Introduction to Numerical Analysis
Lecture Notes

Fall 2026

Changhong Mou

Department of Mathematics and Statistics
Utah State University

changhong.mou@usu.edu

Contents

List of Algorithms

List of Figures

Chapter 1

Direct Methods for Linear Systems: LU Factorization

This first unit develops the standard direct method of numerical linear algebra: the solution of $Ax = b$ by Gaussian elimination, reinterpreted as the matrix factorization $A = LU$ and stabilized by partial pivoting. The factorization built here powers nearly every algorithm in the rest of the course.

1.1 Introduction and LU Factorization

This opening lecture frames what numerical analysis is about and why it is a subject in its own right. Almost no problem of continuous mathematics can be solved exactly on a computer: inputs are measured or rounded, arithmetic itself is inexact, and many algorithms must be stopped after finitely many steps. We organize these issues into three recurring themes: conditioning of the problem, stability of the algorithm, and convergence of iterative methods. These themes guide the entire course. We then begin our first major topic, the solution of linear systems $Ax = b$, by reinterpreting Gaussian elimination as a matrix factorization $A = LU$ and introducing the Frobenius matrices from which the factorization is built.

1.1.1 General Remarks

Before any algorithms, we fix the vocabulary. The definitions below distinguish the mathematical problem from the numerical method that approximates it, and they name the three sources of error that the course studies in turn.

Definition 1.1. ***Numerical Analysis***
Numerical analysis is the study of algorithms for the problems of continuous mathematics (Trefethen).

Problem:

$$\text{Input} \;\rightarrow\; \text{Output: Solution(s) or No Solution}$$

Numerical Methods:

$$\text{(Approximate) Input} \;\rightarrow\; \text{(Approximate) Solution}$$

Sources of Errors:

1. Approximate input errors (cannot represent the data exactly, e.g. irrational numbers) $\Rightarrow$ *conditioning* of the problem. These problems are inherent to the problem.

2. Rounding errors $\Rightarrow$ *stability* of the algorithm

3. Approximation errors (i.e. terminating your algorithm when the result is "close enough") $\Rightarrow$ *convergence*

Example 1.2.
Two problems will occupy the first half of the course, and they illustrate the input–output picture above.

1. Systems of linear equations. The problem is

$$Ax = b, \qquad A \in \mathbb{C}^{n\times n},\ b \in \mathbb{C}^n,\ A \text{ nonsingular.}$$

   The input is the pair (A, b); the output is the unique solution $x = A^{-1}b$. Note that writing $A^{-1}b$ describes the *answer*, not the *method*: in practice we never form A^{-1}. The standard algorithm is LU factorization, which is modern Gaussian elimination, and it is the subject of this chapter.

2. Systems of nonlinear equations. Here the input is a function $f : D \to \mathbb{R}^n$ with $D \subset \mathbb{R}^n$, and the problem is to find the solutions

$$x \in D \quad \text{such that} \quad f(x) = 0.$$

   The linear problem is the special case $f(x) = Ax - b$. In general no finite algorithm produces the answer exactly, and the standard method, Newton's method (Chapter 2), is iterative: it generates a sequence of approximations, and we must decide when to stop. This is where the convergence question enters.

1.1.2 Introduction to LU Factorization

We begin with the first algorithm of the course. The idea is to view Gaussian elimination not as a sequence of row operations but as a factorization of the matrix A into structured factors. This point of view, standard since the work of the 1950s, is developed at length in Golub and Van Loan; the remark below states the goal, and the rest of the section introduces the matrices that accomplish it.

Remark 1.3. ***LU Factorization***
Given: (nonsingular) $A \in \mathbb{C}^{n\times n}$, $b \in \mathbb{C}^n$.
Goal: solve $Ax = b$.

Gaussian elimination:

$$Ax = b \;\leftrightarrow\; Ux = c, \qquad \text{where } U = \begin{bmatrix} * & * & \dots & * \\ 0 & * & \dots & * \\ \vdots & \ddots & \ddots & * \\ 0 & \dots & 0 & * \end{bmatrix}$$

i.e. U is upper-triangular. How do we do this?

$$L_{n-1}\dots L_2L_1A = U \;\Leftrightarrow\; A = \underbrace{L_1^{-1}\dots L_{n-2}^{-1}L_{n-1}^{-1}}_{L}\, U$$

$$L = \begin{bmatrix} 1 & 0 & \dots & 0 \\ * & 1 & \dots & 0 \\ \vdots & \ddots & \ddots & 0 \\ * & \dots & * & 1 \end{bmatrix}$$

i.e. L is lower-triangular.

Example 1.4. ***General LU Factorization Example***
Before doing arithmetic, it helps to watch the *shape* of the computation. Take $n = 4$ and follow which entries each elimination step touches. Applying L_1 zeros out the first column below the diagonal and updates the trailing 3×3 block; applying L_2 zeros out the second column below the diagonal and updates the trailing 2×2 block; and so on:

$$A = \begin{bmatrix} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{bmatrix} \xrightarrow{L_1} \begin{bmatrix} * & * & * & * \\ 0 & \# & \# & \# \\ 0 & \# & \# & \# \\ 0 & \# & \# & \# \end{bmatrix} = L_1A$$

$$\xrightarrow{L_2} \begin{bmatrix} * & * & * & * \\ 0 & * & * & * \\ 0 & 0 & \# & \# \\ 0 & 0 & \# & \# \end{bmatrix} = L_2L_1A \qquad \xrightarrow{L_3} \begin{bmatrix} * & * & * & * \\ 0 & * & * & * \\ 0 & 0 & * & * \\ 0 & 0 & 0 & \# \end{bmatrix} = L_3L_2L_1A = U$$

Here the $*$s represent entries unchanged by the most recent step, and the #s represent entries that were just modified. Two features are worth noticing:

each step works on a smaller trailing block than the one before, which is why the operation count comes out to $O(n^3)$ rather than $O(n^4)$, and each step is multiplication by a matrix L_i of a very particular form. These are the Frobenius matrices defined next.

Definition 1.5. ***Frobenius matrix***
`http://en.wikipedia.org/wiki/Frobenius_matrix`

A Frobenius matrix is a square matrix with the following properties:

- all entries on the main diagonal are ones
- the entries below the main diagonal of at most one column are arbitrary
- every other entry is zero

Frobenius matrices are invertible. The following is an example:

$$A = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & a_{32} & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & a_{n2} & 0 & \cdots & 1 \end{bmatrix}, \qquad A^{-1} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & -a_{32} & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & -a_{n2} & 0 & \cdots & 1 \end{bmatrix}$$

1.1.3 Filling in the Details: Why Frobenius Inverses Are So Simple

Write a Frobenius matrix acting on column k as

$$L_k = I + c\, e_k^T,$$

where e_k is the kth standard basis vector and $c \in \mathbb{C}^n$ collects the subdiagonal entries (so $c_j = 0$ for $j \leq k$).

Key observation: the kth entry of c is zero, so

$$e_k^T c = c_k = 0.$$

Now multiply:

$$\begin{aligned}(I + c\, e_k^T)(I - c\, e_k^T) &= I - c\, e_k^T + c\, e_k^T - c\, \underbrace{(e_k^T c)}_{=0}\, e_k^T \\ &= I.\end{aligned}$$

So the inverse is obtained by *flipping the signs* of the subdiagonal entries; no computation is needed. This is exactly what we saw in the definition: A^{-1} replaces a_{j2} by $-a_{j2}$.

Why this matters for LU: the elimination matrices $L_1, L_2, \ldots, L_{n-1}$ are all Frobenius, so

$$L = L_1^{-1} L_2^{-1} \cdots L_{n-1}^{-1}$$

is available *for free*: each factor just contributes its (sign-flipped) column of multipliers, and (as we will verify next lecture) the product assembles them into one unit lower-triangular matrix with no fill-in. Gaussian elimination therefore produces L and U simultaneously at no extra cost.

1.1.4 The Three Errors on One Example

Take $Ax = b$ solved by elimination on a computer:

1. **Conditioning:** the entries of A, b may come from measurements, or may not be exactly representable in binary. How much can the exact solution $A^{-1}b$ move when (A, b) move slightly? *Property of the problem.*

2. **Stability:** even with exact input, every arithmetic operation is rounded. Does the algorithm amplify these roundings? *Property of the algorithm.*

3. **Convergence:** direct methods like LU terminate exactly, but iterative methods (Newton, QR iteration, ...) must be stopped "when close enough." How fast do they approach the answer?

The first half of this course builds the vocabulary to make all three precise.

1.2 LU Factorization: Algorithm and Existence

In this lecture we turn the factorization idea of the previous lecture into a complete algorithm. Working through a 4×4 example in full detail shows how the multipliers assemble themselves, at no extra cost, into the unit lower triangular factor L while elimination produces U. We then state the algorithm in compact index notation, count its operations (roughly $\frac{2}{3}n^3$ floating-point operations), and record the important practical consequence that additional right-hand sides cost only $O(n^2)$ once the factorization is known. The lecture closes with a cautionary example: without row exchanges, LU factorization can fail outright, or, worse, succeed while destroying all accuracy.

1.2.1 LU Factorization (Continued)

This section carries out the factorization in full on a 4×4 example and then states the general algorithm together with its cost. Readers interested in the blocked variants used by high-performance libraries will find them in Golub and Van Loan; the version here is the textbook form, sufficient for the cost and stability questions that follow.

Example 1.6.
We now carry out the factorization in full on a concrete matrix, keeping track of every multiplier. Let

$$A = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}.$$

Step 1. The pivot is $a_{11} = 2$. To clear the first column we subtract $\frac{4}{2} = 2$ times row 1 from row 2, $\frac{8}{2} = 4$ times row 1 from row 3, and $\frac{6}{2} = 3$ times row 1 from row 4. Collecting the negated multipliers into a Frobenius matrix:

$$L_1 A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -2 & 1 & 0 & 0 \\ -4 & 0 & 1 & 0 \\ -3 & 0 & 0 & 1 \end{bmatrix} A = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 3 & 5 & 5 \\ 0 & 4 & 6 & 8 \end{bmatrix}$$

Step 2. The pivot is now the $(2,2)$ entry 1 of the updated matrix. The multipliers for rows 3 and 4 are $\frac{3}{1} = 3$ and $\frac{4}{1} = 4$:

$$L_2 L_1 A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -3 & 1 & 0 \\ 0 & -4 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 3 & 5 & 5 \\ 0 & 4 & 6 & 8 \end{bmatrix} = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 2 & 2 \\ 0 & 0 & 2 & 4 \end{bmatrix}$$

Step 3. One entry remains below the diagonal. The pivot is the $(3,3)$ entry 2, and the single multiplier is $\frac{2}{2} = 1$:

$$L_3 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 \end{bmatrix}$$

$$L_3L_2L_1A = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 2 & 2 \\ 0 & 0 & 0 & 2 \end{bmatrix} = U \quad \text{upper-triangular}$$

Assembling L. We have $L_3L_2L_1A = U$, so $A = L_1^{-1}L_2^{-1}L_3^{-1}U = LU$. Each inverse is obtained by flipping the signs of the multipliers, and the product simply deposits each column of multipliers into place, with no interaction between columns:

$$L = L_1^{-1}L_2^{-1}L_3^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 4 & 0 & 1 & 0 \\ 3 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 3 & 1 & 0 \\ 0 & 4 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 4 & 3 & 1 & 0 \\ 3 & 4 & 1 & 1 \end{bmatrix}$$

The result is unit lower-triangular, and its subdiagonal entries are exactly the multipliers $2, 4, 3, 3, 4, 1$ computed along the way. This is the practical content of the factorization: Gaussian elimination *is* the computation of L and U, and the multipliers need only be recorded, not recomputed.

Remark 1.7. ***Convention (Matlab Notation)***

$$A = [a_{jk}]_{j,k=1,2,\ldots,n} \in \mathbb{C}^{n\times n}$$

$$a_{j,k:n} = [\, a_{j,k} \;\; a_{j,k+1} \;\; \cdots \;\; a_{j,n} \,]$$

Algorithm 1 LU Factorization without Pivoting

Require: $A \in \mathbb{C}^{n\times n}$

Ensure: L unit lower-triangular, U upper-triangular, with $A = LU$

1: $U \leftarrow A, \quad L \leftarrow I$
2: **for** $k = 1, 2, \ldots, n-1$ **do**
3: **for** $j = k+1, k+2, \ldots, n$ **do**
4: $l_{jk} \leftarrow u_{jk}/u_{kk}$ ▷ potential problem if u_{kk} is zero or tiny
5: $u_{j,k:n} \leftarrow u_{j,k:n} - l_{jk}\, u_{k,k:n}$
6: **end for**
7: **end for**

Remark 1.8. ***Operation Count***

Additions: $\displaystyle\sum_{k=1}^{n-1}(n-k)(n-k+1) = \sum_{l=1}^{n-1} l(l+1) \approx \frac{n^3}{3}$ (where $l = n-k$)

Multiplications: $\approx \frac{n^3}{3}$
Divisions: $\approx n^2 \Rightarrow$ ignore (because it is a lower order term)

Total Work: $\approx \frac{2n^3}{3}$ flops

Definition 1.9. ***Flop***
A *flop* is a floating-point operation: addition, subtraction, multiplication, division, square root.

Remark 1.10. ***Solution of*** $Ax = b$

$$\underbrace{A}_{=LU} x = b$$
$$L \underbrace{Ux}_{=y} = b$$

$$Ly = b \tag{1}$$
$$Ux = y \tag{2}$$

$Ly = b$ is easily solved via forward substitution. $Ux = y$ is easily solved by backward substitution.

One triangular solve requires $\approx n^2$ flops.

1.2.2 Problems with LU Factorization without Pivoting

The algorithm of the previous section can fail in two different ways: the factorization may not exist, and, more subtly, it may exist yet be computed inaccurately. The two examples below illustrate both failures. The second is the more important, because nothing in exact arithmetic predicts it.
LU factorization without pivoting is unstable!

Example 1.11. ***Error from LU Factorization without Pivoting***
The following 2×2 example is the standard illustration, and it repays close reading: the matrix is harmless, the algorithm runs to completion, and the answer is still wrong. Consider

$$A = \begin{bmatrix} 10^{-20} & 1 \\ 1 & 1 \end{bmatrix}, \qquad b = \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

Nothing about this problem is pathological: A is well-conditioned, and its exact solution has entries of moderate size. First run the algorithm with exact arithmetic. The single multiplier is $l_{21} = 1/10^{-20} = 10^{20}$, so:

$$L = \begin{bmatrix} 1 & 0 \\ 10^{20} & 1 \end{bmatrix}, \qquad U = \begin{bmatrix} 10^{-20} & 1 \\ 0 & 1 - 10^{20} \end{bmatrix}$$

Now solve a system:

$$Ly = b \qquad y = \begin{bmatrix} 1 \\ -10^{20} \end{bmatrix}$$

$$Ux = y \qquad x = \frac{1}{1 - 10^{-20}} \begin{bmatrix} -1 \\ 1 \end{bmatrix} \approx \begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

So far, so good: in exact arithmetic the factorization is correct and the solution is recovered. Now repeat the computation in floating-point arithmetic, which carries roughly 16 significant digits. The multiplier 10^{20} is representable, but the entry $u_{22} = 1 - 10^{20}$ is not: the 1 is 20 orders of magnitude below 10^{20} and is rounded away entirely, leaving

$$\tilde{L} = L = \begin{bmatrix} 1 & 0 \\ 10^{20} & 1 \end{bmatrix}, \qquad \tilde{U} = \begin{bmatrix} 10^{-20} & 1 \\ 0 & -10^{20} \end{bmatrix}$$

$$\tilde{L}\tilde{U} = \begin{bmatrix} 10^{-20} & 1 \\ 1 & 0 \end{bmatrix} \not\approx A$$

The computed factors multiply back to a matrix whose $(2,2)$ entry is 0 instead of 1: a 100% error in that entry, even though every individual arithmetic operation was performed to 16 digits. Solving the system with these factors produces a bogus answer, as we verify at the start of the next lecture. The failure belongs to the algorithm, not the problem, and the remedy is pivoting.

1.2.3 Filling in the Details: The Operation Count

At elimination step k we update rows $j = k+1, \ldots, n$ (that is $n-k$ rows), and in each row the entries in columns $k, \ldots, n$ (that is $n-k+1$ entries). Each updated entry costs one multiplication and one subtraction:

$$u_{j,k:n} = u_{j,k:n} - l_{j,k}\, u_{k,k:n}.$$

So the number of additions (and likewise multiplications) is

$$\sum_{k=1}^{n-1} (n-k)(n-k+1) \overset{l:=n-k}{=} \sum_{l=1}^{n-1} l(l+1) = \underbrace{\sum_{l=1}^{n-1} l^2}_{\approx n^3/3} + \underbrace{\sum_{l=1}^{n-1} l}_{\approx n^2/2} \approx \frac{n^3}{3}.$$

(Recall $\sum_{l=1}^{m} l^2 = \frac{m(m+1)(2m+1)}{6} \approx \frac{m^3}{3}$.)

Adding up:

$$\underbrace{\tfrac{n^3}{3}}_{\text{additions}} + \underbrace{\tfrac{n^3}{3}}_{\text{multiplications}} + \underbrace{O(n^2)}_{\text{divisions } l_{jk}} \approx \frac{2n^3}{3} \text{ flops.}$$

The divisions ($n-k$ per step, $\approx n^2/2$ total) are a lower-order term, so we drop them.

Consequence for $Ax = b$: factor once ($\frac{2n^3}{3}$), then each right-hand side costs only two triangular solves ($2n^2$). Solving with a second right-hand side is nearly free compared to refactoring.

1.2.4 Reading the 10^{-20} Example Correctly

What exactly went wrong? Track the sizes:

$$l_{21} = \frac{1}{10^{-20}} = 10^{20}, \qquad u_{22} = 1 - 10^{20}.$$

In double precision ($\approx$ 16 digits), $1 - 10^{20}$ rounds to -10^{20}: the "1" is annihilated. The stored factors then satisfy

$$\tilde{L}\tilde{U} = \begin{bmatrix} 10^{-20} & 1 \\ 1 & 0 \end{bmatrix},$$

which differs from A in the $(2,2)$ entry by 1, a 100% error in that entry, even though *each individual rounding was correct to 16 digits.*

Moral: the algorithm created huge intermediate quantities (10^{20}) not present in the original problem. Information about A was destroyed on the way. Note A itself is perfectly well-conditioned here; the *algorithm*, not the problem, is at fault. Pivoting (next lecture) keeps all multipliers $|l_{jk}| \leq 1$ and repairs this.

1.3 Pivoting and $PA = LU$

The example that closed the previous lecture showed that the plain LU algorithm is unusable in general. The remedy is pivoting: exchanging rows so that the multipliers stay bounded. This lecture develops the bookkeeping needed to make that precise, namely permutation matrices and their properties, and shows that the row exchanges can be commuted past the elimination matrices, so that the final result is a factorization $PA = LU$ of

a row-permuted copy of A. We describe the partial pivoting strategy, verify that it guarantees $|l_{jk}| \leq 1$, and note that its cost, $O(n^2)$ comparisons, is negligible against the $O(n^3)$ arithmetic of the factorization itself.

1.3.1 Recap

We briefly restate where the previous lecture ended before continuing.

Example 1.12.
Continuing from last time, let us finish the computation and see the damage. Using the rounded factors $\tilde{L}$, $\tilde{U}$ from the previous lecture, forward substitution gives

$$\tilde{L}\tilde{y} = b \qquad \tilde{y} = \begin{bmatrix} 1 \\ -10^{20} \end{bmatrix}$$

and back substitution gives

$$\tilde{U}\tilde{x} = \tilde{y} \qquad \tilde{x} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \neq x \approx \begin{bmatrix} -1 \\ 1 \end{bmatrix}.$$

The first component is completely wrong: 0 instead of -1. The culprit is the huge multiplier 10^{20}, which the algorithm itself created out of a harmless matrix. Keeping the multipliers small is exactly what pivoting will accomplish.

1.3.2 Pivoting

Row exchanges repair both failures identified above. This section develops the machinery: permutation matrices to record the exchanges, the commutation argument that collapses the interleaved product into the clean form $PA = LU$, and the partial pivoting rule that decides which rows to exchange. Every production implementation of Gaussian elimination, including the one behind MATLAB's backslash operator, is the algorithm of this section.

$$A = \begin{bmatrix} 10^{-20} & 1 \\ 1 & 1 \end{bmatrix} \qquad P = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad \text{(permutation matrix)}$$

$$PA = \begin{bmatrix} 1 & 1 \\ 10^{-20} & 1 \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 0 \\ 10^{-20} & 1 \end{bmatrix}}_{=L} \underbrace{\begin{bmatrix} 1 & 1 \\ 0 & 1 - 10^{-20} \end{bmatrix}}_{=U}$$

Floating point arithmetic:

$$\tilde{L} = L \qquad \tilde{U} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

$$PAx = Pb \qquad LUx = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$\tilde{L}\tilde{y} = Pb \qquad \tilde{y} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$\tilde{U}\tilde{x} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \qquad \tilde{x} = \begin{bmatrix} -1 \\ 1 \end{bmatrix} \approx x = A^{-1}b$$

Every time you use Gaussian elimination you need to use pivoting.

Example 1.13. ***General Pivoting Example***
Here is the shape of one pivoted elimination step. Take $n = 4$ and suppose we are at step $k = 2$: the first column is already eliminated, and the boxed entry is the current pivot position. Partial pivoting first searches column 2 from the diagonal down for the entry of largest absolute value, say in row 4, and exchanges rows 2 and 4; only then does it eliminate:

$$U = \begin{bmatrix} * & * & * & * \\ 0 & \boxed{*} & * & * \\ 0 & * & * & * \\ 0 & * & * & * \end{bmatrix} \xrightarrow{P_2} \begin{bmatrix} * & * & * & * \\ 0 & \# & \# & \# \\ 0 & * & * & * \\ 0 & \# & \# & \# \end{bmatrix} \quad \text{exchange rows 2 \& 4}$$

$$\xrightarrow{L_2} \begin{bmatrix} * & * & * & * \\ 0 & * & * & * \\ 0 & 0 & \# & \# \\ 0 & 0 & \# & \# \end{bmatrix}$$

Doing this at every step interleaves permutations with eliminations. In general the process computes

$$L_{n-1}P_{n-1}\cdots L_3L_2P_2L_1P_1A = U,$$

where P_k exchanges row k with some row $i \geq k$. If no exchange is needed at step k, then $P_k = I$. At first sight this interleaved product looks like it has lost the clean form $A = LU$; the pleasant surprise, worked out below, is that the permutations can all be commuted to the right, past the eliminations, without changing their structure.

