observe that the solution for c_i in eq. (1.2) gives us a way to write $p_n(x)$ as a linear combination of the basis $\{x^j\}_{j=0}^n$ of $\mathcal{P}_n$. However, we might want to choose a different basis for P_n if it simplifies the calculation.

Lagrange's method uses a special basis of polynomials $\{\ell_j\}$ in which the interpolation equations reduce to the identity matrix. In other words, the coefficients in this basis are just the function values,

$$p_n(x) = \sum_{j=0}^{n} f(x_j)\ell_j(x)$$

Recall that we want $p_n(x_i) = f(x_i)$ for $0 \le i \le n$. Hence, for above formula to be true, we need to construct ℓ_j with $\ell_j(x_i) = 0$ if $j \neq i$, but $\ell_j(x_i) = 1$ if $j = i$.

Definition 3.5

The Lagrange polynomials $\{\ell_j\}$, for $0 \le j \le n$, are defined as

$$\ell_j(x) = \prod_{\substack{i=0 \\ i\neq j}}^{n} \frac{x - x_i}{x_j - x_i}$$

Observe that this approach to constructing basis polynomials leads to a *diagonal matrix* A in the equation $\mathrm{A}\vec{c} = \vec{f}$ for the coefficients.

Note: The Lagrange form of the interpolating polynomial is easy to write down, but expensive to evaluate since all of the ℓ_k must be computed. Moreover, changing any of the nodes means that the ℓ_k must all be recomputed from scratch, and similarly for adding a new node (moving to higher degree).

Question 1. Lab

Compute the quadratic interpolating polynomial to $f(x) = \cos x$ with nodes $\{-\pi/4, 0, \pi/4\}$ using the Lagrange basis.

Then use your favorite mathematical or programming software to plot the polynomials, the interpolant, and the function in the same picture.

§1.4 Interpolation Error Bounds

Whenever we approximate a function $f(x)$ with a (usually) simpler function, we can create the error function associated with the approximation by taking the difference between the two functions. For instance, the Taylor polynomial of degree n at $x = a$

$$T_n(x) := f(a) + f'(a)(x-a) + \cdots + \frac{f^{(n)}(a)}{n!}(x-a)^n$$

approximates $f(x)$ with error function equal to the Taylor remainder term

$$f(x) - T_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}(x-a)^{n+1},$$

where the unidentified point ξ between x and a can change for different values of x. Now we'll look more carefully at the error involved in f-interpolation.

1.4.1 Errors in f–interpolation

From theorem 4, we know that there is a unique polynomial $p_n(x)$ of degree less than or equal to n interpolating $f \in C[a,b]$ at the set of points $\{x_0, x_1, \ldots, x_n\}$. The error function associated with this "approximation" of $f(x)$ is then simply

$$E_n(x) = f(x) - p_n(x) \tag{1.3}$$

Example 4.6

You should have found that the quadratic interpolant for $f(x) = \cos x$ with nodes $\{-\pi/4, 0, \pi/4\}$ using the Lagrange basis was $p_2(x) = \frac{16}{\pi^2}\left(\frac{1}{\sqrt{2}} - 1\right)x^2 + 1$. So the error is

$$E_2(x) = \left|\cos x - \frac{16}{\pi^2}\left(\frac{1}{\sqrt{2}} - 1\right)x^2 - 1\right|$$

■ **Question 2.** **Lab**

Plot the error function for example 6. What is $E_n(x_i)$? Is this true in general for any function $f(x)$?

This means that we can repeatedly apply the Factorization Theorem to factor out terms like $(x - x_i)$ from the error function $E_n(x)$, to get the expression

$$E_n(x) = (x-x_0)(x-x_1)(x-x_2)\ldots(x-x_n)r(x) = r(x)\prod_{i=0}^{n}(x-x_i) \tag{1.4}$$

for some "remainder" function $r(x)$. To determine $r(x)$, we are going to do something that may seem a little strange at this point. We'll define a new function $W(x)$ that is the difference between our two expressions (1.3) and (1.4) for the error function $E_n(x)$, with a slight twist. Since we already know the value of $E_n(x_i) = 0$, we choose any other point $\hat{x} \neq x_i$ in $[a,b]$ as the input to the remainder function $r(\hat{x})$, and we leave the other variables x in $E_n(x)$ as they are:

$$\mathrm{W}(x) = \overbrace{f(x) - p_n(x)}^{\mathrm{E}_n(x) \text{ via } (1.3)} - \overbrace{(x-x_0)(x-x_1)\cdots(x-x_n)\,r(\hat{x})}^{\text{"twisted" } \hat{\mathrm{E}}_n(x) \text{ via } (1.4)}. \tag{1.5}$$

■ Question 3. □

(a) What is $\mathrm{W}(x_i)$ *equal to?*

(b) What is $\mathrm{W}(\hat{x})$ *equal to?*

We conclude that W has at least $n+2$ zeroes in any interval containing the set of points $\{\hat{x}, x_0, x_1, \ldots, x_n\}$. We'll also apply a general fact about "interlacing" zeroes.

Theorem 4.7: Rolle's Theorem

If f is continuous on $[a,b]$ and differentiable on (a,b), with $f(a) = f(b) = 0$, then there exists $\xi \in (a,b)$ with $f'(\xi) = 0$.

If we apply Rolle's theorem to each consecutive pair of zeroes of W from the list $\{\hat{x}, x_0, x_1, \ldots, x_n\}$, we conclude that there are at least $n+1$ zeroes of the derivative function $\mathrm{W}'(x)$ in the interval $[a,b]$ (since there is a zero between each consecutive pair of $n+2$ points).

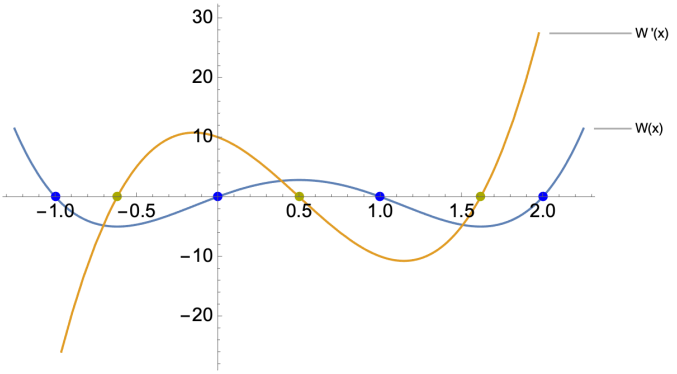


Figure 1.2: Interlacing Zeroes

If we now apply Rolle's theorem again to each consecutive pair of the $n+1$ zeroes of the derivative function $\mathrm{W}'(x)$, we conclude that there are n zeroes of the second-derivative function $\mathrm{W}''(x)$. We can continue in this manner to deduce that there is at least one zero ξ of the $(n+1)$st derivative function $\mathrm{W}^{(n+1)}(x)$.

■ Question 4. Group

Use $\mathrm{W}(x) = f(x) - p_n(x) - (x-x_0)(x-x_1)\cdots(x-x_n)\,r(\hat{x})$ *to evaluate* $\mathrm{W}^{(n+1)}(\xi)$. *Then simplify and solve* $\mathrm{W}^{(n+1)}(\xi) = 0$ *to find* $r(\hat{x})$.

HINT: only the highest power in the last term matters.

We conclude the following essential result.

Theorem 4.8: Interpolation Error Formula (Cauchy)

Suppose $f \in C^{n+1}[a,b]$ and let $p_n \in \mathcal{P}_n$ denote the f–interpolating polynomial with nodes $\{x_0, x_1, \ldots, x_n\}$ for distinct points $x_j \in [a,b]$. Then for every $x \in [a,b]$ there exists $\xi \in [a,b]$ such that

$$f(x) - p_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{j=0}^{n} (x - x_j).$$

From this formula we can conclude that the worst error over $[a,b]$ can be estimated by the following bound:

$$\max_{x\in[a,b]} |f(x) - p_n(x)| \leq \left(\max_{\xi\in[a,b]} \frac{\left|f^{(n+1)}(\xi)\right|}{(n+1)!} \right) \left(\max_{x\in[a,b]} \prod_{j=0}^{n} \left|x - x_j\right| \right). \tag{1.6}$$

This theorem is the foundation for the convergence theory of piecewise polynomial approximation (section 1.5) and interpolatory quadrature rules for definite integrals (**??**).