Example 1.14. ***Specific Pivoting Example***
We redo the matrix of Lecture 1.2, this time with partial pivoting. Recall

$$A = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}.$$

At step 1 the candidates for the pivot are the first-column entries $2, 4, 8, 6$; the largest in absolute value is 8, in row 3, so we exchange rows 1 and 3 before eliminating:

$$\underbrace{\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{P_1} A = \begin{bmatrix} 8 & 7 & 9 & 5 \\ 4 & 3 & 3 & 1 \\ 2 & 1 & 1 & 0 \\ 6 & 7 & 9 & 8 \end{bmatrix}$$

$$L_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -\frac{1}{2} & 1 & 0 & 0 \\ -\frac{1}{4} & 0 & 1 & 0 \\ -\frac{3}{4} & 0 & 0 & 1 \end{bmatrix}$$

$$L_1 P_1 A = \begin{bmatrix} 8 & 7 & 9 & 5 \\ 0 & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \\ 0 & -\frac{3}{4} & -\frac{5}{4} & -\frac{5}{4} \\ 0 & -\frac{1}{2} & -\frac{3}{2} & -\frac{3}{2} \end{bmatrix}$$

Notice that after the exchange every multiplier is a ratio of an entry to the largest entry in its column, so all multipliers satisfy $|l_{jk}| \leq 1$: here $\frac{1}{2}$, $\frac{1}{4}$, $\frac{3}{4}$ at step 1. Continuing in the same way (choosing the largest remaining entry in each column, exchanging, eliminating) yields

$$\vdots$$

$$U = \begin{bmatrix} 8 & 7 & 9 & 5 \\ 0 & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \\ 0 & 0 & -\frac{6}{7} & -\frac{2}{7} \\ 0 & 0 & 0 & \frac{2}{3} \end{bmatrix}.$$

Compare with the unpivoted factorization of the same matrix: the arithmetic is different, but the cost is the same, and the multipliers are now all bounded by 1.

Example 1.15. ***Pivoting in Matrix Form***
It remains to untangle the interleaved product into the form $PA = LU$. The trick is to insert identities $I = P^T P$ between the factors and regroup. For $n = 4$:

$$L_3 P_3 \underbrace{I}_{=P_3^T P_3} L_2 P_2 L_1 \underbrace{I}_{=P_2^T P_3^T P_3 P_2} P_1 A = U$$

$$\underbrace{L_3}_{=L_3'}\underbrace{(P_3L_2P_3^T)}_{=L_2'}\underbrace{(P_3P_2L_1P_2^TP_3^T)}_{=L_1'}\underbrace{(P_3P_2P_1)}_{=P}A = U$$

$$L_3'L_2'L_1'(PA) = U$$

L_1' ($\neq L_1^T$) has the same structure as L_1, with nontrivial elements in column 1 reordered.

$$L_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -\frac{1}{2} & 1 & 0 & 0 \\ -\frac{1}{4} & 0 & 1 & 0 \\ -\frac{3}{4} & 0 & 0 & 1 \end{bmatrix}, \qquad L_1' = \begin{bmatrix} 1 & 0 & 0 & 0 \\ * & 1 & 0 & 0 \\ * & 0 & 1 & 0 \\ * & 0 & 0 & 1 \end{bmatrix}$$

$$PA = \underbrace{(L_1')^{-1}(L_2')^{-1}(L_3')^{-1}}_{=L}U$$

In the general case the same regrouping gives

$$PA = LU,$$

where $P = P_{n-1}\cdots P_2P_1$ is a single permutation matrix, L is unit lower-triangular with $|l_{jk}| \leq 1$, and U is upper-triangular. In words: partial pivoting computes the ordinary LU factorization of a row-reordered copy of A, where the reordering is discovered during the elimination itself. Why the conjugated matrices L_i' keep the Frobenius structure is verified in the "Filling in the Details" section below.

Algorithm 2 LU Factorization with Partial Pivoting

Require: $A \in \mathbb{C}^{n\times n}$

Ensure: P permutation, L unit lower-triangular with $|l_{jk}| \le 1$, U upper-triangular, with $PA = LU$

$U \leftarrow A, \quad L \leftarrow I, \quad P \leftarrow I$
for $k = 1, 2, \ldots, n-1$ **do**
  choose $i \ge k$ such that $|u_{ik}| = \max_{k\le l\le n} |u_{lk}|$
  if $u_{ik} = 0$ **then**
    stop: A is singular
  end if
  $u_{k,k:n} \leftrightarrow u_{i,k:n}$ ▷ interchange rows i and k of U
  $l_{k,1:k-1} \leftrightarrow l_{i,1:k-1}$ ▷ carry the already-computed multipliers along
  $p_{k,:} \leftrightarrow p_{i,:}$ ▷ record the exchange in P
  for $j = k+1, k+2, \ldots, n$ **do**
    $l_{jk} \leftarrow u_{jk}/u_{kk}$
    $u_{j,k:n} \leftarrow u_{j,k:n} - l_{jk}\, u_{k,k:n}$
  end for
end for

1.3.3 Filling in the Details: Why $L_i' = PL_iP^T$ Keeps Its Shape

The reordering trick inserted $I = P^TP$ between factors. The conjugated matrices are, e.g.,

$$L_1' = (P_3P_2)\, L_1\, (P_3P_2)^T.$$

Why does this stay a Frobenius matrix for column 1? Write $L_1 = I + c\, e_1^T$ with c supported on entries $2, \ldots, n$, and let $\Pi := P_3P_2$. Both P_2 and P_3 only permute rows/columns *from index 2 onward* (step k pivots among rows $\ge k$), so $\Pi e_1 = e_1$. Then

$$\Pi L_1 \Pi^T = \Pi(I + c\, e_1^T)\Pi^T = I + (\Pi c)\, \underbrace{(\Pi e_1)^T}_{=e_1^T} = I + (\Pi c)\, e_1^T.$$

So L_1' is again unit lower-triangular with its column-1 multipliers merely *reordered* ($c \mapsto \Pi c$), exactly the structure claimed on the discussion above.

Permutation matrix facts used here:

- P has exactly one 1 in each row and column; $P^TP = I$, so $P^{-1} = P^T$.
- Left multiplication PA permutes *rows*; right multiplication AP^T permutes *columns* the same way.

- Products of permutation matrices are permutation matrices, so $P := P_{n-1} \cdots P_2 P_1$ in $PA = LU$ is one single row reordering of A, decided during elimination.

Cost of pivoting: step k scans $n - k + 1$ candidate pivots, so the total is $O(n^2)$ comparisons, negligible next to $\frac{2n^3}{3}$ flops. Stability is (essentially) free.

1.3.4 Why $|l_{jk}| \leq 1$, and Why We Care

At step k we choose the pivot row $i \geq k$ with the *largest* $|u_{ik}|$, then form

$$l_{jk} = \frac{u_{jk}}{u_{kk}} \qquad (j > k).$$

After the swap, $|u_{kk}| = \max_{l \geq k} |u_{lk}| \geq |u_{jk}|$, hence $|l_{jk}| \leq 1$ automatically.

This bounds $\|L\|$ by a modest constant and prevents the 10^{20}-type blow-up in L. (Whether U can still grow is the "growth factor" question we settle it in Chapter 4.)

Practical note: in MATLAB, `[L,U,P] = lu(A)` returns exactly the objects of today's algorithm, and `A\b` performs $PA = LU$ plus two triangular solves under the hood.

1.4 Notes and References

The factorization viewpoint on Gaussian elimination presented here follows the modern treatment; Trefethen and Bau develop it in the same spirit, and Golub and Van Loan give a comprehensive account that includes blocked and parallel variants used in production software. Deuflhard and Hohmann is the closest source for the overall organization of these notes. The LAPACK library implements the algorithms of this chapter; MATLAB's `lu` and backslash operator call it.

1.5 Exercises

1. Compute by hand the LU factorization of $A = \begin{bmatrix} 2 & 1 & 1 \\ 4 & 3 & 3 \\ 8 & 7 & 9 \end{bmatrix}$, recording the multipliers l_{jk} as you eliminate. Verify that $A = LU$ by multiplying your factors.

2. Prove that the product of two unit lower triangular matrices is unit lower triangular, and that the inverse of a unit lower triangular matrix is unit lower triangular. Conclude that the factor L in $A = LU$ is uniquely determined when A admits an LU factorization with nonsingular U.

3. Let $L_k = I + c\,e_k^T$ with $c_j = 0$ for $j \le k$. Verify the identity $(I + c\,e_k^T)(I - c\,e_k^T) = I$ for general n and k, filling in the step that uses $e_k^T c = 0$.

4. Count the floating-point operations in forward substitution $Ly = b$ and back substitution $Ux = y$, and show that each costs $n^2 + O(n)$ operations. Explain why solving $Ax = b_i$ for m right-hand sides costs about $\frac{2}{3}n^3 + 2mn^2$ operations in total.

5. Find the factorization $PA = LU$ with partial pivoting for $A = \begin{bmatrix} 0 & 1 & 2 \\ 1 & 2 & 1 \\ 2 & 7 & 8 \end{bmatrix}$. State P explicitly and check that all multipliers satisfy $|l_{jk}| \le 1$.

6. Using $PA = LU$, express $\det A$ in terms of the diagonal entries of U and the permutation P. Explain why this is the standard way determinants are computed in practice, and why the cofactor expansion is not.

7. (Computing.) Time `lu(A)` in MATLAB or `scipy.linalg.lu` in Python for random matrices of size $n = 500$, 1000, and 2000. Do the timings scale like n^3? Estimate the constant and compare with $\frac{2}{3}n^3$ flops divided by your machine's peak flop rate.

Chapter 2

Nonlinear Systems and Newton's Method

We now solve systems of nonlinear equations $f(x) = 0$. The central method is Newton's: linearize, solve a linear system, repeat. This unit derives the method, proves a complete local convergence theorem in affine-invariant form, and develops the safeguards (monotonicity tests, damping) that make it robust in practice.

2.1 Newton's Method: Derivation

With linear systems under control, we begin the second unit of the course: systems of nonlinear equations $f(x) = 0$. After a brief recap of the LU toolkit, which will supply the linear solves inside every nonlinear method, we derive Newton's method in one variable from the tangent-line approximation, the model example of a local linearization. Simple examples already reveal the method's character: quadratic convergence from good starting points, and cycling or divergence from bad ones. Quantifying both behaviors is the goal of the next several lectures.

2.1.1 LU Factorization Recap

We summarize the linear-algebra toolkit of Chapter 1, since the methods of this chapter will call on it in every iteration.

$$PA = LU$$

$$L = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ l_{21} & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ l_{n1} & \cdots & l_{n,n-1} & 1 \end{bmatrix}, \qquad |l_{jk}| \leq 1$$

LU factorization with partial pivoting is stable in practice.

2.1.2 Newton's Method

We turn to systems of nonlinear equations. There is no analogue here of the finite elimination algorithm; the fundamental idea is instead iteration: replace the nonlinear problem by a sequence of linear ones. Newton's method is the canonical such iteration, and the remainder of the chapter is devoted

to it. The derivation below treats one equation; the next lecture extends it to systems.

Newton's method is not a finite method, meaning that you would have to run it infinitely many times in order for it to arrive at the answer.

Remark 2.1. ***Newton's Method Overview***
Given: $f : D \to \mathbb{R}^n$, $D \subset \mathbb{R}^n$

$$f = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{bmatrix}, \qquad f_j : D \to \mathbb{R}, \quad j = 1, 2, \ldots, n$$

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \in D \subset \mathbb{R}^n, \qquad x_j \in \mathbb{R}, \quad j = 1, 2, \ldots, n$$

Goal: Find $x \in D$ such that

$$f(x) = 0.$$

Thus, we have n nonlinear equations for n unknowns.

Example 2.2. ***Newton's Method Example 1: Possible Scenarios***
Even the linear case shows that a nonlinear solver must be prepared for several outcomes. Take $D = \mathbb{R}^n$ and $f : \mathbb{R}^n \to \mathbb{R}^n$ with

$$f(x) = b - Ax, \qquad b \in \mathbb{R}^n,\ A \in \mathbb{R}^{n \times n},$$

so that

$$f(x) = 0 \ \Leftrightarrow \ Ax = b.$$

From linear algebra there are exactly three possibilities, depending on A and b: one solution (when A is nonsingular), no solution, or infinitely many solutions. For a general nonlinear f the situation is at least this varied, and usually worse: the number of solutions need not be 0, 1, or ∞, and no amount of inspecting f locally will reveal it. This is why nonlinear methods aim at *a* solution near a starting guess, not at *the* solution set.

Example 2.3. ***Newton's Method Example 2***
A one-parameter family already exhibits all three solution counts. Let $f =$

$f_\alpha : \mathbb{R} \to \mathbb{R}$ with $f_\alpha(x) = x^2 + \alpha$, where $\alpha \in \mathbb{R}$ is a parameter. The graph is an upward-facing parabola whose minimum value is α, so the number of real zeros is decided by the sign of α:

1. if $\alpha > 0$, the parabola stays above the axis and f_α has no zero;
2. if $\alpha = 0$, the parabola touches the axis at a single point, $x = 0$, which is a *double* zero: $f'(0) = 0$ there, precisely the situation in which the Newton step $-f/f'$ is in trouble;
3. if $\alpha < 0$, there are two simple zeros $x = \pm\sqrt{-\alpha}$.

The lesson: an arbitrarily small change in the data (moving α through 0) changes the answer structurally. A numerical method can only be expected to behave well away from such transitions.

Example 2.4. ***Newton's Method Example 3***
Solution sets can also be infinite and crowded. Take $n = 1$, $D = \mathbb{R}^+ = \{x \in \mathbb{R} \mid x > 0\}$, and $f(x) = \sin\frac{1}{x}$. As $x \to 0^+$ the argument $\frac{1}{x}$ grows without bound, so f oscillates faster and faster near the origin. The zeros are found by solving

$$f(x) = 0 \;\Leftrightarrow\; \frac{1}{x} = j\pi, \quad j = 1, 2, \ldots \qquad \Longrightarrow \qquad x = \frac{1}{j\pi}.$$

There are infinitely many zeros, and they accumulate at 0. Near the accumulation point, which zero Newton's method finds (if any) depends delicately on the starting guess: another reminder that convergence statements for nonlinear problems are necessarily *local*.

Remark 2.5. ***Newton's Method Algorithm for*** $n = 1$
The idea is entirely geometric. At the current guess x^k, replace f by its tangent line and slide down the tangent to the x-axis; the landing point is the next guess. The figure shows one step, and the algorithm below records the loop.

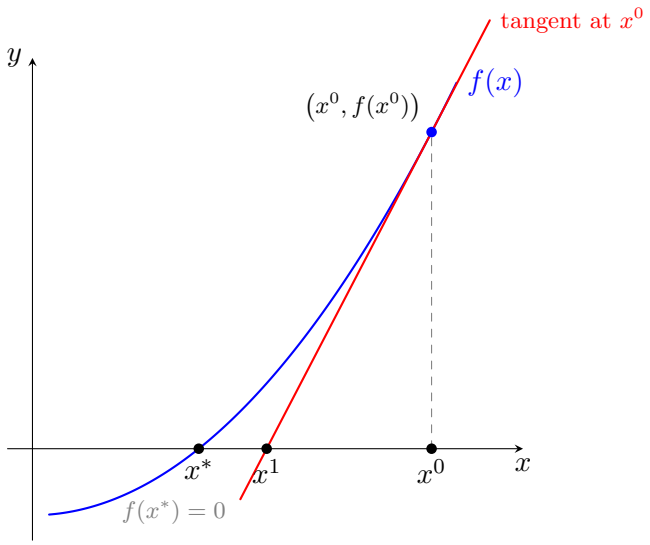


Figure 1: One step of Newton's method: the tangent line to f at x^0 intersects the x-axis at the new approximation x^1.

Algorithm 3 Newton's Method for $f(x^*) = 0$, $n = 1$

Require: initial guess x^0; tolerance $\varepsilon > 0$

1: **for** $k = 0, 1, 2, \ldots$ **do**

2: Linearize f around x^k:

$$t(x) = f(x^k) + f'(x^k)\,(x - x^k)$$

3: Set $t(x^{k+1}) = 0$ and solve (provided $f'(x^k) \neq 0$):

$$x^{k+1} = x^k - \frac{f(x^k)}{f'(x^k)}$$

4: **if** $|x^{k+1} - x^k| < \varepsilon$ **or** $|f(x^{k+1})| < \varepsilon$ **then**

5: **return** x^{k+1}

6: **end if**

7: **end for**

Remark 2.6. ***Questions about Newton's Method*** 1. Can we guarantee that $\lim_{k\to\infty} x^k = x^*$?

2. If yes, what is the speed of convergence?

Example 2.7. ***Failure of Newton's Method***

The following classic example shows that even a maximally well-behaved

function, smooth, increasing, with exactly one zero, can defeat Newton's method if the starting guess is poor. Let $f : \mathbb{R} \to \mathbb{R}$ with $f(x) = \tan^{-1} x$ and $f'(x) = \frac{1}{1+x^2}$.

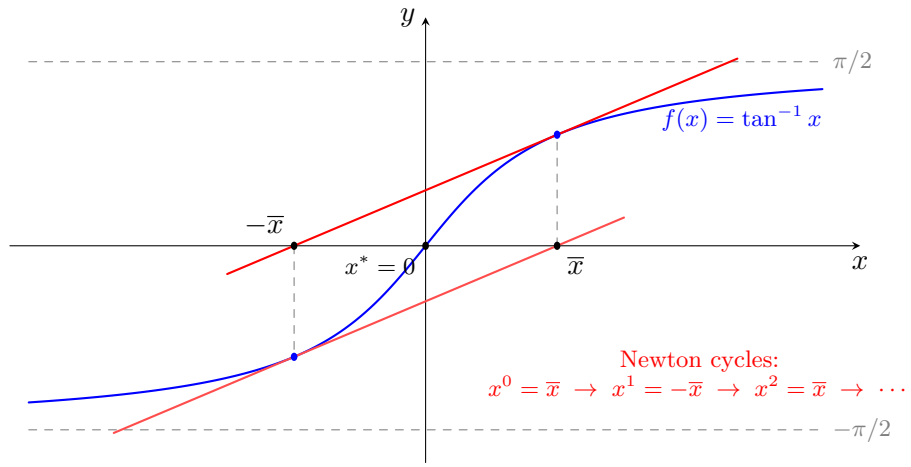


Figure 2: Failure of Newton's method for $f(x) = \tan^{-1} x$: starting from $\pm\overline{x}$ the iterates cycle forever; starting farther out, they diverge.

f has a single zero, $x^* = 0$. Applying Newton's method from a starting guess $x^0 \in \mathbb{R}$ gives the iteration

$$x^{k+1} = x^k - \left(1 + (x^k)^2\right) \tan^{-1}(x^k), \qquad k = 0, 1, 2, \ldots$$

Does it converge? The danger is visible in the figure: far from the origin f flattens out, $f'(x) \to 0$, so the Newton step $-f/f'$ becomes long and throws the iterate far past the zero, to the *other* flat branch. The boundary between recovery and disaster is the starting value whose overshoot lands exactly at its own mirror image. Let $\overline{x}$ be the unique solution of

$$-\overline{x} = \overline{x} - (1 + \overline{x}^2) \tan^{-1} \overline{x}, \qquad \overline{x} > 0$$
$$\overline{x} = 1.39174\ldots$$

where $\overline{x}$ is computed by using Newton's Method for

$$g(x) = 2x - (1 + x^2) \tan^{-1} x.$$

The behavior of the method now splits into three regimes, and the value $\overline{x}$ is the exact threshold:

- If $x^0 = \overline{x}$, then $x^k = (-1)^k \overline{x}$ for all k: the iterates ping-pong between $\overline{x}$ and $-\overline{x}$ forever, converging to nothing. (Starting at $-\overline{x}$ gives the same cycle with the signs reversed.)
- If $\left|x^0\right| < \overline{x}$, each overshoot lands strictly closer to the origin, and $\lim_{k\to\infty} x^k = 0 = x^*$.
- If $\left|x^0\right| > \overline{x}$, each overshoot lands strictly farther out, and $\left|x^k\right| \to \infty$: the method diverges.

Note what this example does *not* show: it does not show Newton's method is unreliable. Started anywhere in $(-\overline{x}, \overline{x})$, an interval of length nearly 2.8, the method converges rapidly. What it shows is that convergence is local, and that a practical implementation needs a safeguard for bad starting points. Both threads are taken up later in this chapter.

2.1.3 Filling in the Details: The Linearization Step

Everything comes from Taylor's theorem at the current guess x^0:

$$f(x) = \underbrace{f(x^0) + f'(x^0)(x - x^0)}_{=:t(x),\ \text{tangent line}} + \underbrace{O((x - x^0)^2)}_{\text{ignored}}.$$

We cannot solve $f(x) = 0$, but we *can* solve $t(x) = 0$:

$$0 = f(x^0) + f'(x^0)(x^1 - x^0) \quad \Longrightarrow \quad x^1 = x^0 - \frac{f(x^0)}{f'(x^0)}.$$

Geometrically: slide down the tangent line to the x-axis.

The neglected term is $O((x - x^0)^2)$: if the current error is small, the *new* error is roughly its square. This is the germ of quadratic convergence (doubling of correct digits), which we will prove later in this chapter.

A worked example to keep in mind: computing $\sqrt{a}$ means solving $f(x) = x^2 - a = 0$. Newton gives

$$x^{k+1} = x^k - \frac{(x^k)^2 - a}{2x^k} = \frac{1}{2}\left(x^k + \frac{a}{x^k}\right).$$

For $a = 2$, $x^0 = 1$:

$$x^1 = 1.5, \quad x^2 = 1.41\overline{6}, \quad x^3 = 1.4142157\ldots, \quad x^4 = 1.4142136\ldots$$

Correct digits: $1, 3, 6, 12, \ldots$: they double each step. This is what quadratic convergence looks like in practice.

2.1.4 Understanding the $\tan^{-1}$ Failure

The iteration is $x^{k+1} = g(x^k)$ with $g(x) = x - (1 + x^2)\tan^{-1} x$. Since g is odd, a 2-cycle $x \to -x \to x$ occurs exactly when

$$g(\overline{x}) = -\overline{x} \quad \Longleftrightarrow \quad 2\overline{x} = (1 + \overline{x}^2)\tan^{-1}\overline{x},$$

which has the unique positive solution $\overline{x} = 1.39174\ldots$

- $\left|x^0\right| < \overline{x}$: each step overshoots but lands *closer*; convergence to $x^* = 0$.
- $\left|x^0\right| = \overline{x}$: perpetual ping-pong $\overline{x} \to -\overline{x} \to \overline{x} \to \cdots$
- $\left|x^0\right| > \overline{x}$: overshoot grows; $\left|x^k\right| \to \infty$.

Takeaways: (1) Newton's convergence is *local*: guarantees require x^0 close enough to x^*; (2) flat functions ($f' \to 0$ far out) make the Newton step $-f/f'$ huge. Both issues motivate the convergence theorem and damping, both later in this chapter.

2.2 Newton's Method: Local Convergence

This lecture extends Newton's method from one equation to systems: the derivative becomes the Jacobian matrix, and each iteration requires the solution of one linear system with that matrix: an LU factorization per step. We establish the method's affine invariance, an exact symmetry under rescaling and recombination of the equations, which will dictate the form of every hypothesis and every stopping test we use later. We also begin assembling the tools (norms for vectors and matrices) needed to state a rigorous convergence theorem.

2.2.1 Newton's Method (Continued)

This section extends the method to n equations in n unknowns. The derivative becomes the Jacobian matrix, the division becomes a linear solve, and the tools of Chapter 1 apply directly. We also record the cost of one iteration and a first structural property, affine invariance, which will shape the convergence theory.

Remark 2.8. ***Newton's Method for*** $n \geq 1$

$$f : D \to \mathbb{R}^n, \qquad f = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{bmatrix}, \qquad x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \in D$$

Initial guess: $x^0 \in D$.

$$f(x) = f(x^0) + Df(x^0)(x - x^0) + O(\|x - x^0\|^2) \tag{3}$$

where

$$\|x\| = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2} = \text{Euclidean norm.}$$

$$Df(x) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix} = \left[\frac{\partial f_j}{\partial x_k}\right]_{j,k=1,2,\ldots,n} = \text{Jacobian of } f$$

Obtain a new approximation x^1 by setting (3) to 0:

$$0 = f(x^0) + Df(x^0)\underbrace{(x^1 - x^0)}_{\Delta x^0}$$

$$Df(x^0)\,\Delta x^0 = -f(x^0)$$
$$x^1 = x^0 + \Delta x^0.$$

This is a system of n linear equations for n unknowns.

Algorithm 4 Newton's Method for Systems

Require: $f : D \to \mathbb{R}^n$, initial guess $x^0 \in D$

for $k = 0, 1, 2, \ldots$ **do**
  if $Df(x^k)$ is singular **then**
    stop
  end if
  solve $Df(x^k)\,\Delta x^k = -f(x^k)$ for Δx^k ▷ one LU factorization + two triangular solves
  $x^{k+1} \leftarrow x^k + \Delta x^k$
  if $x^{k+1} \notin D$ **then**
    stop
  end if
  if $f(x^{k+1}) \approx 0$ **then**
    return x^{k+1}
  end if
end for

Remark 2.9.

1. Need to solve a linear system at each k step $\Rightarrow$ LU factorization of $A = Df(x^k)$
2. Newton's method is *affine-invariant*:
   Let $M \in \mathbb{R}^{n\times n}$ be a fixed nonsingular matrix, and let $g : D \to \mathbb{R}^n$, $g(x) := Mf(x)$ for all $x \in D$. Thus, $f(x) = 0 \Leftrightarrow Mf(x) = 0 \Leftrightarrow g(x) = 0$. Then Newton's method applied to f and g results in the same iterates x^k, $k = 0, 1, 2, \ldots$.

Proof. (That Newton's method is affine-invariant).

$$g(x) = Mf(x)$$
$$Dg(x^k) = MDf(x^k)$$
$$Dg(x^k)\,\Delta x^k = -g(x^k) \;\Leftrightarrow\; Df(x^k)\,\Delta x^k = -f(x^k)$$

□

2.2.2 Convergence of Newton's Method

With the method defined, we ask what can be guaranteed. This section assembles the ingredients of the convergence theorem: the questions it must

answer, the norms in which the answers are stated, and the Lipschitz condition that quantifies the nonlinearity of f.

Proposition 2.10. ***Convergence***
Let x^* denote a zero of f, i.e. $f(x^*) = 0$. If $Df(x^*)$ is nonsingular and x^0 is "close" to x^*, then

$$\lim_{k\to\infty} x^k = x^*$$

and the speed of convergence is quadratic, i.e.

$$\left\| x^{k+1} - x^k \right\| \leq C \left\| x^k - x^* \right\|^2$$

Definition 2.11. ***Quadratic Convergence***
The number of correct digits roughly doubles in each iteration.

Theorem 2.12. ***Convergence of Newton's Method, Affine-Invariant Version***
Assumptions:

- $D \subset \mathbb{R}$ is convex
- $f : D \to \mathbb{R}^n$ is C^1
- $x^0 \in D$ and $Df(x^0)$ is nonsingular
- Df satisfies the affine-invariant Lipschitz condition:

$$\left\| (Df(x^0))^{-1} \big(Df(y) - Df(x) \big) \right\| \leq \gamma \left\| y - x \right\|$$

Here $\|\cdot\| : \mathbb{R}^n \to \mathbb{R}$ is a vector norm and $\|M\| := \max_{\|x\|=1} \|Mx\|$ is the associated matrix norm, which satisfies:

1. $\|Mx\| \leq \|M\| \, \|x\|$
2. $\|I\| = 1$
3. $\|MN\| \leq \|M\| \, \|N\|$

2.2.3 Filling in the Details: One Newton Step, Concretely

Take the 2×2 system

$$f(x) = \begin{bmatrix} x_1^2 + x_2^2 - 1 \\ x_2 - x_1^2 \end{bmatrix} = 0 \qquad (\text{circle} \cap \text{parabola}).$$

Jacobian:

$$Df(x) = \begin{bmatrix} 2x_1 & 2x_2 \\ -2x_1 & 1 \end{bmatrix}.$$

From $x^0 = (1,1)^T$: $f(x^0) = (1,0)^T$ and

$$\begin{bmatrix} 2 & 2 \\ -2 & 1 \end{bmatrix} \Delta x^0 = -\begin{bmatrix} 1 \\ 0 \end{bmatrix} \Rightarrow \Delta x^0 = \begin{bmatrix} -1/6 \\ -1/3 \end{bmatrix}, \qquad x^1 = \begin{bmatrix} 5/6 \\ 2/3 \end{bmatrix}.$$

Note the workflow: evaluate f, evaluate Df, **LU-solve**, update. Never form $Df(x^k)^{-1}$ explicitly; solving the system is cheaper and more stable.

Cost per Newton step for a dense Jacobian:

$$\underbrace{n^2 \text{ entries of } Df}_{\text{evaluation}} + \underbrace{\tfrac{2n^3}{3}}_{\text{LU}} + \underbrace{2n^2}_{\text{triangular solves}}.$$

The LU factorization dominates; this is why the linear algebra of Chapter 1 is the engine inside every nonlinear solver.

2.2.4 Why Affine Invariance Is the Right Point of View

Rescaling or mixing the equations, $g := Mf$ with M nonsingular, does not change the solution set, and the proof showed it does not change the Newton iterates either:

$$Dg(x^k)^{-1} g(x^k) = \big(MDf(x^k)\big)^{-1} Mf(x^k) = Df(x^k)^{-1} f(x^k).$$

Consequences:

- Any *theory* for Newton's method should also be invariant under $f \mapsto Mf$. That is why the Lipschitz condition in the theorem is written with the factor $Df(x^0)^{-1}$ in front; the raw condition $\|Df(y) - Df(x)\| \leq \gamma \|y - x\|$ would change under M, but the affine-invariant one does not.

- Any *convergence test* should be invariant too. Checking $\big\|f(x^k)\big\|$ is not (multiply f by 10^6!); checking the Newton correction $\big\|Df(x^k)^{-1} f(x^k)\big\| = \big\|\Delta x^k\big\|$ is. This foreshadows the monotonicity test introduced later in this chapter.

2.3 The Newton Convergence Theorem

This is the central theoretical lecture on Newton's method. We state a convergence theorem of Newton–Kantorovich type in an affine-invariant form: hypotheses on the length α of the first Newton step, on an affine-invariant Lipschitz constant γ of the Jacobian, and on their product $h = \alpha\gamma < \frac{1}{2}$ guarantee that the iteration is well defined, remains in an explicitly given ball, and converges. The proof introduces two standard tools of numerical analysis, the Banach perturbation lemma and the integral form of the mean value theorem, and controls the vector iteration by a pair of scalar majorant sequences.

2.3.1 Newton's Method Convergence Theorem

This is the central theorem of the chapter. Its affine-invariant form follows Deuflhard's framework; the classical Newton–Kantorovich theorem that it parallels can be found in Stoer and Bulirsch. The statement is long because it delivers four separate conclusions: the iteration is well defined, it stays in an explicit ball, it converges, and it converges quadratically. The proof occupies this section and the next.

Theorem 2.13.
Given:

- $f : D \to \mathbb{R}^n$
- $x^0 \in D$ and $Df(x^0)$ is nonsingular
- $\left\| Df(x^0)^{-1}(Df(y) - Df(x)) \right\| \le \gamma \left\| y - x \right\|$ for all $x, y \in D$ (bigger $\gamma \Rightarrow$ more nonlinear)
- $\left\| Df(x^0)^{-1} f(x^0) \right\| \le \alpha$ for some $\alpha > 0$
- $h := \alpha\gamma < \frac{1}{2}$

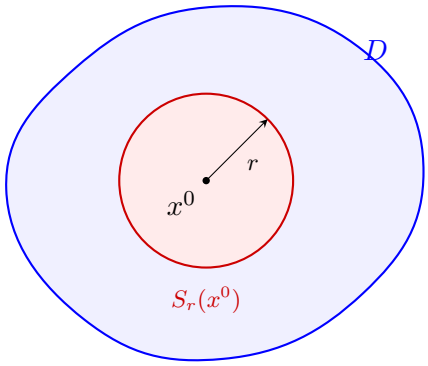


Figure 3: $S_r(x^0) := \{x \in \mathbb{R}^n \mid \|x - x^0\| < r\}$, $\overline{S_r(x^0)} := \{x \in \mathbb{R}^n \mid \|x - x^0\| \le r\}$

- $\overline{S_r(x^0)} \subset D, \quad r = \dfrac{1 - \sqrt{1 - 2h}}{\gamma} > 0$

Claims:

1. All Newton iterates x^k satisfy $x^k \in S_r(x^0)$, $Df(x^k)$ is nonsingular
2. The iterates x^k converge quadratically to a zero of f, $x^* := \lim_{k\to\infty} x^k$, $x^* \in \overline{S_r(x^0)}$
3. x^* is the *only* zero of f in $D \cap S_R(x^0)$, $R := \dfrac{1 + \sqrt{1 - 2h}}{\gamma}$

Note:

$$\|x^1 - x^0\| = \|-Df(x^0)^{-1} f(x^0)\| \le \alpha = \frac{h}{\gamma}$$

Proof. (of parts 1 and 2 of the theorem). We define two scalar sequences (h_k) and (γ_k) as

$$h_k := \begin{cases} h & k = 0 \\ \dfrac{1}{2}\dfrac{h_{k-1}^2}{(1 - h_{k-1})^2} & k > 0 \end{cases} \qquad \gamma_k := \begin{cases} \gamma & k = 0 \\ \dfrac{\gamma_{k-1}}{1 - h_{k-1}} & k > 0 \end{cases}$$

It is easy to verify that for all $k = 0, 1, \ldots$ we have

$$\lim_{j\to\infty} h_j = 0 < h_{k+1} < h_k < \frac{1}{2}$$
$$\gamma_k < \gamma_{k+1} < \frac{\gamma}{\sqrt{1 - 2h}} = \lim_{j\to\infty} \gamma_j$$

2 Properties:

$$\frac{h_k}{\gamma_k} \le \frac{1}{2^k}\,\alpha$$

$$\frac{h_0}{\gamma_0} + \frac{h_1}{\gamma_1} + \cdots + \frac{h_{k-1}}{\gamma_{k-1}} = \frac{1}{\gamma} - \frac{1}{\gamma_k} < r = \frac{1-\sqrt{1-2h}}{\gamma}$$

Claim: For $k = 0, 1, \ldots$, $x^k \in S_r(x^0)$, $Df(x^k)$ is nonsingular, and

$$\left\| Df(x^k)^{-1}(Df(y) - Df(x)) \right\| \le \gamma_k \left\| y - x \right\|, \qquad x, y \in D.$$

We will also prove that

$$\left\| x^{k+1} - x^k \right\| \le \frac{h_k}{\gamma_k} \le \frac{1}{2^k}\,\alpha.$$

Proof by induction on k:
$k = 0 \Rightarrow$ satisfied (by our assumptions).
$k \ge 1$:

$$\left\| x^k - x^0 \right\| \le \underbrace{\left\| x^k - x^{k-1} \right\|}_{\le \frac{h_{k-1}}{\gamma_{k-1}}} + \underbrace{\left\| x^{k-1} - x^{k-2} \right\|}_{\le \frac{h_{k-2}}{\gamma_{k-2}}} + \cdots + \underbrace{\left\| x^1 - x^0 \right\|}_{\le \frac{h_0}{\gamma_0}}$$

$$\le \frac{h_0}{\gamma_0} + \frac{h_1}{\gamma_1} + \cdots + \frac{h_{k-1}}{\gamma_{k-1}} < r$$

Thus, $x^k \in S_r(x^0)$.

$$Df(x^k) = \underbrace{Df(x^{k-1})}_{\text{nonsingular}} \Big(I - \underbrace{Df(x^{k-1})^{-1}\big(Df(x^{k-1}) - Df(x^k)\big)}_{:=A} \Big) \qquad (4)$$

$$\|A\| = \left\| Df(x^{k-1})^{-1}(Df(x^k) - Df(x^{k-1})) \right\| \le \gamma_{k-1} \underbrace{\left\| x^k - x^{k-1} \right\|}_{\le \frac{h_{k-1}}{\gamma_{k-1}}} \le h_{k-1} < \frac{1}{2} < 1$$

Banach Lemma: If $\|A\| < 1$, then $I - A$ is nonsingular and

$$\left\| (I - A)^{-1} \right\| \le \frac{1}{1 - h_{k-1}}.$$

Thus, by (4), $Df(x^k)$ is nonsingular and

$$\begin{aligned}
&\left\| Df(x^k)^{-1} \underbrace{\big(Df(x^{k-1})Df(x^{k-1})^{-1}\big)}_{I} \big(Df(y) - Df(x)\big) \right\| \\
&\quad \le \left\| \underbrace{Df(x^k)^{-1}Df(x^{k-1})}_{(I-A)^{-1}} \right\| \left\| Df(x^{k-1})^{-1}(Df(y) - Df(x)) \right\| \\
&\le \frac{1}{1-h_{k-1}} \gamma_{k-1} \|y - x\| \\
&= \gamma_k \|y - x\|, \qquad x, y \in D.
\end{aligned}$$

$$\begin{aligned}
x^{k+1} - x^k &= -Df(x^k)^{-1}\Big(f(x^k) \underbrace{-f(x^{k-1}) - Df(x^{k-1})(x^k - x^{k-1})}_{=0}\Big) \\
&= -Df(x^k)^{-1} \int_0^1 \Big(Df\big(x^{k-1} + t(x^k - x^{k-1})\big) - Df(x^{k-1})\Big)(x^k - x^{k-1})\, dt
\end{aligned}$$

$$\begin{aligned}
\left\|x^{k+1} - x^k\right\| &\le \int_0^1 \underbrace{\left\|Df(x^k)^{-1}\big(Df(x^{k-1} + t(x^k - x^{k-1})) - Df(x^{k-1})\big)\right\|}_{\cdot \left\|x^k - x^{k-1}\right\|} dt \\
&\le \gamma_k \int_0^1 \left\|\cancel{x^{k-1}} + t(x^k - x^{k-1}) - \cancel{x^{k-1}}\right\| \left\|x^k - x^{k-1}\right\| dt \\
&= \gamma_k \left\|x^k - x^{k-1}\right\|^2 \int_0^1 t\, dt \\
&= \frac{\gamma_k}{2} \underbrace{\left\|x^k - x^{k-1}\right\|^2}_{\le \left(\frac{h_{k-1}}{\gamma_{k-1}}\right)^2} \\
&\le \frac{\gamma_k}{2} \frac{h_{k-1}^2}{\gamma_{k-1}^2} = \frac{h_k}{\gamma_k}
\end{aligned}$$

To be continued... □

2.3.2 Filling in the Details: Reading the Assumptions

Each hypothesis is a measurable quantity at the *starting point*:

- $\alpha \geq \left\|Df(x^0)^{-1}f(x^0)\right\| = \left\|x^1 - x^0\right\|$: the length of the *first Newton step*. Small α = we start near a solution (as measured by Newton itself).
- γ: the affine-invariant Lipschitz constant of Df, a curvature/nonlinearity measure. $\gamma = 0$ means f is affine and Newton finishes in one step.
- $h = \alpha\gamma < \frac{1}{2}$: "first step $\times$ nonlinearity" is small: the tangent model is trustworthy over the region the iterates will explore.

The radius $r = \frac{1-\sqrt{1-2h}}{\gamma}$ then satisfies $\alpha \leq r \leq 2\alpha$ (check: $1 - \sqrt{1-2h} \in [h, 2h]$), so the whole iteration happens within about two first-steps of x^0.

The Banach lemma (used to keep Jacobians invertible): if $\|A\| < 1$ then $I - A$ is nonsingular and

$$(I-A)^{-1} = I + A + A^2 + A^3 + \cdots, \qquad \left\|(I-A)^{-1}\right\| \leq \sum_{j\geq 0} \|A\|^j = \frac{1}{1-\|A\|}.$$

(The Neumann series converges absolutely because $\left\|A^j\right\| \leq \|A\|^j$ by submultiplicativity; multiplying the series by $I - A$ telescopes to I.)

The integral identity (used to compare f at two points): for $\varphi(t) := f\big(x^{k-1} + t(x^k - x^{k-1})\big)$, the chain rule gives $\varphi'(t) = Df\big(x^{k-1} + t\Delta\big)\Delta$ with $\Delta = x^k - x^{k-1}$, and the fundamental theorem of calculus yields

$$f(x^k) - f(x^{k-1}) = \int_0^1 Df\big(x^{k-1} + t\Delta\big)\Delta \, dt.$$

Subtracting the tangent-model term $Df(x^{k-1})\Delta$ (which is exactly $-f(x^{k-1})$ by the Newton equation!) produces the integrand $\big(Df(x^{k-1} + t\Delta) - Df(x^{k-1})\big)\Delta$, and *this* is what the Lipschitz condition can bound, giving the factor $\gamma_k\, t\, \|\Delta\|$ and, after $\int_0^1 t\, dt = \frac{1}{2}$, the quadratic estimate $\left\|x^{k+1} - x^k\right\| \leq \frac{\gamma_k}{2}\|\Delta\|^2$. This one computation is the heart of the whole proof.

2.3.3 The Scalar Majorant Sequences at a Glance

$$h_k = \frac{1}{2}\frac{h_{k-1}^2}{(1-h_{k-1})^2}, \qquad \gamma_k = \frac{\gamma_{k-1}}{1-h_{k-1}}, \qquad h_0 = h,\ \gamma_0 = \gamma.$$

- $h < \frac{1}{2} \Rightarrow \frac{h}{1-h} < 1 \Rightarrow h_1 = \frac{1}{2}\big(\frac{h_0}{1-h_0}\big)^2 < \frac{h_0}{2}$: the h_k decay (in fact doubly exponentially, squaring each time).

- γ_k increases but stays bounded: $\gamma_k \nearrow \gamma/\sqrt{1-2h}$.
- The step bound $\left\|x^{k+1}-x^k\right\| \le h_k/\gamma_k \le \alpha/2^k$ then makes the iterates a Cauchy sequence (next lecture), and the sum $\sum_{j<k} h_j/\gamma_j < r$ keeps every iterate inside the ball.

Big picture: the vector iteration is dominated ("majorized") by a scalar recursion that we can analyze completely. This proof pattern, majorant sequences, recurs throughout numerical analysis.

2.4 Newton's Method in Practice; Damping

We complete the convergence proof begun in the previous lecture: the iterates form a Cauchy sequence, the limit is a zero of f, and the convergence is quadratic, which in practice means the number of correct digits doubles at every step. We then turn to Newton's method as it is actually used. Because the theorem's hypotheses are rarely checkable, practical implementations monitor progress instead; affine invariance leads to the natural monotonicity test, based on the simplified Newton correction, which can be evaluated cheaply by reusing the current LU factorization. When the test fails we must modify the step. This is the subject of damping, introduced here and developed next lecture.

2.4.1 Recap

We restate the theorem and the quantities appearing in it before completing the proof.

$$
\begin{gathered}
x^k \in S_r(x^0) \\
\left\| Df(x^k)^{-1}(Df(y)-Df(x)) \right\| \le \gamma_k \left\| y-x \right\|, \qquad x,y \in D \\
\left\| x^{k+1}-x^k \right\| \le \frac{h_k}{\gamma_k} \le \frac{\alpha}{2^k} \\
f : D \to \mathbb{R}^n, \qquad f(x^*)=0.
\end{gathered}
$$

Claim: (x^k) is a Cauchy sequence.
Let $m > k$.

$$\begin{aligned}
\left\|x^m - x^k\right\| &\le \left\|x^m - x^{m-1}\right\| + \left\|x^{m-1} - x^{m-2}\right\| + \cdots + \left\|x^{k+1} - x^k\right\| \\
&\le \frac{\alpha}{2^{m-1}} + \frac{\alpha}{2^{m-2}} + \cdots + \frac{\alpha}{2^k} \\
&\le \frac{\alpha}{2^k}\left(1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^{m-k-1}}\right) \\
&\le \frac{\alpha}{2^k}\underbrace{\left(1 + \frac{1}{2} + \frac{1}{4} + \cdots\right)}_{=2} \\
&\le \frac{\alpha}{2^{k-1}}
\end{aligned}$$

Thus, (x^k) is Cauchy and therefore convergent:

$$\lim_{k\to\infty} x^k = x^*, \qquad x^* \in \overline{S_r(x^0)}.$$

Claim: x^* is a zero of f.

$$f(x^k) = -Df(x^k)(x^{k+1} - x^k) \tag{5}$$

$$\left\|f(x^k)\right\| \le \underbrace{\left\|Df(x^k)\right\|}_{\le \max_{x\in\overline{S_r(x^0)}} \|Df(x)\| := M} \underbrace{\left\|x^{k+1} - x^k\right\|}_{\le \frac{\alpha}{2^k}} \le \frac{\alpha M}{2^k}$$

Thus,

$$0 = \lim_{k\to\infty} f(x^k) = f\big(\lim_{k\to\infty} x^k\big) = f(x^*)$$

Claim: the convergence is quadratic.

$$\begin{aligned}
x^{k+1} &= x^k - Df(x^k)^{-1} f(x^k) \qquad \text{by (5)} \\
x^{k+1} - x^* &= x^k - x^* - Df(x^k)^{-1}\big(f(x^k) - \underbrace{f(x^*)}_{=0}\big) \\
&= \int_0^1 Df(x^k)^{-1}\Big(Df(x^k) - Df\big(x^* + t(x^k - x^*)\big)\Big)(x^k - x^*)\, dt
\end{aligned}$$

$$\begin{aligned}
\left\|x^{k+1}-x^*\right\| &\le \left\|x^k-x^*\right\| \cdot \int_0^1 \left\|Df(x^k)^{-1}\big(Df(x^k)-Df(x^*+t(x^k-x^*))\big)\right\| dt \\
&\le \left\|x^k-x^*\right\| \gamma_k \int_0^1 \underbrace{\left\|x^k-x^*-t(x^k-x^*)\right\|}_{=(1-t)\left\|x^k-x^*\right\|} dt \\
&= \left\|x^k-x^*\right\|^2 \gamma_k \int_0^1 (1-t)\, dt \\
&= \frac{\gamma_k}{2}\left\|x^k-x^*\right\|^2 \\
&\le \frac{1}{2}\frac{\gamma}{\sqrt{1-2h}}\left\|x^k-x^*\right\|^2
\end{aligned}$$

Note:

$$\frac{d}{dt} f\big(x^*+t(x^k-x^*)\big) = Df\big(x^*+t(x^k-x^*)\big)(x^k-x^*)$$

Parameterizing in terms of t is the reason why we need complexity.