Example 4.9

Let's apply the interpolation error bound formula to $f(x) = \sin x$ on $[-5, 5]$. Note that

$$\max_{\xi\in[-5,5]} \left|f^{(n+1)}(\xi)\right| = 1$$

and

$$\max_{x\in[-5,5]} \prod_{j=0}^{n} \left|x - x_j\right| = 10^{n+1}$$

Hence,

$$\max_{x\in[-5,5]} |f(x) - p_n(x)| = \frac{10^{n+1}}{(n+1)!}$$

which does $\to 0$ as $n \to \infty$.

1.4.2 Conclusions

Here are some conclusions we can draw from theorem 8.

- **"Flatter" $f \implies$ better fit:** The smaller the higher-order derivatives of f are on $[a,b]$, the better the interpolation will fit f. As a special case, any polynomial f will eventually (i.e., for n big enough) be fit exactly by an f–interpolating polynomial.
- **Better fit near center:** The error $E_n(x)$ is smaller if x is centered within $[a,b]$, since that makes the product $(x - x_0)(x - x_1)(x - x_2)\ldots(x - x_n)$ relatively smaller.
- **Extrapolation is Dangerous:** Using $p_n(x)$ to approximate $f(x)$ for values of x outside the smallest interval containing the original interpolation points $\{x_0, x_1, \ldots, x_n\}$ causes relatively larger errors, since the product $(x - x_0)(x - x_1)(x - x_2)\ldots(x - x_n)$ is relatively larger.

Note: An interesting feature of the interpolation bound is the polynomial $w(x) = (x - x_0)(x - x_1)(x - x_2)\dots(x - x_n)$. This quantity plays an essential role in approximation theory, and also a closely allied subdiscipline of complex analysis called *potential theory*. Naturally, one might wonder what choice of points $\{x_j\}$ minimizes $|w(x)|$ - we will revisit this question in the next section. For now, we simply note that the points that minimize $|w(x)|$ over $[a, b]$ are called *Chebyshev points*, which are clustered more densely at the ends of the interval $[a, b]$.

Question 5.

Verify that

$$p_1(x) = f(x_0) + \left[\frac{f(x_1) - f(x_0)}{x_1 - x_0}\right](x - x_0)$$

is the unique polynomial of degree ≤ 1 *interpolating* f *at the (distinct) points* x_0 *and* x_1*, and identify what* p_1 *approaches as the interpolating point* x_1 *approaches* x_0*. Explain your thinking.*

Question 6.

Suppose that p_n *is the polynomial of degree* $\leq n$ *that interpolates* $f(x)$ *at the distinct points* $x_0, x_1, \dots, x_n$*. Develop a conjecture about what* p_n *approaches as the interpolating points* $x_1, \dots, x_n$ *all approach* x_0 *simultaneously, and explain your thinking.*

Hint: use the definition of $\mathrm{E}_n(x)$ *and the alternative formula we developed for it in this section.*

§I.5 Lab Assignment I: Runge's Phenomenon and Chebyshev Nodes

You might expect polynomial interpolation to *converge* as $n \to \infty$. Surprisingly, this is not the case if you take equally-spaced nodes x_i. This was shown by Runge in a famous 1901 paper.

Question 7.

Use your favorite mathematical or programming software to define Runge's function

$$f(x) = \frac{1}{1+25x^2}$$

Mathematica Code:

```
f[x_] := 1/(1+25 x^2)
```

Question 8.

Create three different lists of pairs $\{x_i, f(x_i)\}$ *for evenly spaced points* x_i *between* -1 *and* 1. *The three lists* `list4`, `list10`, `list20` *should be made up of points* 0.5 *apart,* 0.2 *apart, and* 0.1 *apart, respectively.*

Mathematica Code:

```
list4 = Table[{i,f[i]}, {i, -1, 1, 0.5}]
```

Question 9.

Define the f*–interpolating polynomial* $p_4(x), p_{10}(x)$, *and* $p_{20}(x)$ *corresponding to the lists of nodes.*

Mathematica Code:

```
p4[x_] := InterpolatingPolynomial[list4,x]//Simplify
```

Check here for Octave code and check here for Python code.

Question 10.

`Plot` f *and the polynomials* p_4, p_{10}, *and* p_{20} *on the same graph and describe what you observe.*

Question 11.

Predict how the plot of an analogous f*–interpolating polynomial* p_{100} *would compare.*

What you are seeing here is referred to as "Runge phenomenon", named after the German mathematician and physicist Carl David Tolmé Runge who first explored this example.[‡]

The problem is (largely) coming from the polynomial

$$w(x) = (x-x_0)(x-x_1)(x-x_2)\dots(x-x_n)$$

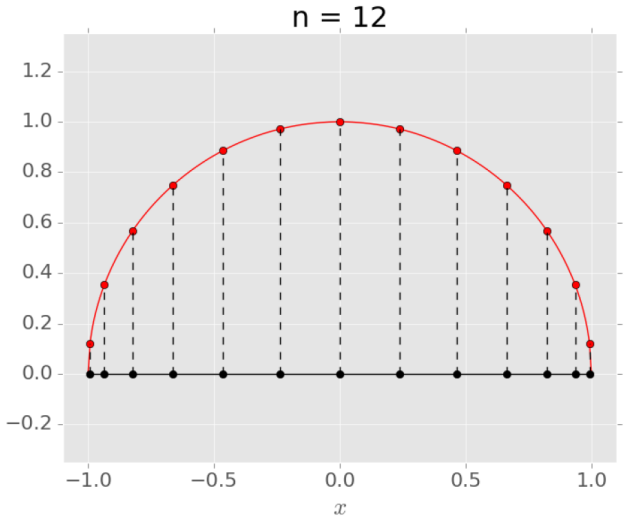


Figure 1.3: Chebyshev Nodes

Since the problems are occurring near the ends of the interval, it would be logical to put more nodes there. A good choice is given by taking equally-spaced points on the unit circle $|z| = 1$, and projecting to the real line.

The points around the circle are

$$\varphi_j = \frac{(2j+1)\pi}{2(n+1)}, \qquad j = 0, \dots, n.$$

so the corresponding Chebyshev nodes are

$$x_j = \cos \frac{(2j+1)\pi}{2(n+1)}, \qquad j = 0, \dots, n. \tag{1.7}$$

Question 12.

The Chebyshev polynomial is defined as[§]

$$T_{n+1}(t) := \cos\left[(n+1)\arccos(t)\right]$$

The fact that T_{n+1} is a polynomial can be proved by induction, you can skip that part. Prove that the Chebyshev nodes are the roots of the Chebyshev polynomial.

Although we will not prove it here, one can show that the Chebyshev nodes are the best possible choice for minimizing the error.

Theorem 5.10: Chebyshev Interpolation

[‡]For a proof of divergence see http://math.stackexchange.com/questions/775405/.

[§]In other words, $T_n(\cos\theta) = \cos(n\theta)$.

Let $x_0, x_1, \ldots, x_n \in [-1,1]$ be distinct. Then $\max_{x\in[-1,1]} |w(x)|$ is minimized if

$$w(x) = \frac{1}{2^n} \mathrm{T}_{n+1}(x)$$

■ Question 13. □

Define and plot the three f–interpolating polynomials corresponding to the following new sets of (not equally-spaced) interpolation points:

(a) the four zeroes of the Chebyshev polynomial $\mathrm{T}_4(x)$. (You can get a list of these zeroes by explicitly defining a list/array of the values from eq. (1.7).)

(b) the ten zeroes of the Chebyshev polynomial $\mathrm{T}_{10}(x)$.

(c) the twenty zeroes of the Chebyshev polynomial $\mathrm{T}_{20}(x)$.

Compare what you observe to Runge's phenomenon for equally-spaced interpolation points.

■ Question 14. □

By expanding the domains (i.e., the x–limits) on your final graphs from question 10 and question 13, describe how extrapolation seems to work in each case.

For `Octave` and `Python`, there are built in additional parameters for creating extrapolation.

§1.6 Spline Interpolation

Last time we saw that Runge's phenomenon can be fixed by using interpolation points (at the zeroes of Chebyshev polynomials) that aren't equally spaced.

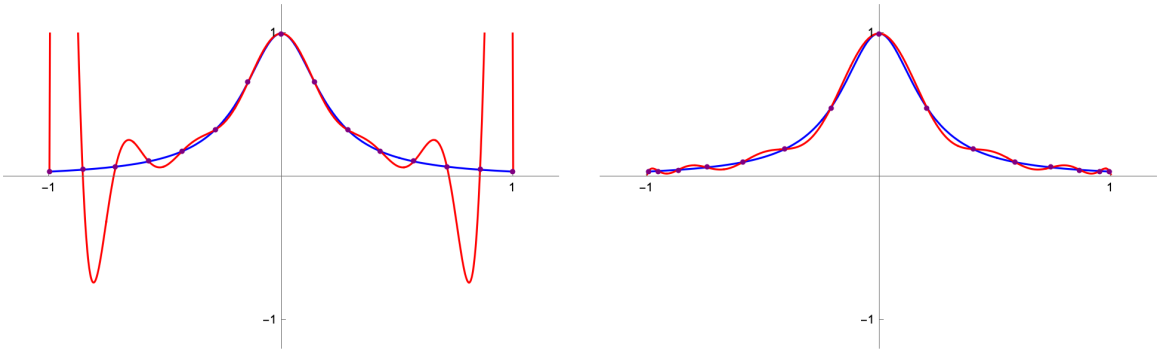


Figure 1.4: Lagrange Interpolant vs. Chebyshev Interpolant

The interpolating polynomial on the left wiggles too much near the ends of the interpolation interval to match Runge's function (which doesn't wiggle at all there). This "over-wiggling" is fixed on the right by using more interpolation points near the ends of the interval (and fewer interpolation points in the middle where the match is good).

Another way to improve the fit of the interpolating function is the basis for modern computer aided design (CAD). This approach doesn't require special interpolation points or high-order polynomials, but instead uses pieces of simple polynomials between interpolation points.

1.6.1 Splines

Long before computers, ship-builders used thin flexible strips of wood called "splines" to draw curves as they were designing ships. The Romanian mathematician Isaac Jacob Schoenberg is credited with using this same term to describe the piecewise polynomial curves that are now fundamental to computer-aided design. The nice thing about polynomial splines is that they are relatively easy to describe as functions, and yet they can assume an extraordinary range of shapes. Think about trying to come up with a simple function whose graph matches some outline you want. This kind of "inverse problem" is solved very nicely by polynomial splines.

1.6.2 Linear Spline Interpolation

The simplest kind of spline uses linear polynomials to interpolate between each pair of adjacent interpolation points. Figure 1.5 shows images of the linear spline interpolation of Runge's function using 5 and 20 equally-spaced interpolation points. These are clearly better global (i.e. on the entire interval) approximations of Runge's function than the high-order (single) polynomials we saw in Runge's phenomenon.

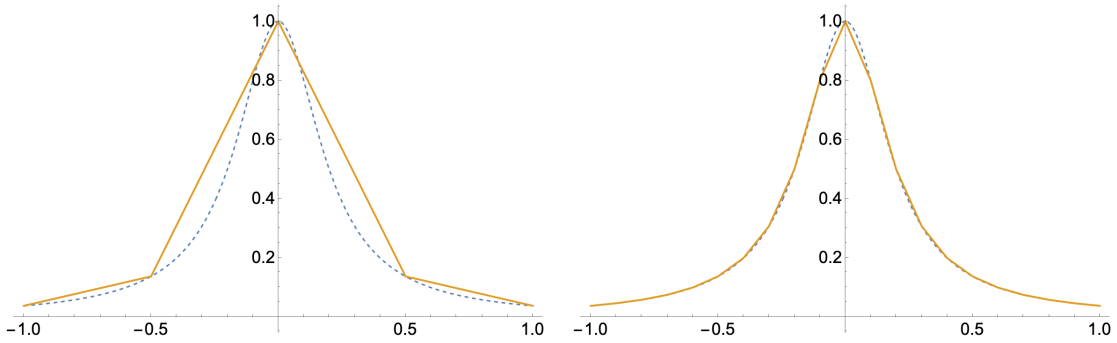


Figure 1.5

Definition 6.11: General Formula

Given $n+1$ data points $\{(x_j, f_j)\}_{j=0}^{n}$, we need to construct n linear polynomials $\{s_j\}_{j=1}^{n}$ such that

$$s_j(x_{j-1}) = f_{j-1}, \quad \text{and} \quad s_j(x_j) = f_j$$

for each $j = 1, \ldots, n$. It is simple to write down a formula for these polynomials (compare to question 5)

$$s_j(x) = f_j - \frac{(x_j - x)}{(x_j - x_{j-1})}(f_j - f_{j-1}).$$

Each s_j is valid on $x \in [x_{j-1}, x_j]$, and the interpolant $S(x)$ is defined as $S(x) = s_j(x)$ for $x \in [x_{j-1}, x_j]$.

Accuracy of Linear Splines

For a function $\varphi \in C^1[a,b]$, the L^2 norm of φ is defined as

$$\|\varphi\|_2 = \int_a^b (\varphi(x))^2 \, dx$$

We will come back to further discussion regarding L^2 norm in the future. For now, we note the following:

Theorem 6.12

If $\sigma(x)$ is a function that interpolates $f(x)$ at $n+1$ distinct points $x_0 < x_2 < \ldots < x_n$ and if $\|\sigma'\|_2$ is finite on $[x_0, x_n]$ then,

$$\|S_L'\|_2 \leq \|\sigma'\|_2$$

where S_L is the linear spline interpolating f at the same points. Thus the linear spline is the "flattest" among all splines.

Question 15. Group

Write a careful argument proving the inequality from theorem 12.

HINT: Start with the difference function $h(x) = \sigma(x) - S(x)$ on a generic subinterval $[x_i, x_{i+1}]$. Show that $\|\sigma'\|_2^2 = \|h'\|_2^2 + \|S_L\|_2^2$. You may need to use integration by parts and the fact that the slope $S'(x)$ is constant on $[x_i, x_{i+1}]$.

One problem with a linear splines can be that they are not necessarily differentiable where two different linear pieces meet. Still, these functions are easy to construct and cheap to evaluate, and can be very useful despite their simplicity. Another problem is that we might need a huge number of linear pieces to reproduce a curved shape. To get around these issues, we can use quadratic or cubic splines instead.

1.6.3 Cubic Splines

The idea behind cubic spline interpolation is to use cubic pieces to interpolate adjacent interpolation points. Cubic functions are defined by 4 coefficients whereas linear functions are defined by only 2.

$$s_i(x) = c_3 x^3 + c_2 x^2 + c_1 x + c_0$$

These extra coefficients allow us to choose cubic pieces that match each other better and/or match the function f better. For instance, we can ensure that the derivatives and the second derivatives of adjacent cubic pieces match at the intersection points, so that the resulting spline will be C^2 everywhere.

The cubic spine requirements can be written as:

$$\begin{aligned}
&s_j\left(x_{j-1}\right) = f\left(x_{j-1}\right), && j = 1, \ldots, n; \\
&s_j\left(x_j\right) = f\left(x_j\right), && j = 1, \ldots, n; \\
&s_j'\left(x_j\right) = s_{j+1}'\left(x_j\right), && j = 1, \ldots, n-1; \\
&s_j''\left(x_j\right) = s_{j+1}''\left(x_j\right), && j = 1, \ldots, n-1.
\end{aligned}$$

Question 16. □

Count the total number of requirements (i.e. equations) and the total number of free variables (the coefficients).

What does this tell you about the number of possible solutions to $S(x)$?

So far, we thus have an *underdetermined system*, and there will be infinitely many choices for the function $S(x)$ that satisfy the constraints.

There are several canonical ways to add two extra constraints that uniquely define S :

- **Natural Boundary Conditions:** requires $S''(x_0) = S''(x_n) = 0$;
- **Clamped Boundary Conditions:** specifies exact values for $S'(x_0)$ and $S'(x_n)$, or assumes both are zero;
- **Not-a-knot Boundary Conditions:**[¶] requires S''' to be continuous at x_1 and x_{n-1}. This forces $s_1 = s_2$ and $s_{n-1} = s_n$.

Natural cubic splines are a popular choice for they can be shown, in a precise sense, to minimize curvature over all the other possible splines. They also model the physical origin of splines, where beams of wood extend straight (i.e., zero second derivative) beyond the first and final 'ducks.'

[¶] In spline parlance, the interpolation points $\{x_j\}_{j=0}^n$ are called *knots*.

Question 17. Lab

In this problem, we will compare the Not-a-knot and clamped boundary conditions.

(a) *Define the function* $f(x) = \sin(20x) + e^{5x/2}$.

(b) *Define a* `list` *of data points* $\{x_i, f(x_i)\}$ *using the* `Table` *command as* x_i *ranges from* 0 *to* 1 *with* 0.1 *increment intervals.*

(c) *Interpolate the data via piecewise-cubic splines, using the Not-a-knot boundary conditions (this is the default).*

In Mathematica, you can use `ResourceFunction["CubicSplineInterpolation"]`. *See this page for usage examples. Alternately, see here for Python, see here for Octave.*

(d) *Next consider clamped boundary conditions. First, analytically compute the derivative at the two endpoints. Then interpolate the data with these values as the BC.*

In Mathematica, use `{{"Clamped",f'[0]},{"Clamped",f'[1]}}` *as an optional argument. In Python, we pass a third argument into cubic spline interpolation package, see* `bc_type`. *In Octave, we extend the supplied* `y` *vector by two elements, making the first element the derivative* α *at the left endpoint, the last element the derivative* β *at the right endpoint, with all of the data previously in* y *in between. In other words, we have* `y_new=[alpha,y,beta]`.