2.4.2 Use of Newton in Practice

A convergence theorem with unverifiable constants does not by itself yield a usable algorithm. In practice one monitors the iteration and reacts. This section derives the standard test, chosen for affine invariance, and shows how to evaluate it at negligible cost by reusing the current factorization.

Remark 2.14. ***Monotonicity Test (Preliminary)***
Newton iterates: $x^{k+1} = x^k \underbrace{-Df(x^k)^{-1}f(x^k)}_{=\Delta x^k}$.
If all goes well: $x^k \to x^*, \quad f(x^*) = 0$.
Check for progress: $\left\|f(x^{k+1})\right\| \le \theta \left\|f(x^k)\right\|$ for some $\theta < 1$. Note that this check is *not* affine-invariant!
Instead:

$$\left\|Df(x^k)^{-1}f(x^{k+1})\right\| \le \theta \underbrace{\left\|Df(x^k)^{-1}f(x^k)\right\|}_{=\|\Delta x^k\|}$$

In addition to

$$Df(x^k)\,\Delta x^k = -f(x^k) \tag{6}$$

we also need to solve a second system:

$$Df(x^k)\,\Delta x^{k+1} = -f(x^{k+1})$$

Additional cost: $O(n^2)$ flop if LU factorization is used to solve (6).

Remark 2.15. ***Monotonicity Test (Refined)*** • Compute Δx^{k+1} in addition to Δx^k.

- Check if $\left\|\Delta x^{k+1}\right\| \leq \theta \left\|\Delta x^k\right\|$ for some $\theta < 1$. (Typical value is $\theta = \frac{1}{2}$)
- If not satisfied, use Newton's method with damping

2.4.3 Newton's Method with Damping

When the monotonicity test fails, the full Newton step has overshot. Damping shortens the step without changing its direction; the justification is the descent property proved below. The globalized method that results is developed fully in Deuflhard's monograph on Newton methods.

Remark 2.16. ***Newton's Method: Standard vs. Damped***
Standard: $x^{k+1} = x^k + \Delta x^k$
Damped: $x^{k+1} = x^{k+1}(\lambda_k) = x^k + \lambda_k \Delta x^k$, where $0 < \lambda_k \leq 1$ for some damping factor.

What is a suitable strategy for selecting λ_k?

2.4.4 Filling in the Details: Three Small Steps Made Explicit

(1) Geometric series in the Cauchy estimate. For $m > k$,

$$\left\|x^m - x^k\right\| \leq \sum_{j=k}^{m-1} \left\|x^{j+1} - x^j\right\| \leq \sum_{j=k}^{m-1} \frac{\alpha}{2^j} \leq \frac{\alpha}{2^k} \sum_{i=0}^{\infty} 2^{-i} = \frac{\alpha}{2^{k-1}} \xrightarrow{k\to\infty} 0,$$

independently of m, which is exactly the Cauchy property. Completeness of $\mathbb{R}^n$ then supplies the limit x^*.

(2) Why $f(x^*) = 0$. The Newton equation rearranges to the identity $f(x^k) = -Df(x^k)\,(x^{k+1} - x^k)$, so

$$\left\|f(x^k)\right\| \leq M\,\frac{\alpha}{2^k} \to 0, \qquad M := \max_{\overline{S_r}} \|Df\|$$

(M is finite: a continuous function on a compact ball attains its maximum). Continuity of f finishes it: $f(x^*) = \lim f(x^k) = 0$.

(3) The quadratic-rate constant. The final estimate was

$$\left\|x^{k+1} - x^*\right\| \leq \frac{\gamma_k}{2} \left\|x^k - x^*\right\|^2 \leq \underbrace{\frac{\gamma}{2\sqrt{1-2h}}}_{=:C} \left\|x^k - x^*\right\|^2.$$

Interpretation: once $C\left\|x^k - x^*\right\| < 10^{-1}$, each iteration squares the error, i.e. *doubles the number of correct digits*. In double precision you typically observe residuals like $10^{-1} \to 10^{-2} \to 10^{-4} \to 10^{-8} \to 10^{-16}$: four or five iterations from a decent guess.

2.4.5 Why the Monotonicity Test Is Designed This Way

Naive test: $\left\|f(x^{k+1})\right\| \le \theta \left\|f(x^k)\right\|$. Flaw: replace f by Mf and the test outcome can flip, but the iterates do not change. (Not affine-invariant.)

Fix: measure residuals in the *Newton metric* at x^k:

$$\left\|Df(x^k)^{-1} f(x^{k+1})\right\| \;\le\; \theta \underbrace{\left\|Df(x^k)^{-1} f(x^k)\right\|}_{=\|\Delta x^k\|}.$$

Both sides are unchanged under $f \mapsto Mf$. The right side is already known; the left side needs one extra solve

$$Df(x^k)\,\overline{\Delta x}^{k+1} = -f(x^{k+1})$$

with the *same* matrix, so we reuse the LU factors: cost only $O(n^2)$. The quantity $\overline{\Delta x}^{k+1}$ is called the *simplified Newton correction*; the test reads $\left\|\overline{\Delta x}^{k+1}\right\| \le \theta \left\|\Delta x^k\right\|$, typically with $\theta = \frac{1}{2}$. If it fails, we do not trust the full step and damping enters.

2.5 Damped Newton; Introduction to Conditioning

The first half of this lecture completes the damped Newton method. We prove that the Newton direction is a descent direction for an affine-invariant level function, which explains why shortening the step, rather than abandoning it, is the right response to a failed monotonicity test, and we describe the practical step-size strategy. The second half opens the third unit of the course: error analysis. We define the condition number of a computational problem, the factor by which relative input perturbations are amplified in the output, entirely independent of any algorithm.

2.5.1 Newton's Method with Damping (Continued)

This section completes the damped method: the step-size selection rule, the doubling strategy that restores full steps near the solution, and the assembled algorithm.

Remark 2.17. ***Goal of Damping***

$$\left\| Df(x^k)^{-1} f(x^{k+1}) \right\| \leq \theta \left\| Df(x^k)^{-1} f(x^k) \right\| \quad \text{for some } \theta < 1 \tag{7}$$

This can always be satisfied if λ_k is chosen small enough.

In the following,

$$\|v\| := \sqrt{v^T v} = \text{Euclidean norm.}$$

Proposition 2.18.
If $f(x^k) \neq 0$, then the Newton increment

$$\Delta x^k = -Df(x^k)^{-1} f(x^k)$$

is a descent direction for the function

$$\phi(x) := \frac{1}{2} \left\| Df(x^k)^{-1} f(x) \right\|^2$$

at x^k, i.e.

$$\frac{d}{d\lambda} \phi(x^k + \lambda \Delta x^k) \Big|_{\lambda=0} < 0.$$

Corollary 2.19.
If $\lambda_k > 0$ is chosen small enough, then

$$x^{k+1} = x^k + \lambda_k \Delta x^k$$

satisfies (7). This implies that

$$\phi(x^{k+1}) \leq \theta^2 \phi(x^k).$$

Proof. (of the descent proposition). Note: $f(x^k) \neq 0 \Rightarrow \Delta x^k \neq 0$.

$$\frac{d}{d\lambda} \phi(x^k + \lambda \Delta x^k) \Big|_{\lambda=0} = D_x \phi(x) \Big|_{x=x^k} \Delta x^k$$

$$D_x \phi(x) = \left(Df(x^k)^{-1} f(x) \right)^T Df(x^k)^{-1} Df(x)$$

$$= -(\Delta x^k)^T \Delta x^k = -\left\| \Delta x^k \right\|^2 < 0$$

□

Remark 2.20. ***Practical Strategy for Selecting*** λ
Determine $0 < \lambda_k \leq 1$ such that the monotonicity test is satisfied with $\theta = 1 - \frac{\lambda_k}{2}$. If possible, use $\lambda_k = 1$ (which implies that $\theta = \frac{1}{2}$), so that near the solution the method takes full Newton steps and retains quadratic convergence. The halving-and-doubling procedure below implements this.

Algorithm 5 Step-Size Selection for the Damped Newton Method

Require: current iterate x^k, Newton correction Δx^k, first trial value λ (from the previous step), minimum $\lambda_{\min} > 0$

for $\lambda_k = \lambda, \frac{\lambda}{2}, \frac{\lambda}{4}, \ldots, \lambda_{\min}$ **do**
  solve $Df(x^k)\,\Delta x^{k+1}(\lambda_k) = -f(x^k + \lambda_k \Delta x^k)$ ▷ reuse the LU factors of $Df(x^k)$: cost $O(n^2)$
  if $\left\|\Delta x^{k+1}(\lambda_k)\right\| \le \left(1 - \frac{\lambda_k}{2}\right) \left\|\Delta x^k\right\|$ **then**
    $x^{k+1} \leftarrow x^k + \lambda_k \Delta x^k$
    first trial value for the next step: $\min\{1, 2\lambda_k\}$
    return x^{k+1}
  end if
end for
stop: no admissible step size found ▷ Newton direction not usable; x^k may be near a critical point of f

2.5.2 Conditioning

We now begin the error-analysis portion of the course, which occupies this chapter and the next. The first concept is conditioning, a property of the problem alone, prior to any algorithm. The treatment follows Trefethen and Bau; Higham's book is the comprehensive reference for the whole chapter.
Problem: Input $\to$ Output (or No Solution)

Abstract Formulation: $X \to Y$, $x \in X$ is the input, $y = F(x) \in Y$ is the corresponding output.

Example 2.21. ***General Problem We Are Considering***
To see how a familiar task fits the abstract input–output frame, take the solution of the linear system $Ay = b$, where $A \in \mathbb{C}^{n\times n}$ is nonsingular and $b \in \mathbb{C}^n$. The input is the pair (A, b), the output is the solution vector, and the "problem" is the map between them:

$$X = \left\{ x = (A, b) \;\middle|\; A \in \mathbb{C}^{n\times n} \text{ is nonsingular},\ b \in \mathbb{C}^n \right\}$$
$$Y = \mathbb{C}^n$$
$$F(x) = F(A, b) := A^{-1} b = y \in \mathbb{C}^n.$$

Note carefully what F is *not*: it is not an algorithm. It is the exact mathematical map from data to solution, and its sensitivity to perturbations of (A, b) is a property of linear algebra alone, the same whether we later solve the system by LU, by QR, or by hand. Separating this sensitivity (condi-

tioning) from the additional errors an algorithm commits (stability) is the program of the next two chapters.

Definition 2.22. ***Conditioning of a Problem***
The impact of perturbation: $x \to x + \delta x$, δx is small. In other words, we are solving a slightly wrong problem and we want to know the effect on the output.

Perturbation of input: $\delta x\,(= x + \delta x - x)$
Perturbation of output: $\delta F(x)\,(= F(x + \delta x) - F(x))$

The *condition number* of a problem, $\kappa = \kappa(x)$, is a measure of the sensitivity of the output to small changes in the input:

$$\frac{\|\delta F(x)\|}{\|x\|} \leq \kappa(x) \frac{\|\delta x\|}{\|x\|} \qquad \text{for small } \|\delta x\|$$

2.5.3 Filling in the Details: The Descent Computation

Level function (note: affine-invariant, Jacobian frozen at x^k):

$$\phi(x) = \tfrac{1}{2} \left\| Df(x^k)^{-1} f(x) \right\|_2^2 = \tfrac{1}{2}\, g(x)^T g(x), \qquad g(x) := Df(x^k)^{-1} f(x).$$

Chain rule: $D_x\phi(x) = g(x)^T Dg(x) = g(x)^T Df(x^k)^{-1} Df(x)$. Evaluate at $x = x^k$, where $Df(x) = Df(x^k)$ cancels its inverse:

$$D_x\phi(x^k) = g(x^k)^T = \left(Df(x^k)^{-1} f(x^k) \right)^T = -(\Delta x^k)^T.$$

Directional derivative along the Newton step:

$$\frac{d}{d\lambda} \phi(x^k + \lambda \Delta x^k) \bigg|_{\lambda=0} = D_x\phi(x^k)\, \Delta x^k = -\left\| \Delta x^k \right\|_2^2 < 0. \qquad \blacksquare$$

Why $\theta = 1 - \lambda/2$? For small λ, first-order expansion of the residual gives

$$g(x^k + \lambda \Delta x^k) \approx (1 - \lambda)\, g(x^k),$$

i.e. the ideal decrease factor is $1 - \lambda$ (full step $\lambda = 1$: residual would vanish to first order). Demanding the *achievable* factor $1 - \frac{\lambda}{2}$, half the predicted decrease, accepts steps that realize at least half of the linear model's promise. This is the nonlinear- equations analogue of the Armijo condition from optimization.

The doubling rule $\lambda_{k+1} = \min\{1, 2\lambda_k\}$ lets the method accelerate back to full Newton steps (and quadratic convergence) once it enters the basin of attraction: damping costs nothing asymptotically.

2.5.4 Conditioning: Setting Up the Right Question

Before analyzing any more algorithms we must separate two issues:

$$\text{error} \;=\; (\text{sensitivity of the problem}) \times (\text{quality of the algorithm}).$$

Today we defined the first factor. Read the definition

$$\frac{\|\delta F(x)\|}{\|F(x)\|} \le \kappa(x)\,\frac{\|\delta x\|}{\|x\|}$$

as: *κ converts relative input error into worst-case relative output error.* Rules of thumb:

- Data known to ≈ 16 digits (ϵ_{mach}) and $\kappa = 10^8 \Rightarrow$ at most ≈ 8 trustworthy digits in the output; *no algorithm can do better.*
- κ depends on the problem *and* the input x; the same formula can be well-conditioned at one x and hopeless at another (next lecture: $x_1 - x_2$).

2.6 Notes and References

The affine-invariant convergence theory of this chapter follows the framework developed by Deuflhard; the textbook of Deuflhard and Hohmann contains the version presented here, and Deuflhard's monograph *Newton Methods for Nonlinear Problems* develops the theory in full, including the damping strategies of the last two lectures. The classical Newton–Kantorovich theorem, which the present theorem parallels, is treated in Stoer and Bulirsch.

2.7 Exercises

1. Derive the Babylonian iteration $x^{k+1} = \tfrac{1}{2}(x^k + a/x^k)$ as Newton's method for $f(x) = x^2 - a$. Show that the error $e_k = x^k - \sqrt{a}$ satisfies $e_{k+1} = e_k^2/(2x^k)$, and conclude quadratic convergence for $x^0 > 0$.

2. Carry out one step of Newton's method for the system $x_1^2 + x_2^2 = 4$, $x_1 x_2 = 1$, starting from $x^0 = (2, 1)^T$. Set up the Jacobian, solve the 2×2 linear system by hand, and state x^1.

3. Show that if $f(x) = Ax - b$ with A nonsingular, Newton's method reaches the exact solution in one step from any starting point.

4. Fill in the proof of affine invariance: for nonsingular M and $g = Mf$, show that the Newton iterates for g coincide with those for f, and that the affine-invariant Lipschitz condition $\left\| Df(x^0)^{-1}(Df(y) - Df(x)) \right\| \le \gamma \|y - x\|$ is unchanged when f is replaced by Mf.

5. For $f = \tan^{-1}$, the boundary between convergence and divergence of Newton's method is the point $\overline{x}$ satisfying $2\overline{x} = (1+\overline{x}^2)\tan^{-1}\overline{x}$. Compute $\overline{x}$ to six digits with the bisection method, and verify numerically that starting values slightly inside and outside produce the claimed behaviors.

6. Prove the Banach lemma refinement $\left\|(I - A)^{-1} - I\right\| \le \dfrac{\|A\|}{1 - \|A\|}$ for $\|A\| < 1$, using the Neumann series.

7. Show that the majorant quantities h_k of the convergence theorem satisfy $h_k \le h_{k-1}/2$ whenever $h_{k-1} \le 1/2$, and that in fact $h_k = O(h_{k-1}^2)$. What does each statement say about the speed of the iteration?

8. (Computing.) Implement the damped Newton method with the monotonicity test ($\theta = \frac{1}{2}$, halving strategy, $\lambda_{\min} = 2^{-10}$). Test it on $f(x) = \tan^{-1}(x)$ with $x^0 = 10$, where the undamped method diverges, and report the sequence of accepted step sizes λ_k.

Chapter 3

Conditioning, Floating-Point Arithmetic, and Stability

This unit builds the error-analysis framework of scientific computing. We quantify the sensitivity of problems (conditioning), describe how computers represent numbers and round arithmetic (the IEEE standard), and define what it means for an algorithm to be stable and backward stable. The unit culminates in the accuracy theorem: for backward stable algorithms, relative error $= O(\kappa\,\epsilon_{\text{mach}})$.

3.1 Conditioning; Floating-Point Numbers

We continue the study of conditioning with a formula for differentiable problems, $\kappa(x) = \|DF(x)\|\,\|x\| \,/\, \|F(x)\|$, and two model examples that will recur all semester: the square root, which is well conditioned everywhere, and the subtraction of nearly equal numbers, whose condition number blows up; this is the true source of the phenomenon called "loss of significant digits." We then prepare to discuss how real numbers are represented in a computer, reviewing positional (binary) representations and motivating the floating-point format.

3.1.1 Conditioning (Continued)

This section computes condition numbers for concrete problems using the derivative formula, and draws the lesson that recurs throughout the chapter: subtraction of nearly equal quantities is the fundamental ill-conditioned operation.

We have $F : X \to Y, \quad x + \delta x \to F(x) + \delta F(x)$.

$$\frac{\|\delta F(x)\|}{\|F(x)\|} \le \kappa(x) \frac{\|\delta x\|}{\|x\|} \qquad \text{for small } \|\delta x\|$$

$$\kappa(x) \le \frac{\|\delta F(x)\|}{\|F(x)\|} \, \frac{\|x\|}{\|\delta x\|}$$

Here, $\|\cdot\|$ are suitably chosen norms in X and Y.

Definition 3.1. $\kappa(x)$

$$\kappa(x) := \limsup_{\substack{\delta \to 0 \\ \|\delta x\| \le \delta}} \frac{\|\delta F(x)\|}{\|F(x)\|} \frac{\|x\|}{\|\delta x\|} \overset{\text{shorthand}}{=} \sup_{\delta x} \frac{\|\delta F(x)\|}{\|\delta x\|} \frac{\|x\|}{\|F(x)\|}$$

Definition 3.2. ***Ill-Conditioned, Well-Conditioned***
A problem is said to be *ill-conditioned* if $\kappa(x) \gg 1$ and *well-conditioned* otherwise.

How do we compute the condition number?

Remark 3.3. ***Conditioning Number for F Differentiable***
If $X \subset \mathbb{R}^n$, $Y \subset \mathbb{R}^m$ and $F : X \to Y$ is differentiable, then

$$\kappa(x) = \|DF(x)\| \frac{\|x\|}{\|F(x)\|}.$$

(Note: $\frac{\|\delta F(x)\|}{\|\delta x\|}$ is kind of like a difference quotient.)

Example 3.4.
As a first application of the formula, consider evaluating $y = F(x) = \sqrt{x}$ for $x > 0$. Here $n = m = 1$, the derivative is

$$F'(x) = \frac{1}{2} \frac{1}{\sqrt{x}},$$

and the formula gives

$$\kappa(x) = \left|F'(x)\right| \frac{|x|}{|F(x)|} = \frac{1}{2} \cdot \frac{1}{\sqrt{x}} \cdot \frac{x}{\sqrt{x}} = \frac{1}{2}.$$

Two things are worth noticing. First, the two factors of $\sqrt{x}$ cancel exactly, so the answer is a constant: the square root is well-conditioned *uniformly*, for every $x > 0$, even though $F'(x) \to \infty$ as $x \to 0^+$. A steep derivative alone does not make a problem ill-conditioned; what matters is the derivative *relative* to the sizes of input and output. Second, $\kappa = \frac{1}{2} < 1$ means the square root actually *dampens* relative errors: perturbing x by 0.1% moves $\sqrt{x}$ by only about 0.05%, as one also sees from $\sqrt{x(1+\epsilon)} \approx \sqrt{x}\,(1 + \frac{\epsilon}{2})$.