(e) *Plot and compare the second derivatives for each of the interpolant. What differences and similarities do you see?*

(f) *Plot the absolute errors for the two boundary condition methods on the same set of axes. Which one did a better job of estimating the function near the endpoints? Does this match your expectations? Why or why not?*

Chapter 2 | Linear Equations

This chapter is concerned with techniques for solving a linear system of the form

$$\mathrm{A}\vec{\boldsymbol{x}} = \vec{\boldsymbol{b}} \tag{2.1}$$

where A is an $n \times n$ square matrix with elements a_{ij} , and $\vec{\boldsymbol{x}}, \vec{\boldsymbol{b}}$ are $n \times 1$ vectors. We will focus on solving eq. (2.1) both *accurately* and *efficiently*.

Note: Although this seems like a conceptually easy problem (just use Gaussian elimination!), it is actually a hard one when n gets large. Nowadays, linear systems with $n = 1$ million arise routinely in computational problems. And even for small n there are some potential pitfalls, as we will see.

Let's start by recalling some background facts from Linear Algebra:

- A^{T} is the transpose of A, so $(a^{\mathrm{T}})_{ij} = a_{ji}$.
- A is symmetric if $\mathrm{A} = \mathrm{A}^{\mathrm{T}}$.
- A is non-singular iff there exists a solution $\vec{\boldsymbol{x}} \in \mathbb{R}^n$ for every $\vec{\boldsymbol{b}} \in \mathbb{R}^n$.
- A is non-singular iff $\det(\mathrm{A}) \neq 0$.
- A is non-singular iff there exists a unique inverse A^{-1} such that $\mathrm{A}\mathrm{A}^{-1} = \mathrm{A}^{-1}\mathrm{A} = \mathrm{I}_n$.

It follows that eq. (2.1) has a unique solution iff A is non-singular, given by $\vec{\boldsymbol{x}} = \mathrm{A}^{-1}\vec{\boldsymbol{b}}$. As such many algorithms are based on the idea of rewriting eq. (2.1) in a form where the matrix is easier to invert. Easiest to invert are diagonal matrices, followed by orthogonal matrices (where $\mathrm{A}^{-1} = \mathrm{A}^{\mathrm{T}}$). However, the most common method for solving $Ax = b$ transforms the system to triangular form.

§2.1 Triangular Systems

If the matrix A is triangular, then $\mathrm{A}\vec{\boldsymbol{x}} = \vec{\boldsymbol{b}}$ is straightforward to solve. Let's take a look at an example for an *upper triangular* matrix U of the form

$$\mathrm{U} = \begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & u_{nn} \end{pmatrix}$$

Example 1.13: Solving $\mathrm{U}\vec{\boldsymbol{x}} = \vec{\boldsymbol{b}}$ for $n = 4$

The system is

$$\begin{pmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} \iff \begin{aligned} &u_{11}x_1 + u_{12}x_2 + u_{13}x_3 + u_{14}x_4 = b_1 \\ &u_{22}x_2 + u_{23}x_3 + u_{24}x_4 = b_2, \\ &u_{33}x_3 + u_{34}x_4 = b_3, \\ &u_{44}x_4 = b_4. \end{aligned}$$

We can just solve step-by-step:

$$x_4 = \frac{b_4}{u_4}, \quad x_3 = \frac{b_3 - u_{34}x_4}{u_{33}}, \quad x_2 = \frac{b_2 - u_{23}x_3 - u_{24}x_4}{u_{22}}, \quad x_1 = \frac{b_1 - u_{12}x_2 - u_{13}x_3 - u_{14}x_4}{u_{11}}.$$

Since $\det \mathrm{U} = \prod_{i=1}^{4} u_{ii} \neq 0$ when a solution exists, we know that $u_{11}, u_{22}, u_{33}, u_{44}$ are all non-zero, hence the formula for x_i are well-defined.

In general,

Theorem 1.14

An upper-triangular system $\mathrm{U}\vec{\boldsymbol{x}} = \vec{\boldsymbol{b}}$ *may be solved by backward substitution*

$$x_j = \frac{b_j - \sum_{k=j+1}^{n} u_{jk}x_k}{u_{jj}}, \quad j = n, \ldots, 1$$

§2.2 Elimination Algorithms

If our matrix A is not triangular, we can try to transform it to triangular form.

Elimination methods for solving linear systems involve three elementary row operations on the matrix equation version of a linear system in order to transform the system to an upper triangular form $U\vec{x} = \vec{y}$. The three operations are:

(a) multiply any row by a constant

(b) add any pair of rows

(c) rearrange the order of the rows

where ''row'' means the entire equation represented by the row of the matrix A and the corresponding components of the vectors $\vec{x}$ and $\vec{b}$. Linear algebra theorem tells us that these operations won't change the solution $\vec{x}$, but they will change the matrix A and the right-hand side $\vec{b}$.

2.2.1 Gaussian Elimination

The first kind of elimination we'll consider has been known for a very long time. For instance, it appears in Chapter 8 of a Chinese math text ''The Nine Chapters on the Mathematical Art'' dating to at least 10 BC, and authored by generations of now anonymous scholars. The same method was later ''invented'' independently and published in 1809 by the German mathematician and astronomer Carl Friedrich Gauss. Since the history of mathematics is essentially ''a fable agreed upon'' (as Napoleon might have said*), Gauss gets his name on this method.

The first stage of Gaussian elimination is to convert the matrix into upper-triangular form (all zeroes below the diagonal) like the one in (2). This process is sometimes called ''forward elimination'' because it ''eliminates'' (turns to zero) entries under the diagonal (i.e., the ''forward'' part of the matrix).

Forward Elimination

Here's an example that doesn't already have an upper-triangular matrix.

$$\begin{bmatrix} 3 & 0 & 1 \\ 0 & -1 & 2 \\ 2 & -2 & 4 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 6 \\ 4 \\ 10 \end{bmatrix}$$

add $-\frac{2}{3}$ times first row to third row

$$\begin{bmatrix} 3 & 0 & 1 \\ 0 & -1 & 2 \\ 0 & -2 & \frac{10}{3} \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 6 \\ 4 \\ 6 \end{bmatrix}$$

add -2 times second row to third row

$$\begin{bmatrix} 3 & 0 & 1 \\ 0 & -1 & 2 \\ 0 & 0 & -\frac{2}{3} \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 6 \\ 4 \\ -2 \end{bmatrix}$$

The second stage of Gaussian elimination is to use ''back-substitution'' to solve for the variables.

*or not, that's also a fable we agree upon.

Back-substitution

Now the system is upper triangular, and we can follow up with back-substitution similar to last section. The general idea is to work our way up from the bottom row; solving the equations represented by each row as we go.

Since the vector $\vec{x}$ doesn't change during the process of *forward elimination*, we can encode the process on the matrix $[\mathrm{A}|\vec{b}]$ where A is **augmented** by the vector $\vec{b}$. For our example above, the augmented matrix is:

$$[\mathrm{A}|\vec{b}] = \left[\begin{array}{ccc|c} 3 & 0 & 1 & 6 \\ 0 & -1 & 2 & 4 \\ 2 & -2 & 4 & 10 \end{array}\right] \tag{2.2}$$

Algorithm for Gaussian Elimination

Let $\mathrm{A}^{(1)} = \mathrm{A}$ and $\vec{b}^{(1)} = \vec{b}$. Then for each k from 1 to $n-1$, compute a new matrix $\mathrm{A}^{(k+1)}$ and right-hand side $b^{(k+1)}$ by the following procedure:

Step 1. Define the row multipliers

$$m_{ik} = \frac{a_{ik}^{(k)}}{a_{kk}^{(k)}}, \quad i = k+1, \ldots, n.$$

Step 2. Use these to remove the unknown x_k from equations $k+1$ to n, leaving

$$a_{ij}^{(k+1)} = a_{ij}^{(k)} - m_{ik} a_{kj}^{(k)}, \quad b_i^{(k+1)} = b_i^{(k)} - m_{ik} b_k^{(k)}, \quad i, j = k+1, \ldots, n.$$

The final matrix $\mathrm{A}^{(n)} = \mathrm{U}$ will then be upper triangular.

Step 3. The upper-triangular system $\mathrm{A}^{(n)}\vec{x} = \vec{b}^{(n)}$ may be solved by *backward substitution*

$$x_j = \frac{b_j^{(n)} - \sum_{k=j+1}^{n} a_{jk}^{(n)} x_k}{a_{jj}^{(n)}}, \quad j = n, n-1, n-2, \ldots, 1$$

Note that this procedure will work provided $a_{kk}^{(k)} \neq 0$ for every k. We will worry about this later (**??**). This will involve the 'swapping' elementary row operation.

2.2.2 Gauss-Jordan elimination

This method gets its name because it follows Gaussian elimination through the forward elimination stage, but then - instead of back-substitution - uses elementary row operations to complete ''reverse elimination'' until the matrix is transformed into zeroes above the diagonal too. As a final step, Gauss-Jordan elimination divides through to get ones along the diagonal, so the solution is simply the resulting right-side target vector. A German geodesist (studying measurements of the earth) named Wilhelm Jordan came up with this twist in 1887 (the same year Clasen published the same idea!), and it has the nice property that the

inverse matrix A^{-1} can be explicitly computed via the process by applying exactly the same operations on the matrix [A|I] where A is augmented by the $n \times n$ identity matrix I. [†]

We'll show how this works by revisiting the upper triangular matrix (2.2) we already generated by forward elimination in the preceding section.

Reverse Elimination

$$\begin{bmatrix} 3 & 0 & 1 \\ 0 & -1 & 2 \\ 0 & 0 & -\frac{2}{3} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 6 \\ 4 \\ -2 \end{bmatrix}$$

add $\frac{3}{2}$ times the last row to the first

$$\begin{bmatrix} 3 & 0 & 0 \\ 0 & -1 & 2 \\ 0 & 0 & -\frac{2}{3} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \\ -2 \end{bmatrix}$$

add 3 times the last row to the second

$$\begin{bmatrix} 3 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -\frac{2}{3} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ -2 \\ -2 \end{bmatrix}$$

Divide to get ones on the diagonal

$$\begin{bmatrix} 3 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -\frac{2}{3} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ -2 \\ -2 \end{bmatrix}$$

divide first row by 3 , second by -1, and third by $-\frac{2}{3}$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

From here, it is trivial to generate the solution $x_1 = 1, x_2 = 2$, and $x_3 = 3$.

2.2.3 Comparing Computational Effort

Since elimination methods take a finite number of steps to solve a system of linear equations, they are often compared by counting the individual operations needed to solve a general $n \times n$ system (n equations for n unknowns). In general, the amount of time required to perform a multiplication or division on a computer is approximately the same and is considerably greater than that required to perform an addition or subtraction. So we have a couple of different ways of computing the number of operations.

(a) total individual operations (multiplications/divisions and additions/subtractions of numbers).

(b) total individual multiplications/divisions only

(c) total individual additions/subtractions only

[†] There is another description of Gauss-Jordan elimination where reverse elimination is performed on each column at the same time as forward elimination, however this is less efficient than the version we're using here.

Note: We do not count additions/subtractions or multiplications/divisions of zeroes created along the steps of the process.

Example 2.15: Total number of operations required for backward substitution.

Consider each x_j. We have

$$\begin{aligned}
&j = n \to 1 \text{ division} \\
&j = n-1 \to 1 \text{ division } + [1 \text{ subtraction } + 1 \text{ multiplication }] \\
&j = n-2 \to 1 \text{ division } + 2 \times [1 \text{ subtraction } + 1 \text{ multiplication }] \\
&\vdots \\
&j = 1 \to 1 \text{ division } + (n-1) \times [1 \text{ subtraction } + 1 \text{ multiplication }]
\end{aligned}$$

So the total number of operations required is

$$\sum_{j=1}^{n} (1 + 2(n-j)) = n^2$$

So solving a triangular system by backward substitution takes n^2 operations.

Note:

- We say that the computational complexity of the algorithm is n^2.
- In practice, this is only a rough estimate of the computational cost, because reading from and writing to the computer's memory also take time. This can be estimated given a "memory model", but this depends on the particular computer.

Question 18. Group

Evaluate how many additions/subtractions are necessary to perform Gauss-Jordan elimination on a general $n \times n$ linear system. You should break your work into two steps:

(a) Find the number of additions/subtractions for forward elimination.

(b) Find the number of additions/subtractions for reverse elimination.

§2.3 Pivoting

Gaussian elimination will fail if we ever hit a zero on the diagonal. But this does not mean that the matrix A is singular. Consider the following example,

Example 3.16

The system

$$\begin{pmatrix} 0 & 3 & 0 \\ 2 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 3 \\ 2 \\ 1 \end{pmatrix}$$

obviously has solution $x_1 = x_2 = x_3 = 1$ (the matrix has determinant -6). But Gaussian elimination will fail because $a_{11}^{(1)} = 0$, so we cannot calculate m_{21} and m_{31}. However, we could avoid the problem by changing the order of the equations to get the equivalent system

$$\begin{pmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 3 \\ 1 \end{pmatrix}.$$

Now there is no problem with Gaussian elimination (actually the matrix is already upper triangular). Alternatively, we could have rescued Gaussian elimination by swapping columns:

$$\begin{pmatrix} 3 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} x_2 \\ x_1 \\ x_3 \end{pmatrix} = \begin{pmatrix} 3 \\ 2 \\ 1 \end{pmatrix}.$$

Swapping rows or columns is called pivoting. It is needed if the ''pivot'' element is zero, as in the above example. But it is also used to reduce rounding error.

Question 19.

Apply Gaussian elimination to the system

$$0.003x_1 + 59.14x_2 = 59.17$$
$$5.291x_1 - 6.130x_2 = 46.78$$

using the program you wrote for question 21.

Above example shows how difficulties can arise when the pivot element $a_{kk}^{(k)}$ is small relative to the entries $a_{ij}^{(k)}$, for $k \le i \le n$ and $k \le j \le n$. To avoid this problem, the simplest strategy is to select an element in the same column that is below the diagonal and has the largest absolute value; specifically, we determine the smallest $p \ge k$ such that

$$\left|a_{pk}^{(k)}\right| = \max_{k \le i \le n} \left|a_{ik}^{(k)}\right|$$

and exchange the kth and the pth row. This method, known as **partial pivoting**, dramatically improves the stability of Gaussian elimination.

Question 20.

Apply Gaussian elimination with partial pivoting to the system

$$0.003x_1 + 59.14x_2 = 59.17$$
$$5.291x_1 - 6.130x_2 = 46.78$$

using the program you wrote for question 21.

Note:

- Gaussian elimination without pivoting is unstable: rounding errors can accumulate.
- The ultimate accuracy is obtained by full pivoting, where both the rows and columns are swapped to bring the largest element possible to the diagonal.
- If it is not possible to rearrange the columns or rows to remove a zero from position $a_{kk}^{(k)}$ then A is singular.

§2.4 Lab Assignment 2: Elimination Methods

Question 21.

Write a function or script that will solve a linear systems of any size (say, up to $n = 500$) by Gaussian elimination.

- *The program should have an input parameter called* `method` *or something similar which has three possible values. Include (perhaps by an* `if - else` *structure) the ability to switch between the following three possibilities.*
  - *If method is 0, the algorithm should not be using pivoting. Make sure to catch any division by zero in such situations and display an error.*
  - *If method is 1, pivoting is done only when the pivot element is zero.*
  - *If method is 2, partial pivoting is performed at each step.*

Question 22.

Solve the augmented matrix system $A\vec{x} = \vec{b}$ below analytically (by hand).

$$A = \begin{pmatrix} \varepsilon & 1 & 2 \\ 2\varepsilon & 2 & -3 \\ 1 & 1 & 0 \end{pmatrix}, \vec{b} = \begin{pmatrix} 5 \\ -4 \\ 2 \end{pmatrix}$$

Question 23.

Use your code to solve the matrix system from question 22 for $\varepsilon = 0.001$, with and without partial pivoting. Calculate the error vector in your computed answer, by subtracting the computed solution from the exact solution obtained in the last question. To get a scalar error value from the error vector, we can compute its 2-norm, the Euclidean distance from the origin in n-dimensional space. This can be done via `numpy.linalg.norm` *in Python or* `norm` *in Octave.*

Question 24.