Example 3.5.
The second example is the most important in the chapter: computing $y = F(x) = x_1 - x_2$ for $x = \begin{bmatrix} x_1 & x_2 \end{bmatrix}^T \in \mathbb{R}^2$. Now $F : \mathbb{R}^2 \to \mathbb{R}$, and F is differentiable with a constant derivative,

$$DF = \begin{bmatrix} 1 & -1 \end{bmatrix}.$$

The condition-number formula (with the 2-norm on inputs) gives

$$\kappa(x) = \|DF(x)\|_2 \frac{\|x\|_2}{|F(x)|} = \frac{\sqrt{2}\sqrt{x_1^2 + x_2^2}}{|x_1 - x_2|}.$$

Read the fraction: the numerator measures the size of the *data*, the denominator the size of the *answer*. When $x_1 \approx x_2$ (and neither is near 0), the answer is far smaller than the data and κ blows up: the problem is ill-conditioned. Nothing exotic has happened; the mechanism is completely elementary. Perturb the inputs and watch:

$$\begin{aligned} F(x + \delta x) &= (x_1 + \delta x_1) - (x_2 + \delta x_2) = \underbrace{x_1 - x_2}_{\approx 0} + \delta x_1 - \delta x_2 \\ &\approx \delta x_1 - \delta x_2. \end{aligned}$$

The intended answer $x_1 - x_2$ is tiny, so the output consists almost entirely of the perturbations themselves: the shared leading digits of x_1 and x_2 cancel, and what remains is noise. This phenomenon is called "loss of significant digits," and we emphasize from the start that it is a property of the *problem*: no algorithm evaluating $x_1 - x_2$ for such inputs can do better.

Remark 3.6.
Conditioning is only a property of the problem to be solved, not of a specific algorithm for obtaining its solution.

To compute a condition number in Matlab, use `cond(A)`.

3.1.2 Floating-Point Numbers

To analyze algorithms as they actually run, we must fix how computers represent real numbers. This section reviews positional notation in base 2, the preliminary step to the floating-point format of the next section.
Problem: How do we represent $x \in \mathbb{R}$ on a computer?

Finite Storage: irrational numbers like $\sqrt{3} = 1.73205\ldots$ or even very large integers cannot be represented exactly!

Base of the representation: $\beta \geq 2$

- $\beta = 2$: binary representation
- $\beta = 10$: decimal representation

If $x \geq 0$ is an integer,

$$\begin{aligned} x &= b_n\beta^n + b_{n-1}\beta^{n-1} + \cdots + b_1\beta + b_0 \\ &=: (b_n b_{n-1} \cdots b_1 b_0)_\beta \end{aligned}$$

where the b_i's are integers such that $0 \leq b_i \leq \beta - 1$.

Example 3.7. ***53 in Base 10 and Base 2***
The same integer looks different in different bases, but the principle is identical: digits are coefficients of powers of the base. Let $x = 53$. In base 10 the familiar reading is

$$x = 5 \cdot 10^1 + 3 \cdot 10^0 = (53)_{10}.$$

In base 2, we express 53 as a sum of powers of two, $53 = 32 + 16 + 4 + 1$, and record which powers appear:

$$\begin{aligned} x &= 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \\ &= (110101)_2, \end{aligned}$$

which requires 6 bits. Note the trade-off: base 2 needs more digits than base 10 for the same number, but each digit is only a 0 or 1, which is exactly what hardware can store and switch reliably.

Example 3.8. ***How many numbers can we store?***
Suppose we allocate 52 bits and store numbers as plain binary integers,

$$x = (b_{51} b_{50} \cdots b_1 b_0)_2.$$

How many numbers can we represent exactly? Each bit takes two values independently, so there are 2^{52} patterns, representing $x = 0, 1, 2, \ldots, 2^{52} - 1$. That is a lot of integers, about 4.5×10^{15} of them, but notice what the scheme cannot do: it has no fractions, no negative numbers, nothing beyond $2^{52} - 1$, and its absolute spacing is fixed at 1 regardless of the size of the number. Scientific computation routinely needs 10^{-300} and 10^{300} in the same code, with comparable *relative* accuracy at both scales. This is why, in general, we use the floating-point representation, even for integers.

3.1.3 IEEE Floating-Point Standard

Virtually all hardware built since 1985 implements the IEEE 754 standard summarized here. Higham gives a complete account, including the history of the standard and the rationale for its design decisions.

Base $\beta = 2$. Any $x \in \mathbb{R}$, $x \neq 0$, can be represented in the binary form

$$x = \pm(1.b_1 b_2 \cdots) \times 2^p$$

where $b_1, b_2, \cdots \in \{0, 1\}$ and p is an integer.

3.1.4 Filling in the Details: Where the κ Formula Comes From

For differentiable F, Taylor gives $\delta F(x) = F(x + \delta x) - F(x) = DF(x)\,\delta x + o(\|\delta x\|)$. Insert into the definition of κ:

$$\frac{\|\delta F(x)\|}{\|\delta x\|} \cdot \frac{\|x\|}{\|F(x)\|} \longrightarrow \frac{\|DF(x)\,\delta x\|}{\|\delta x\|} \cdot \frac{\|x\|}{\|F(x)\|} \overset{\sup_{\delta x}}{=} \|DF(x)\| \, \frac{\|x\|}{\|F(x)\|},$$

using the definition of the induced matrix norm in the last step. So the recipe is: *differentiate, take norms, normalize.*

The subtraction example, dissected. For $F(x) = x_1 - x_2$:

$$\kappa(x) = \frac{\sqrt{2}\sqrt{x_1^2 + x_2^2}}{|x_1 - x_2|}.$$

Try $x_1 = 1.000001$, $x_2 = 1.000000$: $\kappa \approx 2 \times 10^6$. Perturb each input in the 7th digit and the *output* 10^{-6} can change completely. Nothing was computed wrong: the difference genuinely carries only the information beyond the shared leading digits. "Loss of significant digits" is thus a *conditioning* phenomenon; algorithms can avoid *creating* such subtractions (Section 3.4) but cannot repair one demanded by the problem itself.

3.1.5 Why Floating Point (and Not Fixed Point)?

With 52 bits as a plain integer we cover $0, 1, \ldots, 2^{52} - 1$: a range of about 4.5×10^{15}, with *absolute* spacing 1. Scientific computing needs 10^{-300} and 10^{300} in the same code.

Floating point $\pm(1.b_1 b_2 \cdots b_{52}) \times 2^p$ instead fixes the number of *significant digits* and lets the exponent place the binary point: the representable numbers have constant *relative* spacing $\approx 2^{-52}$. That matches how error propagates through multiplication and division, and it is why our entire analysis (conditioning, stability) is phrased in *relative* errors.

Note the leading "1." is forced (normalization), so it need not be stored, buying one free bit of precision. The bookkeeping of exponent bits, hidden bit, and special cases is next lecture's IEEE standard.

3.2 IEEE Floating-Point Numbers and Machine Precision

This lecture makes computer arithmetic concrete. We convert a decimal number to normalized binary form by hand, observing that most finite decimals become infinite binary expansions and must therefore be rounded. We describe the IEEE single- and double-precision formats, define machine precision $\epsilon_{\text{mach}} = 2^{-52}$, and prove the fundamental representation bound: every real number in range is stored with relative error at most $\frac{1}{2}\epsilon_{\text{mach}}$. This single inequality underlies every stability estimate in the rest of the course.

3.2.1 Quick Review of the IEEE Standard

We restate the format before filling in its details.

$\beta = 2, \quad x \in \mathbb{R}, \quad x \neq 0$

$$x = (1.b_1\, b_2 \ldots) \times 2^p$$

where $b_1, b_2, \cdots \in \{0, 1\}$ and p is an integer.

Example 3.9. ***41.7 in the IEEE standard***
We convert 41.7 to binary by hand, treating the integer and fractional parts separately. For the integer part 41, divide by 2 repeatedly and collect the remainders (these are the bits, least significant first). For the fractional part 0.7, multiply by 2 repeatedly; each integer part that appears is the next bit, most significant first:

$\frac{41}{2} = 20$	remainder 1	$2 \times 0.7 = 0.4$	remainder 1
$\frac{20}{2} = 10$	remainder 0	$2 \times 0.4 = 0.8$	remainder 0
$\frac{10}{2} = 5$	remainder 0	$2 \times 0.8 = 0.6$	remainder 1
$\frac{5}{2} = 2$	remainder 1	$2 \times 0.6 = 0.2$	remainder 1
$\frac{2}{2} = 1$	remainder 0	$2 \times 0.2 = 0.4$	remainder 0
$\frac{1}{2} = 0$	remainder 1		

Reading the remainders upward gives $41 = (101001)_2$, and reading the integer parts downward gives $0.7 = (0.10110\ldots)_2$. Observe that the fractional computation has begun to cycle: the value 0.4 has reappeared, so the bit pattern 0110 will repeat forever. Assembling and shifting the binary point to just after the leading 1:

$$\begin{aligned} 41.7 &= (101001.10110\ldots)_2 \\ &= +(1.01001100110\ldots) \times 2^5. \end{aligned}$$

This is its *normalized binary representation*. The point to remember: 41.7 is a perfectly ordinary one-decimal-digit number, yet its binary expansion is *infinite*. It cannot be stored exactly in any finite binary format; rounding will be forced before a single arithmetic operation takes place.

3.2.2 Normalized IEEE Floating-Point Numbers

This section works a complete conversion example and defines the normalized representation precisely. The example rewards following digit by digit: it shows that ordinary decimal inputs already require rounding before any arithmetic takes place.

$$\underbrace{\pm}_{\text{sign}}(1.\underbrace{b_1\, b_2\, \ldots\, b_N}_{\text{mantissa}}) \times 2^{\overbrace{p}^{\text{exponent}}}$$

where $b_1, b_2, \cdots \in \{0, 1\}$ and p can be represented with M bits.

Precision	sign	N (mantissa)	M (exponent)	total bits	total bytes
Single	1	23	8	32	4
Double	1	52	11	64	8
Long Double	1	64	15	80	10

From now on we stick with double precision.

A double floating-point number looks like:

$$\pm(1.b_1\, b_2\, \ldots\, b_{52}) \times 2^p$$

where p is an integer in $[-1022, 1023]$.

Some numbers:

$$1 := +1.\underbrace{0\,0\,\ldots\,0}_{52} \times 2^0$$
$$1 + 2^{-52} := +1.\underbrace{0\,0\,\ldots\,0}_{51}1 \times 2^0$$

3.2.3 Machine Precision

The resolution of the floating-point system is measured by a single number, machine epsilon. This section defines it and derives the bound on representation error that underlies every stability proof in the course.

$1 + 2^{-52}$ is the smallest floating-point number that is still larger than 1. To check this, we loop for `n=0,1,2,...`:

$$1 + 2^{-n} \overset{?}{>} 1$$

It should stop for $n = 53$.

Definition 3.10. ***Machine Precision***
Machine precision (or machine epsilon) is a measure of precision. Its value is

$$\epsilon_{\text{mach}} := 2^{-52} = (1 + 2^{-52} - 1)$$

(`eps` in Matlab)

Note:

$$\epsilon_{\text{mach}} = 2.220\ldots \times 10^{-16}$$

which is why double precision has ≈ 16 significant digits.

3.2.4 Floating-Point Representation

We collect the rounding rules and state the representation bound formally.

$$x \in \mathbb{R} \xrightarrow{\text{round to nearest}} \text{fl}(x) = \text{floating-point number closest to } x$$

Remark 3.11. ***Chopping vs. Rounding to Nearest***
Consider a real number with more than 52 fractional bits,

$$x = \pm 1.b_1\, b_2 \cdots b_{52}\, b_{53} \cdots \times 2^p.$$

The simplest way to fit it into the format is *chopping*: discard everything beyond b_{52}, i.e. $x \to \pm 1.b_1\, b_2 \cdots b_{52} \times 2^p$. Rounding down is the same as chopping; rounding up instead adds 1 to b_{52} (with carries). The IEEE standard prescribes the more accurate *round to nearest*, with ties broken toward an even last bit; the decision procedure is the following.

Algorithm 6 Round to Nearest (ties to even)

Require: $x = \pm 1.b_1\, b_2 \cdots b_{52}\, b_{53}\, b_{54} \cdots \times 2^p$
Ensure: the double-precision number nearest to x

if $b_{53} = 0$ **then**
    round down ▷ discarded tail $<$ half a unit in b_{52}
else if $b_j = 1$ for at least one $j > 53$ **then**
    round up ▷ discarded tail $>$ half a unit in b_{52}
else ▷ exact tie: tail is exactly half a unit
    if $b_{52} = 1$ **then**
        round up ▷ makes b_{52} even (0)
    else
        round down ▷ b_{52} already even
    end if
end if

Definition 3.12. ***Relative Error***
For $x \neq 0$, the relative error of $\mathrm{fl}(x)$ is

$$\left| \frac{\mathrm{fl}(x) - x}{x} \right| \leq \frac{1}{2}\epsilon_{\text{mach}} = 2^{-53} = 1.11\ldots \times 10^{-16}$$

Example 3.13. ***Representation of 41.7 with Rounding***
We now apply the rounding rule to 41.7, whose infinite binary expansion was computed above. Writing out the first bits and marking the cutoff after b_{52} with a vertical bar:

$$41.7 = +1.\underbrace{010011}_{6}\overbrace{\underbrace{0110}_{4}\,\underbrace{0110}_{4}\cdots\underbrace{0110}_{4}\,0\,\underset{b_{52}}{1}}^{11}\,\big|\,10\,0110\cdots \times 2^5$$

The first discarded bit is $b_{53} = 1$ and the tail beyond it is not all zeros, so the rule says round up:

$$\mathrm{fl}(41.7) = +1.010011\,0110\,0110\ldots0110\,01 \times 2^5.$$

The stored value differs from 41.7 by less than half a unit in the last stored place, a relative error below $2^{-53} \approx 10^{-16}$, invisible in ordinary use, but not zero, and we will watch this particular error resurface in the arithmetic examples of the next lecture.

3.2.5 Filling in the Details: Why the Conversion Algorithms Work

Integer part: repeated division. If $41 = b_5 2^5 + \cdots + b_1 2 + b_0$, then dividing by 2 leaves remainder b_0 and quotient $b_5 2^4 + \cdots + b_1$. Each division peels off the next bit, least significant first: $41 = (101001)_2$.

Fractional part: repeated multiplication. If $0.7 = b_{-1}2^{-1} + b_{-2}2^{-2} + \cdots$, then $2 \times 0.7 = 1.4$ has integer part $b_{-1} = 1$ and fractional part $0.4 = b_{-2}2^{-1} + \cdots$. Each doubling exposes the next bit, most significant first. Since 0.7's sequence of fractional parts cycles $(.7 \to .4 \to .8 \to .6 \to .2 \to .4)$, the bits repeat: $0.7 = (0.1\overline{0110})_2$. *Most finite decimals are infinite binaries*; rounding is unavoidable even for "nice" inputs.

The $\frac{1}{2}\epsilon_{\text{mach}}$ bound, derived. Let $x = (1.b_1 b_2 \cdots) \times 2^p$, so $2^p \le |x| < 2^{p+1}$. Representable numbers with exponent p form a uniform grid of spacing $2^p \cdot 2^{-52} = 2^{p-52}$. Rounding to nearest lands within half a grid step:

$$|\text{fl}(x) - x| \le \tfrac{1}{2} \cdot 2^{p-52} = 2^{p-53}.$$

Divide by $|x| \ge 2^p$:

$$\left|\frac{\text{fl}(x) - x}{x}\right| \le 2^{-53} = \tfrac{1}{2}\epsilon_{\text{mach}} \approx 1.11 \times 10^{-16}.$$

Equivalently $\text{fl}(x) = x(1+\epsilon)$, $|\epsilon| \le \frac{1}{2}\epsilon_{\text{mach}}$, the form we will use constantly in stability proofs.

(The "ties to even" clause, round toward $b_{52} = 0$ when exactly halfway, removes the statistical bias that "always round up" would inject over millions of operations.)

3.2.6 Orders of Magnitude Worth Memorizing

$\epsilon_{\text{mach}} = 2^{-52}$	$\approx 2.22 \times 10^{-16}$
$\frac{1}{2}\epsilon_{\text{mach}} = 2^{-53}$ (rounding bound)	$\approx 1.11 \times 10^{-16}$
significant decimal digits (double)	$\approx$ 15–16
largest double	$\approx 1.80 \times 10^{308}$
smallest normalized double 2^{-1022}	$\approx 2.23 \times 10^{-308}$

Sanity checks in MATLAB: `eps`, `realmax`, `realmin`; and try `0.1 + 0.2 == 0.3` (false!); now you know exactly why: 0.1, 0.2, 0.3 are all infinite binaries, each rounded separately.

3.3 Floating-Point Arithmetic and Round-Off Errors

We complete the description of IEEE arithmetic: the biased exponent, the special values, and the fundamental axiom of floating-point arithmetic, which asserts that each single operation is performed with relative error at most $\frac{1}{2}\epsilon_{\text{mach}}$. Individual operations are thus essentially perfect; large errors arise only from the way roundings propagate through many operations. A detailed example of catastrophic cancellation shows the canonical failure mode: subtraction of nearly equal quantities promotes old, previously harmless rounding errors into the leading digits.

3.3.1 Representation of the Exponent p

The exponent field has its own encoding, described in this section. The bias avoids a separate sign bit for the exponent and allows floating-point comparisons to be implemented with integer hardware.

Double precision uses 11 bits:

$$e_{10}2^{10} + e_9 2^9 + \cdots + e_1 2^1 + e_0 2^0, \qquad e_0, e_1, \ldots, e_{10} \in \{0, 1\}$$

We can represent all integers from 0 to $2^{11} - 1 = 2047$. BUT p is written in the shifted form:

$$p = \underbrace{-1023}_{\text{fixed}} + e_{10}2^{10} + e_9 2^9 + \cdots + e_1 2^1 + e_0 2^0$$

Thus, p ranges from -1023 to 1024.

3.3.2 Machine Representation

This section assembles the full 64-bit memory layout and the special values reserved by the standard.

$$\boxed{s \;\; e_0\, e_1 \;\; \cdots \;\; e_{10} \;\; b_1\, b_2 \;\; \cdots \;\; b_{52}}$$

$$\begin{aligned} x \neq \pm 0, \pm\infty: &\quad s \;\; e_0\, e_1 \ldots e_{10} \;\big|\; b_1\, b_2 \ldots b_{52} \\ x = \pm 0: &\quad s \;\; 0\,0 \ldots 0 \;\big|\; 0\,0 \ldots 0 \\ x = \pm\infty: &\quad s \;\; 1\,1 \ldots 1 \;\big|\; 0\,0 \ldots 0 \end{aligned}$$

Undefined Cases:

$$\begin{aligned}
\text{NaN}: &\quad s\ \ 1\,1\ldots1 \ \big|\ b_1\,b_2\ldots b_{52},\ \text{at least one } b_j = 1\\
\underbrace{\text{subnormal floating point numbers}}_{<2^{-1022}}: &\quad s\ \ 0\,0\ldots0 \ \big|\ b_1\,b_2\ldots b_{52},\ \text{at least one } b_j = 1\\
2^{-1022}: &\quad 1\ \ 1\,0\ldots0 \ \big|\ 0\,0\ldots0
\end{aligned}$$

3.3.3 Floating Point Arithmetic

Representation is only half of the standard; the other half prescribes the arithmetic. The requirement is clean: each operation must return the correctly rounded exact result. Its consequence, the fundamental axiom below, is the interface between hardware and analysis, and everything we prove about algorithms rests on it.

Example 3.14. $41.7 \oplus 10.425$
Let us watch one floating-point addition in complete detail. The two summands are

$$\begin{aligned}
x &= \mathrm{fl}(41.7) = 1.010011\,0110\,0110\ldots0110\,01 \times 2^5\\
y &= \mathrm{fl}(\underbrace{10.425}_{=41.7/4}) = 1.010011\,0110\,0110\ldots0110\,01 \times 2^3,
\end{aligned}$$

where y has the same bit pattern as x because $10.425 = 41.7/4$ and division by 4 only lowers the exponent by 2. To add, the hardware first *aligns the exponents*, rewriting y with exponent 5, which shifts its bits two places to the right; two of y's bits now extend past the 52-bit window. The exact sum is then computed in extended registers and rounded:

$$\begin{aligned}
x &= +1.0100110110\ldots011001 \ \big|\ \times 2^5\\
y &= +0.010100110110\ldots0110 \ \big|\ 01 \times 2^5\\
x+y &= \ \ 1.101000010\ldots00000 \ \big|\ 01 \times 2^5\\
\underbrace{\mathrm{fl}(x+y)}_{=x\oplus y\neq x+y} &= \ \ 1.101000010\ldots00000 \ \big|\ \times 2^5
\end{aligned}$$

The final rounding discards the two overhanging bits, so $x \oplus y \neq x + y$: even though both inputs were floating-point numbers and the hardware computed their exact sum internally, the *result* did not fit and had to be rounded. This is the general situation, and it is exactly what the fundamental axiom below quantifies: each operation returns the exact result correctly rounded, hence with relative error at most $\frac{1}{2}\epsilon_{\text{mach}}$.

Exact Arithmetic	floating-point arithmetic
$x + y$	$x \oplus y$
$x - y$	$x \ominus y$
$x \times y$	$x \otimes y$
x/y	$x \oslash y$

For all floating-point numbers x, y:

$$\begin{aligned} & x \oplus y = (x+y)(1+\epsilon) && \text{where } |\epsilon| \le \tfrac{1}{2}\epsilon_{\text{mach}} = 2^{-53} \\ & x \ominus y = (x-y)(1+\epsilon) && \text{where } |\epsilon| \le \tfrac{1}{2}\epsilon_{\text{mach}} \\ & x \otimes y = (x \times y)(1+\epsilon) && \text{where } |\epsilon| \le \tfrac{1}{2}\epsilon_{\text{mach}} \\ (y \neq 0) \quad & x \oslash y = (x/y)(1+\epsilon) && \text{where } |\epsilon| \le \tfrac{1}{2}\epsilon_{\text{mach}} \end{aligned}$$

Remark 3.15.
The relative error of the floating-point implementations of the four basic arithmetic operations is bounded by

$$\frac{1}{2}\epsilon_{\text{mach}} = 2^{-53} = 1.11\ldots \times 10^{-16}.$$

3.3.4 Catastrophic Effects of Round-Off Errors

The axiom guarantees each operation individually; it does not guarantee compositions of operations. This section presents the canonical failure, catastrophic cancellation, in a fully worked example.