Regardless of the methodology used, some systems of equations are very difficult to solve with a high degree of accuracy, due to properties of the coefficient matrix. One example is a Hilbert matrix, whose entries are given by the formula

$$H_{ij} = \frac{1}{i + j - 1}.$$

(a) Find the (scalar) error associated with solving a system involving the 6×6 Hilbert matrix. For simplicity, set up your system so that the exact solution is $[1\,2\,3\,4\,5\,6]^T$, computing the associated vector $\vec{b}$.

(b) Does partial pivoting help in this example?

Question 25.

Hilbert matrices are known to be difficult to work with in numerical linear algebra settings, in that they induce a high degree of error in floating-point computations. The condition number is one measure of how "bad" a matrix is in this aspect. For a square matrix A, we can define the condition number $\kappa_*(A)$ as $\|A\|_*\|A^{-1}\|_*$, for some matrix norm $\|\cdot\|_*$.

(a) Compute the condition number κ of the coefficient matrix from question 22 and also for the Hilbert matrix.

See here for Python code, here for Octave code, and here for Mathematica. The default is the l^2-norm (see theorem 25).

(b) Comment briefly on the relationship between the condition number and the observed levels of error in each case.

§2.5 Norms of Vectors and Matrices

2.5.1 Vector Norm

To measure the error when the solution is a vector, as opposed to a scalar, we usually summarize the error in a single number called a norm.

Definition 5.17

vector norm on $\mathbb{R}^n$ is a real-valued function that satisfies

$$\|x+y\| \le \|x\| + \|y\| \quad \text{for every } x, y \in \mathbb{R}^n, \tag{N1}$$
$$\|\alpha x\| = |\alpha| \|x\| \quad \text{for every } x \in \mathbb{R}^n \text{ and every } \alpha \in \mathbb{R}, \tag{N2}$$
$$\|x\| \ge 0 \quad \text{for every } x \in \mathbb{R}^n \text{ and } \|x\| = 0 \implies x = 0. \tag{N3}$$

Property (N1) is called *the triangle inequality.*

Example 5.18

There are three common examples:

(a) The ℓ_2-norm

$$\|x\|_2 := \sqrt{\sum_{k=1}^{n} x_k^2} = \sqrt{x^{\mathrm{T}} x}.$$

This is just the usual Euclidean length of $\vec{x}$.

(b) The ℓ_1-norm

$$\|x\|_1 := \sum_{k=1}^{n} |x_k|$$

This is sometimes known as the taxicab or Manhattan norm, because it corresponds to the distance that a taxi has to drive on a rectangular grid of streets to get to $x \in \mathbb{R}^2$.

(c) The $\ell_\infty - norm$

$$\|x\|_\infty := \max_{k=1,\dots,n} |x_k|$$

This is sometimes known as the maximum norm.

The norms in the example above are all special cases of the ℓ_p-norm,

$$\|x\|_p = \left(\sum_{k=1}^{n} |x_k|^p \right)^{1/p}$$

which is a norm for any real number $p \ge 1$. Increasing p means that more and more emphasis is given to the maximum element $|x_k|$.

Question 26.

Consider the vectors $\vec{a} = (1,-2,3)^T$, $\vec{b} = (2,0,-1)^T$, *and* $\vec{c} = (0,1,4)^T$. *Evaluate their* ℓ_1-, ℓ_2-, *and* $\ell_\infty-$*norms.*

You should be able to observe that for the same vector $\vec{x}$, the norms satisfy the ordering $\|\vec{x}\|_1 \geq \|\vec{x}\|_2 \geq \|\vec{x}\|_\infty$, but two vectors may be ordered differently by different norms.

Question 27.

Sketch the "unit circles" $\left\{\vec{x} \in \mathbb{R}^2 : \|\vec{x}\|_p = 1\right\}$ *for* $p = 1, 2, \infty$.

2.5.2 Matrix Norm

We also use norms to measure the "size" of matrices. Since the set $\mathbb{R}^{n\times n}$ of $n \times n$ matrices with real entries is a vector space, we could just use a vector norm on this space. But usually we add an additional axiom.

Definition 5.19

A matrix norm is a real-valued function $\|\cdot\|$ on $\mathbb{R}^{n\times n}$ that satisfies:

$$\|A + B\| \leq \|A\| + \|B\| \quad \text{for every } A, B \in \mathbb{R}^{n\times n} \tag{M1}$$
$$\|\alpha A\| = |\alpha|\|A\| \quad \text{for every } A \in \mathbb{R}^{n\times n} \text{ and every } \alpha \in \mathbb{R}, \tag{M2}$$
$$\|A\| \geq 0 \quad \text{for every } A \in \mathbb{R}^{n\times n} \text{ and } \|A\| = 0 \Rightarrow A = 0, \tag{M3}$$
$$\|AB\| \leq \|A\|\|B\| \quad \text{for every } A, B \in \mathbb{R}^{n\times n}. \tag{M4}$$

The new axiom (M4) is called consistency. We usually want this additional axiom because matrices are more than just vectors. Some books call this a submultiplicative norm and define a "matrix norm" to satisfy just (M1), (M2), (M3), perhaps because (M4) only works for square matrices.

Note: If we treat a matrix as a big vector with n^2 components, then the ℓ_2-norm is called the Frobenius norm of the matrix:

$$\|A\|_F = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij}^2}$$

This norm is rarely used in numerical analysis because it is not induced by any vector norm (as we are about to define).

The most important matrix norms are so-called induced or *operator norms*. Remember that A is a linear map on $\mathbb{R}^n$, meaning that it maps every vector to another vector. So we can measure the size of A by how much it can stretch vectors with respect to a given vector norm.

Definition 5.20

if $\|\cdot\|_p$ is a vector norm, then the induced or operator norm is defined as

$$\|A\|_p := \sup_{\vec{x}\neq 0} \frac{\|A\vec{x}\|_p}{\|\vec{x}\|_p} = \max_{\|\vec{x}\|_p = 1} \|A\vec{x}\|_p.$$

Question 28.

Use property (M2) to prove that the two definitions are equivalent.

Example 5.21

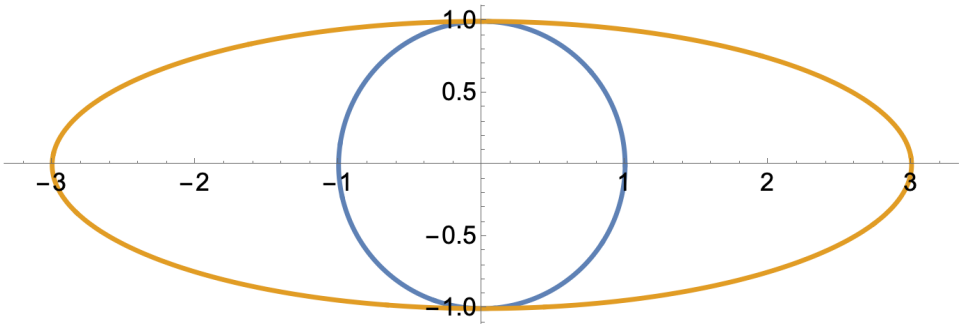
Let

$$A = \begin{pmatrix} 0 & 3 \\ 1 & 0 \end{pmatrix}.$$

In the ℓ_2-norm, a unit vector in $\mathbb{R}^2$ has the form $\vec{x} = (\cos\theta, \sin\theta)^T$, so the image of the unit circle is

$$A\vec{x} = \begin{pmatrix} 3\sin\theta \\ \cos\theta \end{pmatrix}.$$

This is illustrated below:



The induced matrix norm is the maximum stretching of this unit circle, which is

$$\|A\|_2 = \max_{\|\vec{x}\|_2=1} \|A\vec{x}\|_2 = \max_\theta \left(9\sin^2\theta + \cos^2\theta\right)^{1/2} = \max_\theta \left(1 + 8\sin^2\theta\right)^{1/2} = 3.$$

Theorem 5.22

The induced norm corresponding to any vector norm is a matrix norm, and the two norms satisfy $\|A\vec{x}\| \le \|A\|\|\vec{x}\|$ *for any matrix* $A \in \mathbb{R}^{n\times n}$ *and any vector* $\vec{x} \in \mathbb{R}^n$.

Properties (M1)-(M3) follow from the fact that the vector norm satisfies (N1)-(N3).

Question 29.

Prove (M4) using the following steps:

(a) Show that for any vector $\vec{y} \in \mathbb{R}^n$, *we have* $\|A\vec{y}\| \le \|A\|\|\vec{y}\|$.

(b) Taking $\vec{y} = B\vec{x}$ *for some* $\vec{x}$ *with* $\|\vec{x}\| = 1$, *show that*

$$\|AB\| \le \|A\|\|B\|.$$

It is cumbersome to compute the induced norms from their definition, but fortunately there are some very useful alternative formulae.