Loss of Significant Digits
Occurs in subtraction of nearly equal numbers, i.e.

$$x - y, \quad \text{where } x \approx y.$$

Example 3.16. ***Relative Error of*** $x - y$ ***when*** $x \approx y$
The following pair of numbers is engineered so that every step can be followed bit by bit. Take

$$\begin{aligned} x &= 1 + 2^{-52} + 2^{-53} + 2^{-54} = +1.0\ldots01 \;\big|\; 110\cdots \times 2^0 \\ y &= 1 + 2^{-54} = +1.0\ldots00 \;\big|\; 010\cdots \times 2^0, \end{aligned}$$

two numbers agreeing in their first 52 bits; the bar marks the 52-bit storage cutoff. In exact arithmetic:

$$x - y = 2^{-52} + 2^{-53} = 3 \times 2^{-53}$$

Floating-point arithmetic:

$$\begin{aligned} \mathrm{fl}(x) &= +1.0\ldots010 \;\big|\; \times 2^0 \\ \mathrm{fl}(y) &= +1.0\ldots000 \;\big|\; \times 2^0 \\ \mathrm{fl}(x)-\mathrm{fl}(y) &= +0.0\ldots010 \;\big|\; \times 2^0 = +1.0\ldots0\times 2^{-51} \end{aligned}$$

Compare: the true difference is 3×2^{-53}, the computed difference is $2^{-51} = 4\times 2^{-53}$. Relative error:

$$\left|\frac{2^{-51}-3\times 2^{-53}}{3\times 2^{-53}}\right| = \frac{1}{3}.$$

A 33% error, from inputs each stored to 16 digits, and, remarkably, from a subtraction that was itself performed *exactly* (check: the stored bit patterns subtract with no rounding at all). The damage was done when x and y were rounded to 52 bits: rounding threw away exactly the bits in which x and y differ. The subtraction then cancelled the 52 correct shared bits and promoted what was left, which is mostly rounding error, into the leading position. This mechanism, old rounding errors elevated to prominence by cancellation, is the single most common way accuracy is lost in practice.

3.3.5 Filling in the Details: The Fundamental Axiom and How to Use It

Why is each operation this accurate? The hardware computes $x+y$ (etc.) *exactly* in extended internal registers, then rounds the exact result to the nearest double. So

$$x \oplus y = \mathrm{fl}(x+y) = (x+y)(1+\epsilon), \qquad |\epsilon| \le \tfrac{1}{2}\epsilon_{\mathrm{mach}},$$

by the representation bound from last lecture, and likewise for $\ominus, \otimes, \oslash$. **Each single operation is essentially perfect.** All large numerical errors are therefore *accumulation* or *amplification* effects, never a single bad operation.

Template for stability proofs (memorize this manipulation): a computed expression is unwound by attaching one $(1+\epsilon_i)$ per operation, e.g.

$$\mathrm{fl}\big((a\otimes b)\oplus c\big) = \big(ab(1+\epsilon_1)+c\big)(1+\epsilon_2) = ab(1+\epsilon_1)(1+\epsilon_2)+c(1+\epsilon_2),$$

then products of $(1+\epsilon_i)$'s are absorbed: $(1+\epsilon_1)(1+\epsilon_2) = 1+\epsilon_1+\epsilon_2+\epsilon_1\epsilon_2 =: 1+\eta$ with $|\eta| \le \epsilon_{\mathrm{mach}} + O(\epsilon_{\mathrm{mach}}^2)$. The computed value is the *exact* result

for slightly perturbed data, the seed of "backward error analysis" (Sections 3.5–3.6).

Warning: the axiom does *not* give associativity: $(x \oplus y) \oplus z \neq x \oplus (y \oplus z)$ in general. Summation order matters; compilers and parallel reductions can change your answers in the last digits.

3.3.6 Reading the Cancellation Example Quantitatively

$x = 1 + 2^{-52} + 2^{-53} + 2^{-54}$, $y = 1 + 2^{-54}$: each is stored with relative error $\leq 2^{-53}$, which is excellent. But the exact difference $x - y = 3 \times 2^{-53}$ is *tiny*, of the same size as the storage errors themselves. The stored values subtract *exactly* (no new rounding!) to 2^{-51}, yet

$$\text{relative error} = \left| \frac{2^{-51} - 3 \cdot 2^{-53}}{3 \cdot 2^{-53}} \right| = \frac{1}{3} \approx 33\%.$$

What went wrong: subtraction of nearly equal numbers cancels the correct leading bits and *promotes* the old rounding errors from the 53rd bit position to the 1st. The subtraction itself is backward stable (Section 3.5); the damage was done when the inputs were rounded. This distinction (ill-conditioned step vs unstable step) is the theme of the next three lectures.

3.4 Loss of Significant Digits; Stability

Having diagnosed cancellation, we first collect the standard remedies: rewriting expressions using conjugates, splitting the domain, and algebraic simplification to avoid huge intermediate quantities. We then begin the formal theory of algorithm quality. An algorithm is stable if it produces nearly the right answer to nearly the right problem; this deliberately modest requirement is the strongest property one can demand in the presence of ill-conditioning, and separating it from the conditioning of the problem itself is the key conceptual step of this unit.

3.4.1 Loss of Significant Digits

This section examines cancellation more closely and separates two issues that are often conflated: an ill-conditioned subtraction demanded by the problem, and an unnecessary cancellation introduced by the algorithm. The distinction determines whether the cure is a better algorithm or acceptance of the problem's sensitivity.

$$x - y, \qquad x \approx y$$

In practice, this problem can often be avoided easily.

Example 3.17. ***Trouble Is Avoidable***
Cancellation demanded by the problem cannot be cured, but cancellation introduced by a *formula* often can. Consider evaluating

$$f(x) = \frac{1 - \cos x}{\sin^2 x} \qquad \text{for } 0 < x < \pi.$$

The function itself is harmless: it is smooth and takes moderate values (indeed $f(x) \to \frac{1}{2}$ as $x \to 0$). The *formula*, however, computes $1 - \cos x$, and for $x \approx 0$ we have $\cos x \approx 1$: a subtraction of nearly equal numbers, with the cancellation catastrophe of the previous lecture waiting. The cure is algebra. Multiply by the conjugate:

$$f(x) = \frac{1 - \cos x}{\sin^2 x} \cdot \frac{1 + \cos x}{1 + \cos x} = \frac{1 - \cos^2 x}{\sin^2 x} \cdot \frac{1}{1 + \cos x} = \frac{1}{1 + \cos x}$$

The rewritten formula $\frac{1}{1+\cos x}$ contains an *addition* of positive quantities near $x = 0$: no cancellation, problem solved. But check the whole domain before declaring victory: near $x = \pi$ we have $\cos x \approx -1$, and now it is the *new* formula whose denominator $1 + \cos x$ suffers cancellation, while the original formula is perfectly safe there. So use each formula where it behaves:

$$f(x) = \begin{cases} \dfrac{1}{1 + \cos x} & 0 < x \le \dfrac{\pi}{2} \\ \dfrac{1 - \cos x}{\sin^2 x} & \dfrac{\pi}{2} < x < \pi. \end{cases}$$

This pattern, rewrite to remove the cancellation, then split the domain so each formula is used where it is safe, is a standard piece of numerical craftsmanship.

Example 3.18. ***Huge Intermediate Quantities***
Cancellation is not the only formula-induced disease; enormous intermediate values are another. Consider

$$f(x) = \frac{1}{x}\left(\frac{1}{x}\,\frac{1}{\frac{1}{x} - 1} - 1\right), \qquad 0 < x < 1,$$

to be evaluated at $x = 2^{-54}$. Note first what the answer *should* be. In exact arithmetic:

$$y = f(2^{-54}) = 2^{54}\left(\frac{2^{54}}{2^{54}-1} - 1\right) = \frac{1}{1-2^{-54}} \approx 1$$

In floating-point arithmetic:

$$\begin{aligned}
\tilde{y} &= 2^{54} \otimes \Big(\big(2^{54} \oslash \underbrace{(2^{54} \ominus 1)}\big) \ominus 1 \Big) \\
&= 2^{54} \otimes \Big(\underbrace{(2^{54} \oslash 2^{54})} \ominus 1 \Big) \\
&= 2^{54} \otimes \underbrace{(1 \ominus 1)} \\
&= 2^{54} \otimes 0 \\
&= 0
\end{aligned}$$

$$\text{Relative Error} = \left|\frac{\tilde{y} - y}{y}\right| = 1$$

A 100% error: the computed answer is 0 where the true answer is essentially 1. Trace the failure to its source: the subtraction $2^{54} \ominus 1$ returns 2^{54}, because near 2^{54} consecutive doubles are 4 apart and $2^{54} - 1$ is simply not representable. That one absorbed unit is a relative error of only 2^{-54}, harmless in itself, but the algorithm then divides two nearly equal huge numbers and subtracts 1, an operation exquisitely sensitive to exactly that lost unit. The huge intermediate quantity 2^{54}, absent from both the input and the output, is what set the trap.

Remedy: simplify the algebra *before* computing. The expression collapses:

$$f(x) = \frac{1}{x}\left(\frac{1}{\frac{1}{x} - 1}\right) = \frac{1}{1-x}$$

In floating-point arithmetic:

$$\tilde{\tilde{y}} = 1 \oslash (1 \ominus 2^{-54}) = 1 \oslash 1 = 1$$

$$\text{Relative Error} = \left|\frac{\tilde{\tilde{y}} - y}{y}\right| = \left|\frac{1 - \frac{1}{1-2^{-54}}}{\frac{1}{1-2^{-54}}}\right| = 2^{-54} < \epsilon_{\text{mach}}$$

Full accuracy, from the same inputs on the same machine. The two computations are mathematically identical; the difference is purely in which intermediate quantities they manufacture. The working rule: intermediate values much larger than both the input and the output should always raise an alarm.

3.4.2 Stability

We can now say what it means for an algorithm to be good. The definition must be forgiving enough to be achievable in floating-point arithmetic and strong enough to be useful; stability, defined below, is the accepted compromise. Our formulation follows Trefethen and Bau.

Abstract formulation of a problem:

$$\text{Input } x \in X \xrightarrow[F:X\to Y]{F} F(x) = y \in Y, \quad y \text{ is the output}$$

An algorithm in floating-point arithmetic:

$$\tilde{F} : X \to Y$$

$\tilde{F}$ captures *all* errors:

- $x \to \mathrm{fl}(x)$
- round-off errors of floating-point arithmetic
- possible approximation error (e.g. stopping Newton's method)

Exact solution of the problem: $y = F(x)$
Computed solution: $\tilde{y} = \tilde{F}(x)$

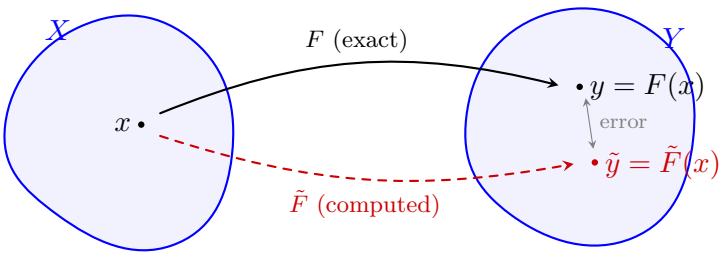


Figure 4: The exact solution $y = F(x)$ and the computed solution $\tilde{y} = \tilde{F}(x)$.

Relative error of the computed solution:

$$\frac{\|\tilde{y} - y\|}{\|y\|} = \frac{\left\|\tilde{F}(x) - F(x)\right\|}{\|F(x)\|}$$

Ideally we would like to have

$$\frac{\left\|\tilde{F}(x) - F(x)\right\|}{\|F(x)\|} = O(\epsilon_{\text{mach}}) \quad \text{for all } x \in X \tag{8}$$

Recall: $g(\epsilon) = O(\epsilon) \Leftrightarrow$ there exists a constant C such that $\|g(\epsilon)\| \leq C\,|\epsilon|$ for $\epsilon \to 0$.

If the condition number is 1, then (8) is possible. If the condition number is big, i.e. the problem is ill-conditioned, then (8) is unrealistic.

Instead...

Definition 3.19. ***Stable***
The "algorithm" $\tilde{F} : X \to Y$ for solving the problem is *stable* if for each $x \in X$, there exists an $\tilde{x} \in X$ such that

$$(1) \qquad \frac{\left\|\tilde{F}(x) - F(\tilde{x})\right\|}{\|F(\tilde{x})\|} = O(\epsilon_{\text{mach}})$$

$$(2) \qquad \frac{\|\tilde{x} - x\|}{\|x\|} = O(\epsilon_{\text{mach}})$$

Meaning of Stable
A stable algorithm gives nearly the right solution to nearly the right problem.

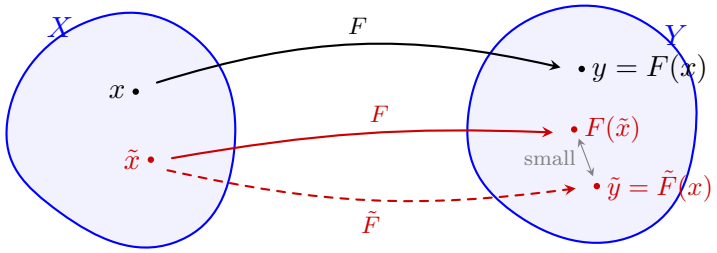


Figure 5: $\tilde{y} = \tilde{F}(x)$ is the computed solution.

$\tilde{y} = \tilde{F}(x)$ *is the computed solution.*

This is a realistic goal.

3.4.3 Filling in the Details: A Toolbox Against Cancellation

The examples illustrate general-purpose tricks; collect them:

1. **Multiply by the conjugate.** $1-\cos x = \dfrac{\sin^2 x}{1+\cos x}$; likewise $\sqrt{x+1}-\sqrt{x} = \dfrac{1}{\sqrt{x+1}+\sqrt{x}}$. The subtraction of close quantities is traded for a benign addition.

2. **Case-split the domain.** The rewritten formula may be bad elsewhere ($\cos x \approx -1$ near π): use each formula where it is well-behaved.

3. **Use series expansions near the bad point.** E.g. $1 - \cos x = \frac{x^2}{2} - \frac{x^4}{24} + \cdots$ for small x.

4. **Algebraic simplification first.** The huge-intermediate example collapses to $f(x) = \frac{1}{1-x}$: simplify *before* coding. Intermediate quantities much larger than input and output should raise an alarm.

Why $2^{54} \ominus 1 = 2^{54}$: the gap between consecutive doubles near 2^{54} is $2^{54-52} = 4$, so $2^{54} - 1$ is not representable and rounds back to 2^{54}. One absorbed unit out of 2^{54} is a relative error of only 2^{-54}, harmless *locally*. The loss occurs when the later subtraction and the multiplication by 2^{54} amplify precisely that lost unit into 100% error. Again: individually perfect operations, disastrous composition.

3.4.4 Parsing the Stability Definition

Data: problem $F : X \to Y$; implemented algorithm $\tilde{F} : X \to Y$ (includes input rounding, all round-off, any truncated iteration). Wish list:

$$\frac{\left\|\tilde{F}(x) - F(x)\right\|}{\|F(x)\|} = O(\epsilon_{\text{mach}}) \qquad (\text{“forward stability”}).$$

Unreachable in general: if $\kappa(x) \gg 1$, even rounding the *input* already moves the exact answer by $\kappa\,\epsilon_{\text{mach}}$. The definition of **stable** retreats gracefully: there exists nearby data $\tilde{x}$ ($\|\tilde{x} - x\| / \|x\| = O(\epsilon_{\text{mach}})$) with

$$\frac{\left\|\tilde{F}(x) - F(\tilde{x})\right\|}{\|F(\tilde{x})\|} = O(\epsilon_{\text{mach}}) :$$

nearly the right answer to *nearly* the right question. The algorithm is then blameless: any remaining error is the problem's sensitivity, not the method's

fault. Next lecture strengthens "nearly the right answer" to "exactly the right answer" (backward stability).

3.5 Backward Stability

This lecture introduces the sharper and more useful notion of backward stability: the computed result is the exact answer to a nearby problem. We prove that floating-point subtraction is backward stable. This proof is the prototype of all backward error analyses: each rounding is absorbed as a small relative perturbation of the data. We also exhibit a simple algorithm that is stable but not backward stable. The payoff of the definition is the accuracy theorem: for a backward stable algorithm, the relative error of the output is of order $\kappa(x)\,\epsilon_{\text{mach}}$, cleanly splitting responsibility between problem and algorithm.

3.5.1 Backward Stability

A stronger and cleaner property is backward stability: the computed answer is the exact answer to a nearby problem. Most core algorithms of numerical linear algebra are backward stable, a fact established by the style of proof introduced here. Both the idea and the technique are due to Wilkinson.

Definition 3.20. ***Backward Stability***
A stronger form of stability:
The "algorithm" $\tilde{F} : X \to Y$ for solving the problem $F : X \to Y$ is said to be *backward stable* if for each input there exists an $\tilde{x} \in X$ such that

$$\tilde{F}(x) = F(\tilde{x})$$

and

$$\frac{\|\tilde{x} - x\|}{\|x\|} = O(\epsilon_{\text{mach}})$$

Meaning of Backward Stability:
A backward stable algorithm gives the right solution to nearly the right problem.

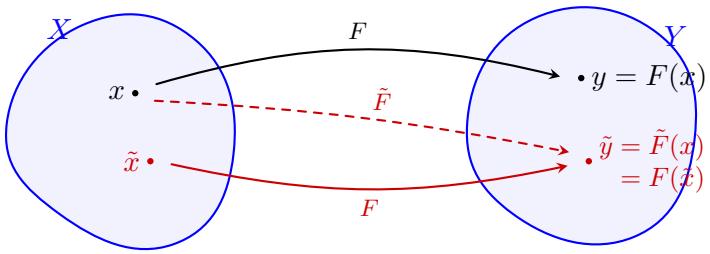


Figure 6: Backward stability: the computed solution is the exact solution of a nearby problem, $\tilde{y} = \tilde{F}(x) = F(\tilde{x})$.

Example 3.21. ***Backward Stability of Floating-Point Subtraction*** We verify the definition for the simplest nontrivial algorithm: subtract two real numbers in floating-point arithmetic. The problem and its implementation are

$$
\begin{aligned}
y = F(x) &= x_1 - x_2, \qquad x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \in \mathbb{R}^2 \\
\tilde{y} = \tilde{F}(x) &= \mathrm{fl}(x_1) \ominus \mathrm{fl}(x_2) \\
&= [x_1(1+\epsilon_1) - x_2(1+\epsilon_2)](1+\epsilon_3) \qquad \text{where } |\epsilon_1|, |\epsilon_2|, |\epsilon_3| \le \frac{1}{2}\epsilon_{\text{mach}} \\
&= \underbrace{x_1(1+\epsilon_1)(1+\epsilon_3)}_{=:\tilde{x}_1} - \underbrace{x_2(1+\epsilon_2)(1+\epsilon_3)}_{=:\tilde{x}_2} \\
&= F(\tilde{x})
\end{aligned}
$$

$$
\begin{aligned}
\left|\frac{\tilde{x}_1 - x_1}{x_1}\right| &= |(1+\epsilon_1)(1+\epsilon_3) - 1| = |\epsilon_1 + \epsilon_3 + \epsilon_1\epsilon_3| \\
&\le |\epsilon_1| + |\epsilon_3| + |\epsilon_1|\,|\epsilon_3| = O(\epsilon_{\text{mach}}) \\
\left|\frac{\tilde{x}_2 - x_2}{x_2}\right| &= O(\epsilon_{\text{mach}}) \qquad \text{by similar analysis}
\end{aligned}
$$

The floating-point algorithm for subtraction is backward stable. Pause on the decisive move, the third line of the first display: the roundings $\epsilon_1, \epsilon_2, \epsilon_3$ were not estimated, they were *relocated*, absorbed as multiplicative perturbations of the data x_1 and x_2. The computed value is then *exactly* $F(\tilde{x})$ for the perturbed data $\tilde{x}$, and the only thing left to check is that $\tilde{x}$ is close to x, which the second display does. This push-the-errors-onto-the-data maneuver is the template for every backward error analysis in the course. Note also how this resolves the cancellation puzzle of the last lecture: subtraction

is a backward stable *algorithm* applied to a possibly ill-conditioned *problem*; when $x_1 \approx x_2$, the large relative error in the output comes entirely from κ, not from the arithmetic.

Remark 3.22. ***Backward Stability of Floating-Point Addition, Multiplication, and Division***
Similar to Example 13.2, we can show that addition, multiplication, and division are backward stable.

Example 3.23. ***Stable, but not Backward Stable***
The two stability notions genuinely differ, and the gap is exposed by an algorithm of almost comic simplicity: add 1 to a number. The subtlety is that the constant 1 is part of the *problem*, not the data, so it is not available for absorbing errors. Formally:

$$
\begin{aligned}
&y = F(x) = x + 1, \qquad x \in \mathbb{R} \\
&\tilde{y} = \tilde{F}(x) = \mathrm{fl}(x) \oplus 1 = [x(1+\epsilon_1) + 1](1+\epsilon_2) \qquad \text{where } |\epsilon_1|, |\epsilon_2| \le \frac{1}{2}\epsilon_{\text{mach}} \\
&\quad = \underbrace{x(1+\epsilon_1)(1+\epsilon_2) + \epsilon_2}_{=:\tilde{x}} + 1 = \tilde{x} + 1 = F(\tilde{x})
\end{aligned}
$$

$$
\begin{aligned}
\left| \frac{\tilde{x} - x}{x} \right| &= \left| \frac{x(1+\epsilon_1)(1+\epsilon_2) + \epsilon_2 - x}{x} \right| \\
&= \left| \epsilon_1 + \epsilon_2 + \epsilon_1\epsilon_2 + \frac{\epsilon_2}{x} \right| = O(\epsilon_{\text{mach}}) + O\left(\frac{\epsilon_{\text{mach}}}{|x|} \right) \\
&\neq O(\epsilon_{\text{mach}}) \quad \text{if } x \approx 0
\end{aligned}
$$

Thus, this algorithm is not backward stable: the attempt to blame the rounding ϵ_2 on the data x leaves an *additive* residue ϵ_2, and for $x \approx 0$ this residue is enormous relative to x; there is simply nowhere to hide the error when the data is tiny and the constant 1 is off-limits. Is the algorithm at least stable? Yes: we exhibit $\tilde{x}$ meeting the weaker definition, which permits

the answer to be *nearly* (rather than exactly) $F(\tilde{x})$.

$$F(x) = x + 1$$
$$\tilde{F}(x) = \underbrace{x(1+\epsilon_1)}_{=:\tilde{x}}(1+\epsilon_2) + \epsilon_2 + 1$$
$$\tilde{F}(\tilde{x}) - F(\tilde{x}) = (\tilde{x}+1)(1+\epsilon_2) - (\tilde{x}+1) = (\tilde{x}+1)\epsilon_2 = F(\tilde{x})\epsilon_2$$
$$\left|\tilde{F}(x) - F(\tilde{x})\right| = |F(\tilde{x})|\,|\epsilon_2| = |F(\tilde{x})|\,O(\epsilon_{\text{mach}})$$
$$\tilde{x} - x = x\epsilon_1$$
$$|\tilde{x} - x| = |x|\,O(\epsilon_{\text{mach}})$$

Thus, it is stable. The example calibrates the two definitions: backward stability demands that all errors be expressible as data perturbations, which fails here for smallness of the data, while stability tolerates a residual output error of relative size $O(\epsilon_{\text{mach}})$, which is all that remains. In practice this algorithm is perfectly satisfactory; the example matters not as a warning but as proof that the implication "backward stable $\Rightarrow$ stable" cannot be reversed.