Theorem 5.23

The matrix norms induced by the ℓ_1-norm and ℓ_∞-norm satisfy

$$\|A\|_1 = \max_{1\le j\le n} \sum_{i=1}^{n} |a_{ij}|, \quad \textit{(maximum column sum)}$$

$$\|A\|_\infty = \max_{1\le j\le n} \sum_{j=1}^{n} |a_{ij}|. \quad \textit{(maximum row sum)}$$

Proof of theorem 23.

We will prove the result for the ℓ_1-norm first. Starting from the definition of the ℓ_1 vector norm, we have

$$\|A\vec{x}\|_1 = \sum_{i=1}^{n} \left| \sum_{j=1}^{n} a_{ij}x_j \right| \le \sum_{i=1}^{n}\sum_{j=1}^{n} |a_{ij}||x_j| = \sum_{j=1}^{n} |x_j| \sum_{i=1}^{n} |a_{ij}|.$$

If we let

$$c = \max_{j=1,\dots,n} \sum_{i=1}^{n} |a_{ij}|$$

then

$$\|A\vec{x}\|_1 \le c\|\vec{x}\|_1 \implies \|A\|_1 \le c. \tag{2.3}$$

Now let m be the column where the maximum sum is attained. If we choose y to be the vector with components $y_k = \delta_{km}$, then we have $\|A\vec{y}\|_1 = c$. Since $\|\vec{y}\|_1 = 1$, we must have that

$$\max_{\|x\|_1=1} \|Ax\|_1 \ge \|Ay\|_1 = c \implies \|A\|_1 \ge c. \tag{2.4}$$

The only way to satisfy both eq. (2.3) and eq. (2.4) is if $\|A\|_1 = c$. ■

Question 30.

Prove theorem 23 for the ℓ_∞-norm.

Question 31.

Find $\|A\|_1$ and $\|A\|_\infty$ for the matrix

$$A = \begin{pmatrix} -7 & 3 & -1 \\ 2 & 4 & 5 \\ -4 & 6 & 0 \end{pmatrix}$$

2.5.3 Eigenvalues and Spectral Radius

What about the matrix norm induced by the ℓ_2-norm? This turns out to be related to the eigenvalues of A. Recall that $\lambda \in \mathbb{C}$ is an eigenvalue of A with associated eigenvector $\vec{u}$ if $A\vec{u} = \lambda\vec{u}$.

Definition 5.24

The *spectral radius* $\rho(A)$ of A is the maximum $|\lambda|$ over all eigenvalues λ of A.

Theorem 5.25

The matrix norm induced by the ℓ_2-norm satisfies

$$\|A\|_2 = \sqrt{\rho(A^T A)}.$$

As a result the ℓ_2-norm is sometimes known as the spectral norm.

Example 5.26

For our matrix

$$A = \begin{pmatrix} 0 & 3 \\ 1 & 0 \end{pmatrix},$$

we have

$$A^T A = \begin{pmatrix} 0 & 1 \\ 3 & 0 \end{pmatrix}\begin{pmatrix} 0 & 3 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 9 \end{pmatrix}.$$

We see that the eigenvalues of $A^T A$ are $\lambda = 1, 9$, so $\|A\|_2 = \sqrt{9} = 3$ (as we calculated earlier).

Theorem 5.27

If A is an $n \times n$ matrix, then $\rho(A) \leq \|A\|$ for any induced norm $\|.\|$.

Proof of theorem 27.

If $A\vec{x} = \lambda\vec{x}$ and $\|\vec{x}\| = 1$, then $|\lambda| = \|A\vec{x}\| \leq \|A\|$. ■

We include the proof of theorem 25 for the sake of completion, but you can skip it for the purpose of the current course.

Proof of theorem 25.

We want to show that

$$\max_{\|x\|_2=1} \|Ax\|_2 = \max\left\{\sqrt{|\lambda|} : \lambda \text{ eigenvalue of } A^T A\right\}.$$

For A real, $A^T A$ is symmetric, so has real eigenvalues $\lambda_1 \le \lambda_2 \le \ldots \le \lambda_n$ with corresponding orthonormal eigenvectors $\vec{u}_1, \ldots, \vec{u}_n$ in $\mathbb{R}^n$. (Orthonormal means that $\vec{u}_j^T \vec{u}_k = \delta_{jk}$.) Note also that all of the eigenvalues are non-negative, since

$$A^T A \vec{u}_1 = \lambda_1 \vec{u}_1 \quad \Longrightarrow \lambda_1 = \frac{\vec{u}_1^T A^T A \vec{u}_1}{\vec{u}_1^T \vec{u}_1} = \frac{\|A\vec{u}_1\|_2^2}{\|\vec{u}_1\|_2^2} \ge 0.$$

So we want to show that $\|A\|_2 = \sqrt{\lambda_n}$. The eigenvectors form a basis, so every vector $\vec{x} \in \mathbb{R}^n$ can be expressed as a linear combination $\vec{x} = \sum_{k=1}^n \alpha_k \vec{u}_k$. Therefore

$$\|A\vec{x}\|_2^2 = \vec{x}^T A^T A \vec{x} = \vec{x}^T \sum_{k=1}^n \alpha_k \lambda_k \vec{u}_k = \sum_{j=1}^n \alpha_j \vec{u}_j^T \sum_{k=1}^n \alpha_k \lambda_k \vec{u}_k = \sum_{k=1}^n \alpha_k^2 \lambda_k,$$

where the last step uses orthonormality of the $\vec{u}_k$. It follows that

$$\|A\vec{x}\|_2^2 \le \lambda_n \sum_{k=1}^n \alpha_k^2.$$

But if $\|x\|_2 = 1$, then $\|x\|_2^2 = \sum_{k=1}^n \alpha_k^2 = 1$, so $\|Ax\|_2^2 \le \lambda_n$. To show that the maximum of $\|Ax\|_2^2$ is equal to λ_n, we can choose x to be the corresponding eigenvector $x = u_n$. In that case, $\alpha_1 = \ldots = \alpha_{n-1} = 0$ and $\alpha_n = 1$, so $\|A\vec{x}\|_2^2 = \lambda_n$. ■

§2.6 Iterative Methods

So far, we have studied elimination methods for solving linear systems $A\vec{x} = \vec{b}$, which converted the systems via elementary row operations until the solutions were easy to deduce. These solution methods are familiar from a course on Linear Algebra, but they can be very computationally expensive. In particular, some zero entries in the original matrix (which don't require storage or processing) may be replaced by non-zero entries as the method proceeds; necessitating more storage and processing at subsequent steps. This time, we'll consider a new kind of solution method for linear systems that avoids this problem by maintaining the original matrix structure.

The basic idea behind *iteration methods* for solving linear systems is to start with an initial approximation $\vec{x}^{(0)}$ to the solution $\vec{x}$ and generate a sequence of vectors

$$\left\{\vec{x}^{(0)}, \vec{x}^{(1)}, \vec{x}^{(2)} \ldots, \vec{x}^{(k)}, \ldots\right\}$$

that converges to $\vec{x}$. To generate the sequence, we'll use some iteration vector-function $g(\vec{x}) = T\vec{x} + \vec{c}$ where T is a (square) matrix and $\vec{c}$ is a vector (of the same dimension as $\vec{x}$) satisfying

$$\vec{x} = g(\vec{x}) \quad \Longleftrightarrow \quad A\vec{x} = \vec{b} \tag{2.5}$$

The iterative methods for linear system consist of starting with an initial approximation $\vec{x}^{(0)}$ and repeatedly calculating

$$\vec{x}^{(k+1)} = T\vec{x}^{(k)} + \vec{c}$$

until there is ''sufficiently small change'' between the input and the output vectors. For example, we can require that

$$\frac{\|\vec{x}^{(k+1)} - \vec{x}^{(k)}\|_*}{\|\vec{x}^{(k+1)}\|_*} \leq \varepsilon$$

for some prescribed error tolerance ε and some convenient norm $\|.\|_*$, usually the l_∞ norm.

Note: At this point, you might ask why should the sequence $\vec{x}^{(0)}, \vec{x}^{(1)}, \vec{x}^{(2)} \ldots, \vec{x}^{(k)}, \ldots$ converge? In fact, it may not always converge. We will come back to this question at the end of this section.

■ Question 32. □

Let D *be a diagonal matrix. Show that if*

$$T = D^{-1}(D - A) \quad \textit{and} \quad \vec{c} = D^{-1}\vec{b}, \tag{2.6}$$

then condition 2.5 is satisfied.