3.5.2 Accuracy of Backward Stable Algorithms

Backward stability pays off through the theorem of this section, which combines a backward error bound with a condition number to produce a forward error bound. The result explains the division of labor: the algorithm controls the backward error, and the problem sets the amplification factor.

Theorem 3.24.
Let the problem $F : X \to Y$ have condition number $\kappa = \kappa(x)$, $x \in X$, and let $\tilde{F} : X \to Y$ be a backward stable algorithm for solving the problem. Then:

$$\underbrace{\frac{\left\|\tilde{F}(x) - F(x)\right\|}{\|F(x)\|}}_{=\frac{\|\tilde{y}-y\|}{\|y\|}} = O(\kappa(x)\epsilon_{\text{mach}}) \quad \text{for all } x \in X.$$

Thus, if the condition number is really high, then even a backward stable algorithm cannot save you.

3.5.3 Filling in the Details: The Proof Technique on Display

The subtraction proof is the prototype backward error analysis. Watch the moves:

$$\begin{aligned}
\tilde{F}(x) &= \mathrm{fl}(x_1) \ominus \mathrm{fl}(x_2) && \text{write what is computed} \\
&= \big[x_1(1+\epsilon_1) - x_2(1+\epsilon_2)\big](1+\epsilon_3) && \text{one } (1+\epsilon) \text{ per op} \\
&= \underbrace{x_1(1+\epsilon_1)(1+\epsilon_3)}_{\tilde{x}_1} - \underbrace{x_2(1+\epsilon_2)(1+\epsilon_3)}_{\tilde{x}_2} && \textit{push errors onto data} \\
&= F(\tilde{x}).
\end{aligned}$$

The art is the third line: every rounding is absorbed as a small *multiplicative* perturbation of an input. Then $(1+\epsilon_1)(1+\epsilon_3) = 1 + \epsilon_1 + \epsilon_3 + \epsilon_1\epsilon_3$, so $|\tilde{x}_i - x_i| \,/\, |x_i| \le \epsilon_{\text{mach}} + \frac{1}{4}\epsilon_{\text{mach}}^2 = O(\epsilon_{\text{mach}})$.

Why $F(x) = x + 1$ fails. The rounding ϵ_2 from the addition must be blamed on the data x, giving $\tilde{x} = x(1+\epsilon_1)(1+\epsilon_2) + \epsilon_2$, an *additive* contamination ϵ_2. Relative to x:

$$\left|\frac{\tilde{x} - x}{x}\right| = O(\epsilon_{\text{mach}}) + O\left(\frac{\epsilon_{\text{mach}}}{|x|}\right) \;\longrightarrow\; \infty \quad \text{as } x \to 0.$$

The constant 1 cannot absorb error (it is exact by definition of the problem), so near $x = 0$ there is nowhere to hide the rounding. The algorithm is still *stable*: the computed value equals $F(\tilde{x})(1+\epsilon_2)$, i.e. nearly the right answer to a nearby problem, just not *backward* stable. Moral: backward stability is the stronger, cleaner property, and luckily the one that real linear-algebra algorithms tend to have.

3.5.4 The Error Budget Picture

The accuracy theorem separates responsibilities:

$$\underbrace{\frac{\|\tilde{y} - y\|}{\|y\|}}_{\text{observed error}} \;=\; O\Big(\underbrace{\kappa(x)}_{\text{problem}} \cdot \underbrace{\epsilon_{\text{mach}}}_{\text{algorithm delivers}} \Big).$$

Practical readings:

- Backward stable algorithm + well-conditioned problem $\Rightarrow$ answers correct to ≈ 16 digits.
- $\kappa = 10^{10} \Rightarrow$ expect only ≈ 6 digits, *and no algorithm can beat this*, because rounding the input alone already commits this error.

- An algorithm losing digits on a *well-conditioned* problem is unstable and should be replaced (cf. LU without pivoting).

Proof of the theorem: next lecture, one chain of inequalities.

3.6 Backward Stability; Vector and Matrix Norms

We prove the accuracy theorem stated last time and then develop the normed-space machinery that quantitative linear algebra requires: the classical vector norms, the Frobenius norm, and the induced (least-upper-bound) matrix norms. We compute the induced 1- and ∞-norms explicitly as maximal column and row sums, note that the induced 2-norm is an eigenvalue quantity with no entrywise formula, and prove submultiplicativity, the property that makes induced norms the right tool for perturbation arguments.

3.6.1 Backward Stability (Continued)

This section proves the accuracy theorem and begins the study of the norms needed to apply it to matrix problems.

Theorem 3.25.

$$\begin{gathered} F : X \to Y, \quad \kappa(x) \\ \tilde{F} : X \to Y \text{ backward stable} \\ \frac{\left\| \tilde{F}(x) - F(x) \right\|}{\|F(x)\|} = O(\kappa(x)\epsilon_{\text{mach}}) \quad \forall\, x \in X \end{gathered}$$

Proof. Let $x \in X$ and let $\tilde{F}$ be backward stable. Then

$$\tilde{F}(x) = F(\tilde{x}) \quad \text{for some } \tilde{x} \in X \text{ with } \frac{\|\tilde{x} - x\|}{\|x\|} = O(\epsilon_{\text{mach}}).$$

Recall:

$$\kappa(x) = \sup_{\delta x} \frac{\|\delta F(x)\|}{\|\delta x\|} \cdot \frac{\|x\|}{\|F(x)\|}$$

Then

$$\begin{aligned}
\frac{\left\|\tilde{F}(x) - F(x)\right\|}{\|F(x)\|} &= \frac{\|F(\tilde{x}) - F(x)\|}{\|F(x)\|} && \text{(backward stability)} \\
&= \frac{\|\delta F(x)\|}{\|F(x)\|} && (\delta x := \tilde{x} - x) \\
&= \left(\frac{\|\delta F(x)\|}{\|\delta x\|} \cdot \frac{\|x\|}{\|F(x)\|}\right) \frac{\|\tilde{x} - x\|}{\|x\|} \\
&\leq \big(\kappa(x) + \underbrace{o(1)}_{\to 0 \text{ as } \epsilon_{\text{mach}} \to 0}\big) \underbrace{\frac{\|\tilde{x} - x\|}{\|x\|}}_{= O(\epsilon_{\text{mach}})} \\
&= O(\kappa(x)\epsilon_{\text{mach}})
\end{aligned}$$

□

3.6.2 Norms

Error analysis for vector-valued problems requires a way to measure the size of vectors and matrices. This section reviews the standard vector norms; Golub and Van Loan tabulate their properties and relationships.

Definition 3.26. ***Norm***
A norm on $\mathbb{C}^n$ is a function

$$\|\cdot\| : \mathbb{C}^n \to \mathbb{R}$$

such that for all $x, y \in \mathbb{C}^n$, $\alpha \in \mathbb{C}$:

1. $\|x\| \geq 0$, and $\|x\| = 0$ if and only if $x = 0$
2. $\|x + y\| \leq \|x\| + \|y\|$
3. $\|\alpha x\| = |\alpha| \, \|x\|$

Example 3.27. ***Some Norms***

$$\|x\|_1 = \sum_{i=1}^{n} |x_i|$$

$$\|x\|_2 = \sqrt{\sum_{i=1}^{n} |x_i|^2}$$

$$\|x\|_\infty = \max_{1 \le i \le n} |x_i|$$

Example 3.28. ***Vec Norm***
Let $A \in \mathbb{C}^{n \times n}$. $\text{vec}(A) := [\, a_{11}\ a_{21}\ \cdots\ a_{n1}\ a_{12}\ \cdots\ a_{nn}\,]^T \in \mathbb{C}^{n^2}$. $\|A\| := \|\text{vec}(A)\|$, where $\|\cdot\| : \mathbb{C}^{n^2} \to \mathbb{R}$.

Example 3.29. ***Frobenius Norm***

$$\|A\|_F = \|\text{vec}(A)\|_2 = \sqrt{\sum_{j,k=1}^{n} |a_{jk}|^2}$$

3.6.3 Matrix Norms

Among matrix norms, the induced (operator) norms interact best with products, through the submultiplicativity property proved below. The three classical induced norms and the Frobenius norm cover nearly every need in practice.

Definition 3.30. ***Matrix Norm***
Any matrix norm satisfies:

1. $\|A\| \ge 0$, with $\|A\| = 0$ if and only if $A = 0$
2. $\|A + B\| \le \|A\| + \|B\|$
3. $\|\alpha A\| = |\alpha|\, \|A\|$

Definition 3.31. ***Induced Matrix Norm***
Often more useful are induced matrix norms:
Let $\|\cdot\| : \mathbb{C}^n \to \mathbb{R}$ be a matrix norm on $\mathbb{C}^n$. Set

$$\text{lub}(A) = \text{lub}_{\|\cdot\|}(A) := \max_{x \ne 0} \frac{\|Ax\|}{\|x\|} = \max_{\|x\|=1} \|Ax\|$$

It is easy to see that $\text{lub}(A)$ is indeed a matrix norm.

Note: $\|Ax\| \leq \text{lub}_{\|\cdot\|}(A)\,\|x\|$ for all $A \in \mathbb{C}^{n\times n}$, $x \in \mathbb{R}^n$.

Norm on $\mathbb{C}^n$	Induced Matrix Norm
$\|\cdot\|_1$	$\|A\|_1 := \text{lub}_{\|\cdot\|_1}(A) = \max\limits_{1\leq k\leq n} \sum\limits_{j=1}^{n} \|a_{jk}\|$ (max column sum)
$\|\cdot\|_\infty$	$\|A\|_\infty := \text{lub}_{\|\cdot\|_\infty}(A) = \max\limits_{1\leq j\leq n} \sum\limits_{k=1}^{n} \|a_{jk}\|$ (max row sum)
$\|\cdot\|_2$	$\|A\|_2 := \text{lub}_{\|\cdot\|_2}(A) = \max\limits_{\|x\|_2=1} \|Ax\|_2 = \sigma_{\max}(A)$ = the largest singular value of A

$$\sigma_{\max} = \sqrt{\lambda_{\max}(A^H A)}$$

Theorem 3.32. ***lub Norms Are Submultiplicative***
lub norms are submultiplicative:

$$\text{lub}(AB) \leq \text{lub}(A)\,\text{lub}(B) \qquad \forall\, A, B \in \mathbb{C}^{n\times n}$$

Proof.

$$\text{lub}(AB) = \max_{x\neq 0} \frac{\|ABx\|}{\|x\|}$$

For $x \neq 0$, $y := Bx \neq 0$

$$\frac{\|ABx\|}{\|x\|} = \frac{\|ABx\|}{\|Bx\|} \cdot \frac{\|Bx\|}{\|x\|} = \frac{\|Ay\|}{\|y\|} \frac{\|Bx\|}{\|x\|} \leq \text{lub}(A)\,\text{lub}(B)$$

□

3.6.4 Filling in the Details: The 1-Norm Formula, Derived

Claim: $\|A\|_1 = \max_k \sum_j |a_{jk}|$ (max column sum). Write $A = [a_1\ a_2\ \cdots\ a_n]$ by columns.

Upper bound. For any x with $\|x\|_1 = \sum_k |x_k| = 1$:

$$\|Ax\|_1 = \Big\|\sum_k x_k a_k\Big\|_1 \leq \sum_k |x_k|\ \|a_k\|_1 \leq \Big(\max_k \|a_k\|_1\Big) \underbrace{\sum_k |x_k|}_{=1}.$$

Attained. Let k^* maximize $\|a_k\|_1$ and take $x = e_{k^*}$: then $\|Ax\|_1 = \|a_{k^*}\|_1$ hits the bound. Hence equality.

Dually, $\|A\|_\infty$ is the max *row* sum (same argument applied to rows / to A^T). Both are $O(n^2)$ to evaluate, so both are cheap diagnostics.

The 2-norm is different. No entrywise formula:

$$\|A\|_2 = \sigma_{\max}(A) = \sqrt{\lambda_{\max}(A^H A)},$$

an eigenvalue computation (cf. Chapter 7). Useful special case: for unitary U, $\|U\|_2 = 1$.

Annotating the accuracy-theorem proof: the chain was

$$\frac{\left\|\tilde{F}(x) - F(x)\right\|}{\|F(x)\|} \overset{\text{(i)}}{=} \frac{\|F(\tilde{x}) - F(x)\|}{\|F(x)\|} \overset{\text{(ii)}}{=} \left(\frac{\|\delta F\|}{\|\delta x\|}\frac{\|x\|}{\|F(x)\|}\right)\frac{\|\tilde{x} - x\|}{\|x\|}$$
$$\overset{\text{(iii)}}{\leq} \big(\kappa(x) + o(1)\big)O(\epsilon_{\text{mach}}).$$

(i) backward stability substitutes the data; (ii) is multiply-and-divide; (iii) the bracket is a difference quotient whose sup defines κ, up to $o(1)$ because $\delta x \to 0$ as $\epsilon_{\text{mach}} \to 0$. Every backward-stable algorithm inherits this bound with *no further work*.

3.6.5 Which Norm When?

- **All norms on $\mathbb{C}^n$ are equivalent:** e.g. $\|x\|_\infty \leq \|x\|_2 \leq \|x\|_1 \leq n\,\|x\|_\infty$. So *qualitative* statements ($O(\epsilon_{\text{mach}})$, convergence) are norm-independent; choose whichever is convenient.
- **Constants do change:** condition numbers $\kappa_1, \kappa_2, \kappa_\infty$ can differ by factors of n. Report which norm you use.
- $\|\cdot\|_1, \|\cdot\|_\infty$: trivially computable, good for bounds and software.
- $\|\cdot\|_2$: unitarily invariant ($\|UAV\|_2 = \|A\|_2$ for unitary U, V), the natural norm for the eigenvalue/orthogonality part of the course.
- $\|\cdot\|_F$: also unitarily invariant, computable, but *not* induced by any vector norm (e.g. $\|I\|_F = \sqrt{n} \neq 1$).

3.7 Notes and References

Conditioning and backward stability in the form used here are the organizing ideas of Trefethen and Bau, whose treatment shaped this chapter. Higham's *Accuracy and Stability of Numerical Algorithms* is the definitive reference for floating-point error analysis, including the IEEE 754 standard, the fundamental axiom, and backward error analyses of the standard algorithms. The technique of pushing rounding errors onto the data originates with Wilkinson.

3.8 Exercises

1. Convert 0.3 and 5.6 to normalized binary floating-point form, exhibiting the repeating bit patterns. Which decimal fractions have terminating binary expansions?

2. Compute the condition number $\kappa(x)$ for $F(x) = e^x$, $F(x) = \ln x$, and $F(x) = x_1 + x_2$. For each, identify the inputs (if any) at which the problem is ill-conditioned, and interpret the answer for $\ln x$ near $x = 1$.

3. Using the fundamental axiom, show that the computed sum $\mathrm{fl}(x_1 + x_2 + \cdots + x_n)$ (summed left to right) equals $\sum_j x_j(1+\eta_j)$ with $|\eta_j| \le (n-1)\epsilon_{\mathrm{mach}} + O(\epsilon_{\mathrm{mach}}^2)$. Conclude that recursive summation is backward stable.

4. Show in detail that the algorithm $\tilde{F}(x) = \mathrm{fl}(x) \oplus 1$ for $F(x) = x + 1$ is stable but not backward stable, following the argument of the text. Where exactly does the attempt to push ϵ_2 onto the data fail?

5. Prove that the induced matrix norm satisfies $\|A\|_\infty = \max_j \sum_k |a_{jk}|$ (maximum row sum), by adapting the column-sum proof given for $\|A\|_1$.

6. Assuming the singular value decomposition $A = U\Sigma V^H$, show that $\|A\|_2 = \sigma_{\max}$ and $\left\|A^{-1}\right\|_2 = 1/\sigma_{\min}$, so that $\kappa_2(A) = \sigma_{\max}/\sigma_{\min}$. Deduce that $\kappa_2(U) = 1$ for unitary U.

7. The roots of $x^2 + bx + c = 0$ for large $b > 0$ are $x_\pm = \frac{1}{2}(-b \pm \sqrt{b^2 - 4c})$. Explain why the formula for x_+ suffers cancellation, rewrite it using $x_+ x_- = c$, and compare both formulas numerically for $b = 10^8$, $c = 1$.

8. (Computing.) Determine ϵ_{mach} experimentally: find the smallest n for which $\mathrm{fl}(1 + 2^{-n}) = 1$ in double precision. Compare with `eps` in MATLAB or `numpy.finfo(float).eps`.

Chapter 4

Stability of Gaussian Elimination

With the framework in place, we return to the algorithm of the first unit and analyze it: the condition number of $Ax = b$ is $\kappa(A) = \|A\| \left\|A^{-1}\right\|$, and the backward stability of LU factorization with partial pivoting hinges on a single quantity, the growth factor.

4.1 Conditioning of $Ax = b$; Stability of LU Factorization

The tools of the previous lecture are now applied to the central problem $Ax = b$. A first-order perturbation argument identifies the condition number as $\kappa(A) = \|A\| \left\|A^{-1}\right\|$, and we interpret its extreme values geometrically. We then return to Gaussian elimination and assemble its stability story: the backward error of the computed factorization is governed by the size of $\|L\| \|U\|$ relative to $\|A\|$, which pivoting controls through the bound $|l_{jk}| \leq 1$ and the growth factor $\rho(A)$.

4.1.1 Conditioning of $Ax = b$

We apply the framework to the central problem of the course. The answer is the most quoted formula of numerical linear algebra, $\kappa(A) = \|A\| \left\|A^{-1}\right\|$, and its derivation is a short perturbation argument.

Let $A \in \mathbb{C}^{n\times n}$ be nonsingular.
Let $b \in \mathbb{C}^n$ be fixed, and consider small perturbations δA of A.

$$\left.\begin{aligned} Ax &= b \\ (A+\delta A)(x+\delta x) &= b \end{aligned}\right\} \;\Rightarrow\; A(\delta x) + (\delta A)x + \underbrace{(\delta A)(\delta x)}_{\approx 0} = 0$$

$$A(\delta x) \approx -(\delta A)x \;\Rightarrow\; \delta x \approx -A^{-1}(\delta A)x$$

Let $\|\cdot\|$ be any vector norm on $\mathbb{C}^n$ and $\|\cdot\|$ be the associated lub norm on $\mathbb{C}^{n\times n}$.

$$\begin{aligned} \|\delta x\| &\approx \left\|A^{-1}(\delta A)x\right\| \leq \left\|A^{-1}(\delta A)\right\| \|x\| \\ &\leq \left\|A^{-1}\right\| \|\delta A\| \|x\| \end{aligned}$$

$$\frac{\|\delta x\|}{\|x\|} \frac{\|A\|}{\|\delta A\|} \lessapprox \|A\| \left\|A^{-1}\right\| =: \kappa(A) \quad \left(= \kappa_{\|\cdot\|}(A)\right)$$

Recall our definition of $\kappa(x)$:

$$\sup_{\delta A} \frac{\|\delta x\|}{\|x\|} \frac{\|A\|}{\|\delta A\|} = \kappa(A) = \|A\| \left\|A^{-1}\right\|$$

Theorem 4.1.
Let $b \in \mathbb{C}^n$ be fixed. Let $\|\cdot\|$ be a vector norm on $\mathbb{C}^n$ and let $\|\cdot\|$ be the associated lub norm. Let A be nonsingular. The conditioning number of solving $Ax = b$ is given by

$$\kappa(A) = \|A\| \left\|A^{-1}\right\|.$$

Corollary 4.2.
$\kappa(A) \geq 1$ for all A (since we are using lub norms).

Proof.

$$\begin{aligned} I &= AA^{-1} \\ \|I\| &= \left\|AA^{-1}\right\| \\ &\leq \|A\| \left\|A^{-1}\right\| \\ &= \kappa(A) \end{aligned}$$

$$\|I\| = \max_{\|x\|=1} \|Ix\| = 1$$

□

4.1.2 Stability of LU Factorization

With the conditioning of the problem settled, the remaining question is the stability of the algorithm. This section explains why LU factorization without pivoting fails the test, in precisely the sense the definitions anticipate.

Let $A \in \mathbb{C}^{n \times n}$ be nonsingular. Assume that A has an LU factorization: $A = LU$.

Then: for all "sufficiently small" ϵ_{mach}, LU factorization without partial pivoting in floating-point arithmetic encounters no zero pivots. The algorithm will run to completion, computing a lower-triangular matrix $\tilde{L}$ and an upper-triangular matrix $\tilde{U}$ such that

$$\tilde{L}\tilde{U} = A + \delta A \qquad \text{and} \qquad \frac{\|\delta A\|}{\|L\| \cdot \|U\|} = O(\epsilon_{\text{mach}}).$$

If $\|L\| \cdot \|U\| = O(\|A\|)$, then

$$\frac{\|\delta A\|}{\|A\|} = O(\epsilon_{\text{mach}})$$

and LU factorization without pivoting is backward stable.

However, for general nonsingular $A \in \mathbb{C}^{n\times n}$,

$$\|L\| \cdot \|U\| \neq O(\|A\|)$$

In fact, without pivoting $\|L\|$ can be arbitrarily large (see the 2×2 example we did).

$\Rightarrow$ LU factorization without pivoting is *not* backward stable (and not even stable).

4.1.3 LU Factorization with Partial Pivoting

Partial pivoting bounds the factor L; whether the algorithm is backward stable then reduces to a single quantity, the growth factor, defined here. The complete story, including Wilkinson's analysis and the modern average-case perspective, is told in Higham.

Partial pivoting ensures that $|l_{jk}| \leq 1$ for all $j, k \Rightarrow \|L\| = O(1)$. But what can we say about $\|U\|$?

Define the so-called growth factor.

Definition 4.3. ***Growth Factor***

$$\rho = \rho(A) = \frac{\max\limits_{j,k=1,2,\ldots,n} |u_{jk}|}{\max\limits_{j,k=1,2,\ldots,n} |a_{jk}|}$$

Theorem 4.4.
Let $A \in \mathbb{C}^{n\times n}$ be a nonsingular matrix. Then LU factorization with partial pivoting in floating-point arithmetic generates matrices $\tilde{L}$, $\tilde{U}$, and $\tilde{P}$ such that

$$\tilde{L}\tilde{U} = \tilde{P}A + \delta A$$

and $\dfrac{\|\delta A\|}{\|A\|} = O(\rho\, \epsilon_{\text{mach}})$, where $\rho = \rho(A)$ is the growth factor.