2.6.1 The Jacobi Method

The following version of an iteration method was named after the German mathematician Carl Gustav Jacob Jacobi who proposed a (more complicated) variant of it in the mid-1800's. The Jacobi method uses the formula from eq. (2.6) with the matrix D consisting of only the diagonal entries of the matrix A in the linear system:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & \cdot \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \implies D = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & \cdot \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix} \tag{2.7}$$

Note: Diagonal matrices are very easy to invert (as long as the diagonal entries a_{ii} of A are nonzero), the inverse D^{-1} in this case is the diagonal matrix whose entries are the inverses a_{ii}^{-1} of their counterparts in D.

2.6.2 Gauss-Seidel Method

The next iteration method we'll consider is usually credited to Gauss and another German mathematician, Philipp Ludwig von Seidel (though it is also sometimes credited to Liebmann, or simply called the method of "successive displacement"). The Gauss-Seidel method uses

$$T = -(L+D)^{-1}U \quad \text{and} \quad \vec{c} = (L+D)^{-1}\vec{b} \tag{2.8}$$

, in terms of the same matrix D as in the Jacobi method, and the upper (U) and lower (L) triangular parts of A left over when the diagonal D is removed:

$$U = \begin{bmatrix} 0 & a_{1,2} & \cdots & a_{1,n} \\ 0 & 0 & \cdots & a_{2,n} \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & a_{n-1,n} \\ 0 & 0 & \cdots & 0 \end{bmatrix} \quad L = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ a_{2,1} & 0 & \cdots & 0 \\ \cdot & \cdot & \cdots & \cdot \\ a_{n-1,1} & \cdot & \cdots & \cdot \\ a_{n,1} & \cdots & a_{n,n-1} & 0 \end{bmatrix} \tag{2.9}$$

■ **Question 33.** □

Show that eq. (2.8) gives an iteration vector-function satisfying condition 2.5.

2.6.3 Convergence Criteria

To study the convergence of general iteration techniques, we need to analyze the formula

$$\vec{x}^{(k+1)} = T\vec{x}^{(k)} + \vec{c}$$

where $\vec{x}^{(0)}$ is arbitrary. We will mention two necessary and sufficient conditions for the convergence, without proof.

Theorem 6.28

For any $\vec{x}^{(0)} \in \mathbb{R}^n$, the sequence $\left\{\vec{x}^{(k)}\right\}_{k=0}^{\infty}$ defined by

$$\vec{x}^{(k)} = T\vec{x}^{(k-1)} + \mathbf{c}, \quad \text{for each } k \geq 1,$$

converges to the unique solution of $\vec{\boldsymbol{x}} = \mathrm{T}\vec{\boldsymbol{x}} + \mathbf{c}$ *if and only if* $\rho(\mathrm{T}) < 1$.

Definition 6.29

A square matrix A is said to be *diagonally dominant* if

$$|a_{ii}| \geq \sum_{j \neq i} |a_{ij}| \quad \text{for all } i$$

where a_{ij} denotes the entry in the ith row and jth column.

If the inequality is strict, we say A is *strictly diagonally dominant*.

One can show that a strictly diagonally dominant is nonsingular. Gaussian elimination algorithm can be performed for such a matrix without any pivoting, and the computations remain stable with respect to the growth of round-off errors. The following theorem gives a sufficient (but not necessary) condition for convergence of Jacobi and Gauss-Seidel algorithms.

Theorem 6.30

If A *is strictly diagonally dominant, then for any choice of* $\vec{\boldsymbol{x}}^{(0)}$*, both the Jacobi and the Gauss-Seidel methods give sequences* $\left\{\vec{\boldsymbol{x}}^{(k)}\right\}_{k=0}^{\infty}$ *that converge to the unique solution of* $\mathrm{A}\vec{\boldsymbol{x}} = \vec{\boldsymbol{b}}$.

Question 34.

Show that $\vec{\boldsymbol{x}} - \vec{\boldsymbol{x}}^{(k)} = \mathrm{T}^k\left(\vec{\boldsymbol{x}} - \vec{\boldsymbol{x}}^{(0)}\right)$. *Conclude that* $\|\vec{\boldsymbol{x}} - \vec{\boldsymbol{x}}^{(k)}\| \to 0$ *if* $\|\mathrm{T}\| < 1$.

□

§2.7 Lab Assignment 3: Stationary Iterative Methods

Question 35.

Write a script or function that approximates the solution to the system $A\vec{x} = \vec{b}$ *using the Jacobi Method. The inputs should be an* $n \times n$ *matrix* A, *an* n*-dimensional vector* $\vec{b}$, *a starting vector* $\vec{x}_0$, *an error tolerance* ε, *and a maximum number of iterations* N. *The outputs should be either an approximate solution to the system* $A\vec{x} = \vec{b}$ *or an error message, along with the number of iterations completed. Additionally, it would be wise to build in functionality that allows you to optionally print the current estimated solution value at each iteration.*

Question 36.

Make a copy of your above code and modify it slightly, so that it approximates the solution using the Gauss-Seidel Method, with the same inputs and outputs.

Question 37.

Consider the system $\begin{pmatrix} 3 & 1 & -1 \\ 1 & 2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \vec{x} = \begin{pmatrix} 0 \\ 5 \\ -5 \end{pmatrix}$.

(a) *Solve the above system analytically, using methods from prior weeks.*

(b) *Why should the Jacobi method converge to the solution for the above system?*

(c) *Test your Jacobi code for the above problem, using several combinations (of your choosing) of starting vector* $\vec{x}_0$ *and tolerance* ε, *then report the results (final output and number of iterations.) Make sure that the maximum number of iterations* N *is sufficient to allow convergence, as it should eventually converge for all starting vectors.*

Question 38.

Consider the system $\begin{pmatrix} -1 & 1 & 2 \\ 6 & -1 & 5 \\ 68.5 & -28.5 & -1 \end{pmatrix} \vec{x} = \begin{pmatrix} 1 \\ 0 \\ -20.5 \end{pmatrix}$. *The coefficient matrix is not invertible, and this system has infinitely many solutions, including* $\begin{pmatrix} -4 \\ -9 \\ 3 \end{pmatrix}$. *The spectral radius of the iteration matrix* $D^{-1}(D - A)$ *used in the Jacobi method is* 1.

(a) *Test your Jacobi code using several combinations of starting vector* $\vec{x}_0$, *tolerance* ε, *and max number of iterations* N. *For each, report your inputs, whether it successfully converged, and if so, the number of iterations. If the Jacobi method failed, report the last few candidate vectors* $\vec{x}_j$ *before the iteration cap was reached.*

(b) *Repeat part (a) with your Gauss-Seidel code, for the same combinations.*

(c) *Compare the results of these two methods.*

Question 39.

Consider the system $\begin{pmatrix} 1 & 2 & -1 \\ 2 & 1 & 1 \\ -1 & 0 & 1 \end{pmatrix} \vec{x} = \begin{pmatrix} 3 \\ 0 \\ 3 \end{pmatrix}$, *whose solution is* $\begin{pmatrix} -2 \\ 3 \\ 1 \end{pmatrix}$.

(a) Explain why the Jacobi method should fail for this system. If this requires any matrix-related computations, you don't need to do these by hand.

(b) As in question 38.a, test your Jacobi code and report the results.

(c) Similarly, test your Gauss-Seidel code and report the results.

(d) Compare the methods. Though neither worked, does one appear to have performed worse than the other? How can you tell?

■ Question 40. □

Consider the system $\begin{pmatrix} 1 & \gamma - 1 \\ \gamma - 1 & 1 \end{pmatrix} \vec{x} = \begin{pmatrix} \gamma \\ \gamma \end{pmatrix}$, *whose solution is* $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$.

(a) Find the spectral radius of the iteration matrix $D^{-1}(D - A)$ *by analytic methods, e.g. via the characteristic polynomial, as learned in MATH 211.*

(b) Using the initial vector $\vec{x}_0 = \vec{0}$, *tolerance* $\varepsilon = 10^{-8}$, *and a maximum number of iterations* N *large enough to assure convergence, run your Jacobi code for parameter values* $\gamma = 2^{-2}$, $2^{-4}, 2^{-6}$, *and* 2^{-8}. *Based on the numbers of iterations required for each to converge, make a prediction for the number of iterations needed for* $\gamma = 2^{-10}$, *then test that prediction.*

Appendices

§A First Appendix

Exercises

§B Weekly Exercises

Resources

[1] Lloyd N. Trefethen. ‘‘The Definition of Numerical Analysis’’. In: **Bulletin of the Institute for Mathematics and Applications** (1993). URL: https://webs.um.es/eliseo/um/uploads/Main/TrefethendefNA.pdf.