Consequence:
If $\rho(A)$ is bounded by a constant for all nonsingular matrices $A \in \mathbb{C}^{n\times n}$,

then

$$\frac{\|\delta A\|}{\|A\|} = O(\epsilon_{\text{mach}})$$

and thus, LU factorization with partial pivoting is backward stable.

This is indeed the case. It can be shown that

$$\rho(A) \leq 2^{n-1} \qquad \text{for all nonsingular matrices } A \in \mathbb{C}^{n\times n}$$

and the inequality is sharp. However, in practice, $\rho(A)$ does not approach this bound.

4.1.4 Filling in the Details: The Perturbation Derivation, Line by Line

Perturb A only (b fixed): $(A + \delta A)(x + \delta x) = b$. Expand and subtract $Ax = b$:

$$A\,\delta x + (\delta A)\,x + (\delta A)(\delta x) = 0.$$

The term $(\delta A)(\delta x)$ is a product of two small quantities, second order, and is dropped (this is exactly the linearization that the $\limsup_{\delta\to 0}$ in the definition of κ makes rigorous). Solve:

$$\delta x \approx -A^{-1}(\delta A)x \;\Rightarrow\; \|\delta x\| \leq \left\|A^{-1}\right\|\,\|\delta A\|\,\|x\|\,,$$

using $\|MN\| \leq \|M\|\,\|N\|$ (lub norms, last lecture). Normalize:

$$\frac{\|\delta x\|}{\|x\|} \leq \left\|A^{-1}\right\|\,\|\delta A\| = \underbrace{\|A\|\,\left\|A^{-1}\right\|}_{\kappa(A)}\,\frac{\|\delta A\|}{\|A\|}.$$

Sharpness: one can choose δA aligning with the worst directions of A^{-1} so the bound is attained: $\kappa(A)$ is the condition number itself, not merely an upper bound.

Geometric reading (2-norm). $\kappa_2(A) = \sigma_{\max}/\sigma_{\min}$: the ratio of the largest to the smallest stretch of the map $x \mapsto Ax$. $\kappa_2 = 1$ exactly for (multiples of) unitary matrices, whose image spheres are perfectly round. $\kappa_2 \to \infty$ as A approaches singularity: indeed $1/\kappa_2(A)$ equals the relative distance from A to the nearest singular matrix.

Perturbing b instead: $A\,\delta x = \delta b$ gives the same constant, $\frac{\|\delta x\|}{\|x\|} \leq \kappa(A)\frac{\|\delta b\|}{\|b\|}$ (use $\|b\| \leq \|A\|\,\|x\|$). Either data channel is amplified by at most $\kappa(A)$.

4.1.5 Assembling the Stability Story for LU

Backward error of the factorization: $\tilde{L}\tilde{U} = A + \delta A$ with

$$\frac{\|\delta A\|}{\|L\|\,\|U\|} = O(\epsilon_{\text{mach}}).$$

So everything hinges on whether $\|L\|\,\|U\|$ is comparable to $\|A\|$:

- **No pivoting:** $\|L\|$ unbounded (the 10^{20} multiplier) $\Rightarrow$ δA can be enormous relative to $A \Rightarrow$ *not stable*.
- **Partial pivoting:** $|l_{jk}| \le 1 \Rightarrow \|L\| = O(1)$ (e.g. $\|L\|_\infty \le n$), and $\|U\| \le \rho(A)\,\|A\|$-ish by the definition of the growth factor. Hence $\|\delta A\| \,/\, \|A\| = O(\rho\,\epsilon_{\text{mach}})$.

The whole question of Gaussian elimination's reliability is compressed into one scalar: *how big can $\rho(A)$ get?* Worst case 2^{n-1} (next lecture's explicit matrix), but essentially never in practice. Combined with the accuracy theorem:

$$\frac{\|\tilde{x} - x\|}{\|x\|} = O\big(\kappa(A)\,\rho(A)\,\epsilon_{\text{mach}}\big).$$

4.2 Stability of LU with Pivoting; Interpolation

We first settle the worst case of the growth factor: an explicit $n \times n$ matrix on which partial pivoting produces element growth 2^{n-1}, so LU with partial pivoting is not backward stable in the worst case. Yet decades of practice show such matrices essentially never arise, a tension we discuss candidly. We then open the interpolation unit: the polynomial interpolation problem, its solution by the Lagrange basis, and the reasons (oscillation and global coupling) why high-degree global polynomials are the wrong tool for large data sets, motivating splines.

4.2.1 Backward Stability of LU Factorization with Partial Pivoting (Continued)

This section exhibits Wilkinson's worst-case matrix, on which the growth factor attains its exponential bound, and discusses the gap between this worst case and sixty years of successful practice.

Example 4.5. ***Large*** $\rho(A)$
Does the growth factor ever actually reach its exponential bound? It does,

and the matrix that achieves it, due to Wilkinson, could hardly look more innocent: ones on the diagonal, -1 everywhere below it, ones in the last column, zeros elsewhere:

$$A = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & 1 \\ -1 & 1 & 0 & \cdots & 0 & 1 \\ -1 & -1 & 1 & \ddots & 0 & 1 \\ \vdots & \vdots & \ddots & \ddots & \vdots & \vdots \\ \vdots & \vdots & & \ddots & \ddots & \vdots \\ -1 & -1 & -1 & \cdots & -1 & 1 \end{bmatrix} \in \mathbb{R}^{n \times n}$$

Watch what elimination does to it. At every step the pivot candidate on the diagonal is a 1, and every entry below it is -1: tied in modulus, so partial pivoting exchanges nothing, and the pivoted algorithm behaves exactly like the unpivoted one. Eliminating the first column adds row 1 to every row below, which changes only the last column, doubling its lower entries from 1 to 2:

$$A \to \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & 1 \\ 0 & 1 & 0 & \cdots & 0 & 2 \\ 0 & -1 & 1 & \ddots & \vdots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 0 & 2 \\ 0 & -1 & -1 & \ddots & 1 & 2 \\ 0 & -1 & -1 & \cdots & -1 & 2 \end{bmatrix}$$

$$U = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & 1 \\ 0 & 1 & 0 & \cdots & 0 & 2 \\ 0 & 0 & 1 & \ddots & \vdots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 0 & 2^{n-3} \\ 0 & 0 & 0 & \ddots & 1 & 2^{n-2} \\ 0 & 0 & 0 & \cdots & 0 & 2^{n-1} \end{bmatrix} \qquad L = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & 0 \\ -1 & 1 & 0 & \cdots & 0 & 0 \\ -1 & -1 & 1 & \ddots & \vdots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 0 & 0 \\ -1 & -1 & -1 & \ddots & 1 & 0 \\ -1 & -1 & -1 & \cdots & -1 & 1 \end{bmatrix}$$

Each subsequent step repeats the pattern: the last-column entries below the current row double again, from 2 to 4 to 8, and after $n-1$ steps the corner entry has doubled $n-1$ times. Meanwhile L simply collects the -1 multipliers, all perfectly bounded, exactly as partial pivoting promises. The growth is entirely in U:

$$\rho(A) = \frac{\max_{j,k} |u_{jk}|}{\max_{j,k} |a_{jk}|} = \frac{2^{n-1}}{1} = 2^{n-1}.$$

To appreciate the scale: for $n = 64$ the growth factor is $2^{63} \approx 9 \times 10^{18}$, enough to wipe out all 16 digits of double precision, on a matrix that is itself well-conditioned. The instability, when it occurs, belongs entirely to the algorithm.

Example 4.6. ***Large*** $\rho(A)$ ***(Continued)***
What does the worst-case matrix mean for the theory? Taken over nonsingular matrices $A \in \mathbb{C}^{n\times n}$ of *all* sizes $n \geq 1$, LU factorization with partial pivoting is *not* backward stable. The reasoning is direct: backward stability requires $\|\delta A\| / \|A\| = O(\epsilon_{\text{mach}})$ with a constant independent of the input, but the theorem only delivers

$$\frac{\|\delta A\|}{\|A\|} = O(\rho(A)\, \epsilon_{\text{mach}}),$$

and Wilkinson's matrix shows $\rho(A) = 2^{n-1} \to \infty$ as $n \to \infty$: no uniform constant exists.

And yet, in more than sixty years of scientific computing, matrices with large $\rho(A)$ have essentially never arisen in applications: growth factors observed in practice are small, typically a modest power of n at worst. So we are left with an honest but unusual state of affairs: the workhorse algorithm of numerical linear algebra is provably unstable in the worst case and overwhelmingly reliable in every case that occurs. The remark below records the working conclusion that all of practical computation is built on.

Remark 4.7. ***Backward Stability of LU Factorization***
LU factorization with partial pivoting is backward stable in practice!

4.2.2 Interpolation

We change subjects, from solving equations to reconstructing functions from samples. Polynomial interpolation is the classical starting point. This section proves existence and uniqueness by exhibiting the Lagrange basis; Stoer and Bulirsch treat the subject in depth, including the Newton form and divided differences that we omit.
Problem: Given $n + 1$ data points

$$(x_j, y_j), \qquad j = 0, 1, \ldots, n \tag{9}$$

where $x_j, y_j \in \mathbb{R}$ and $x_0 < x_1 < x_2 < \cdots < x_n$.

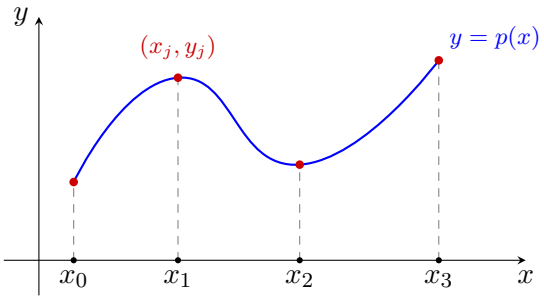


Figure 7: Data points (x_j, y_j) interpolated by a curve $y = p(x)$.

Find a "simple" function p that *interpolates* the data.

4.2.2.1 Polynomial Interpolation

Remark 4.8. ***Polynomial Notation***

$$P_n \in \prod_n := \left\{ P(x) = c_0 + c_1 x + \cdots + c_n x^n \;\middle|\; c_0, c_1, \ldots, c_n \in \mathbb{R} \right\}$$
$$= \text{ set of all (real) polynomials of degree } \leq n$$

Theorem 4.9.
There is a unique $P_n \in \prod_n$ that interpolates the data (9):

$$P_n(x) = \sum_{j=0}^{n} y_j \prod_{\substack{k=0 \\ k \neq j}}^{n} \frac{x - x_k}{x_j - x_k}$$

(*Lagrange interpolation formula*)

Proof.

$$P_n(x_l) = \sum_{j=0}^{n} y_j \prod_{\substack{k=0 \\ k \neq j}}^{n} \frac{x_l - x_k}{x_j - x_k}$$

$$\prod_{\substack{k=0 \\ k \neq j}}^{n} \frac{x_l - x_k}{x_j - x_k} = \begin{cases} 0 & l \neq j \\ 1 & l = j \end{cases}$$

$$P_n(x_l) = y_l \qquad \forall\, l$$

P_n is unique: Let $Q \in \prod_n$ be another interpolating polynomial. Then

$$D(x) = P_n(x) - Q(x) \in \prod_n$$

$$D(x_j) = P_n(x_j) - Q(x_j) = y_j - y_j = 0, \qquad j = 0, 1, \ldots, n$$

Thus, D has at least $n+1$ zeros. Since D is a polynomial of degree n, this implies that $D = 0 \;\Rightarrow\; P_n = Q$. □

But...
Polynomials are usually not *flexible* enough.

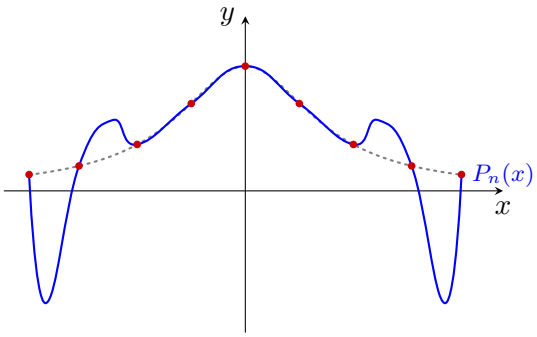


Figure 8: A high-degree interpolating polynomial oscillating wildly between equally spaced data points.

Remedy: piecewise polynomials ⇒ *splines*

4.2.2.2 Splines

Given: data points (x_0, y_0), (x_1, y_1), (x_2, y_2), ..., (x_n, y_n), where $a := x_0 < x_1 < x_2 < \cdots < x_n =: b$ and $n \geq 1$.

Definition 4.10. ***Spline***
A function $S : [a, b] \to \mathbb{R}$ is called a *spline* of degree $k-1$ (order k) if

(a) $S \in C^{k-2}[a, b]$

(b) On each interval $[x_{j-1}, x_j]$, $j = 1, 2, \ldots, n$, $S(x) \in \prod_{k-1}$

4.2.3 Filling in the Details: Watching the Doubling Happen

In the worst-case matrix, no rows are ever exchanged (each pivot column already has its maximum, 1, on the diagonal: all entries below are -1, tied

in modulus, and ties keep the current row). Eliminating column 1 adds row 1 to each lower row; only the *last column* changes:

$$\text{last column: } (1,1,\ldots,1)^T \to (1,2,2,\ldots,2)^T.$$

Step 2 adds the new row 2 to rows $3,\ldots,n$: last column entries below become 4. Each step doubles them:

$$u_{nn} = 2^{n-1}, \qquad \rho(A) = \frac{\max|u_{jk}|}{\max|a_{jk}|} = \frac{2^{n-1}}{1}.$$

With $n = 64$, $\rho = 2^{63} \approx 9 \times 10^{18}$: elimination would lose *all* 16 digits. Yet the matrix is perfectly well-conditioned! The instability is genuinely the algorithm's.

Why doesn't this bite in practice? Random and application matrices exhibit ρ growing like a small power of n at most; the exponential examples are measure-zero curiosities. Statistical explanations exist (average-case analysis), but the honest summary is empirical: 60+ years of scientific computing rest on "ρ is small in practice."

Uniqueness in the Lagrange theorem (the key counting argument): a nonzero polynomial of degree $\leq n$ has at most n roots. The difference $D = P_n - Q$ of two interpolants has degree $\leq n$ but $n+1$ roots (the nodes), forcing $D \equiv 0$. This "dimension vs constraints" argument returns for splines (Chapter 5) and quadrature (Chapter 6).

Lagrange vs Vandermonde: solving for coefficients in the monomial basis means a Vandermonde system, which is notoriously ill-conditioned. The Lagrange basis $\ell_j(x) = \prod_{k\neq j} \frac{x-x_k}{x_j-x_k}$ satisfies $\ell_j(x_l) = \delta_{jl}$, so the "system" is the identity: the right basis eliminates the computation. Choosing good bases is a running theme (B-splines, orthogonal polynomials, ...).

4.2.4 Why Not Just Use One Big Polynomial?

Two distinct problems with degree-n interpolation for large n:

- **Oscillation (Runge phenomenon):** for equally spaced nodes the interpolant of even a smooth bell-shaped function develops wild boundary oscillations, and the error *diverges* as $n \to \infty$.
- **Global coupling:** moving one data point changes the polynomial everywhere on $[a,b]$.

Splines fix both: low, fixed degree on each subinterval (no room to oscillate), glued with just enough smoothness (C^{k-2}), and, once we have the B-spline basis, *local* dependence on the data. The price is bookkeeping: counting how many free parameters survive the gluing. That count is exactly where next lecture begins.

4.3 Notes and References

The worst-case growth factor example is due to Wilkinson, whose backward error analysis of Gaussian elimination founded the subject; Higham surveys the modern understanding, including the average-case behavior that explains why partial pivoting succeeds in practice. Trefethen and Bau discuss the same tension. The perturbation theory of linear systems is classical and appears in all of the references.

4.4 Exercises

1. Run Gaussian elimination with partial pivoting by hand on the 4×4 worst-case matrix of the text (diagonal 1, subdiagonal entries -1, last column 1). Verify that no rows are exchanged and that the final entry is $u_{44} = 8$.

2. Prove that with partial pivoting the entries of the intermediate matrices satisfy $\max_{j,k} \left|u_{jk}^{(i+1)}\right| \le 2 \max_{j,k} \left|u_{jk}^{(i)}\right|$ at each elimination step, and conclude the bound $\rho(A) \le 2^{n-1}$.

3. Show that $\kappa(A) = \|A\| \left\|A^{-1}\right\| \ge 1$ for every induced matrix norm.

4. Derive the perturbation bound for the right-hand side: if $Ax = b$ and $A(x + \delta x) = b + \delta b$, then $\dfrac{\|\delta x\|}{\|x\|} \le \kappa(A) \dfrac{\|\delta b\|}{\|b\|}$.

5. (Computing.) The Hilbert matrix H_n has entries $1/(i + j - 1)$. For $n = 5, 8, 11, 14$, set $x = (1, \ldots, 1)^T$, form $b = H_n x$, solve the system with backslash, and tabulate the relative error against $\kappa_2(H_n)\, \epsilon_{\text{mach}}$. Do the results match the accuracy theorem?

6. (Computing.) Generate random 100×100 matrices with standard normal entries and compute the growth factor $\rho(A)$ of partial-pivoting elimination for a few hundred samples. How large are typical values compared with the worst-case bound 2^{99}?

Chapter 5

Interpolation and Splines

This unit treats the reconstruction of functions from data. After polynomial interpolation and its limitations, we develop spline interpolation in depth: existence and uniqueness, the minimum-curvature optimality of cubic splines, efficient construction via tridiagonal systems, and the B-spline basis that makes spline computation local, stable, and fast.

5.1 Splines

This lecture introduces spline interpolation. After defining splines of order k (piecewise polynomials of degree $k-1$ glued together with C^{k-2} smoothness), we carry out the degrees-of-freedom count showing that interpolation determines a spline up to exactly $k-2$ remaining conditions. For cubic splines this leaves two conditions, and we catalog the four classical ways to spend them: natural, not-a-knot, periodic, and clamped boundary conditions. We close with the theorem that gives cubic splines their name and their reputation: among all C^2 interpolants, they minimize the curvature energy $\int (f'')^2$.

$$a = x_0 < x_1 < x_2 < \cdots < x_n = b$$
$$S \in C^{k-2}[a,b]$$
$$S(x) \in \prod_{k-1}, \qquad x_{j-1} \le x_j$$

$k \ge 2$, k order, $k-1$ degree

5.1.1 Working with Splines

Piecewise polynomials avoid the failures of high-degree interpolation. This section defines spline spaces, counts their degrees of freedom, and identifies the boundary conditions under which cubic spline interpolation is uniquely solvable. De Boor's book is the standard reference for the entire chapter.

Example 5.1. ***Linear & Cubic Splines***
$k = 2 \Rightarrow$ **linear spline**
$k = 4, \Rightarrow$ **cubic spline**

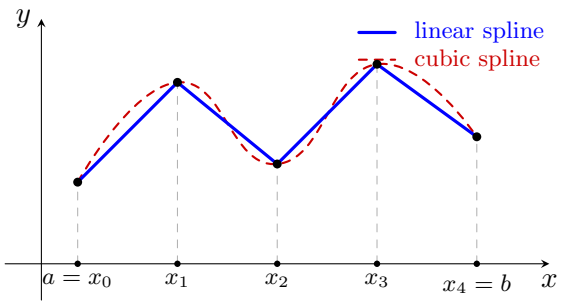


Figure 9: Linear spline (solid) and cubic spline (dashed).

Remark 5.2. ***Counting Degrees of Freedom***
On each $[x_{j-1}, x_j]$: $S = S_j \in \prod_{k-1} \Rightarrow k$ coefficients.
$\Rightarrow$ The total number of parameters $= nk$ coefficients.

$S \in C^{k-2}$. For each x_j, $j = 1, 2, \ldots, n-1$, $S_j^{(i)}(x_j) = S^{(i)}(x_j - 0) = S^{(i)}(x_j + 0) = S_{j+1}^{(i)}(x_j)$, $i = 0, 1, \ldots, k-2$. $\Rightarrow$ This is a total of $(n-1)(k-1)$ conditions.
Use a spline of degree $k-1$ to interpolate the data (x_0, y_0), (x_1, y_1), $\ldots$, (x_n, y_n):

$$S(x_j) = y_j, \qquad j = 0, 1, \ldots, n$$

$\Rightarrow$ Another $n+1$ conditions.

$$\begin{aligned} \#\text{ coefficients} - \#\text{ conditions} &= nk - (n-1)(k-1) - (n-1) \\ &= k - 2 \end{aligned}$$

Thus, we need to impose $k-2$ more conditions in order to get a unique spline.

Example 5.3.
For $k = 2$, there is a unique linear spline that interpolates the data.
For $k = 4$, we need 2 additional conditions to have a unique cubic spline that interpolates the data.

5.1.1.1 Cubic Splines

$k = 4$, $S \in C^2[a, b]$,

$$S(x) = S_j(x) \in \prod_3 \quad \text{on } [x_{j-1}, x_j],\ j = 1, 2, \ldots, n.$$

Remark 5.4. ***Additional 2 Conditions*** 1. Natural spline: $S''(a) = S_1''(a) = 0, \quad S''(b) = S_n''(b) = 0.$

2. Not-a-knot condition: we need that $n \geq 3$. $S_1'''(x_1) = S_2'''(x_1)$, $S_{n-1}'''(x_{n-1}) = S_n'''(x_{n-1})$.

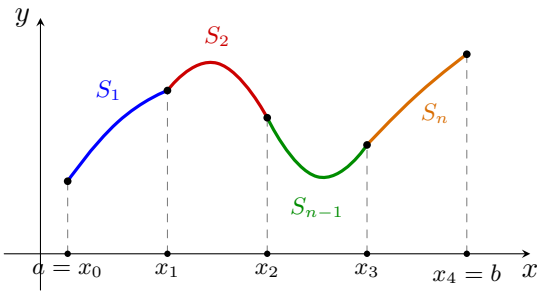


Figure 10: Spline pieces $S_1, \ldots, S_n$ joined at the knots $a = x_0 < x_1 < \cdots < x_4 = b$.

Recall: $S_1, S_2 \in \prod_3, \quad S_1, S_2 \in C^2, \quad S_1^{(i)}(x_1) = S_2^i(x_2)$ for $i = 0, 1, 2, 3$. Thus, $S_1 = S_2$ for all $x_0 \leq x \leq x_2$, i.e. x_1 is not really a knot. Similarly, x_{n-1} is not really a knot.
$\Rightarrow$ This condition is the default for Matlab's `spline` function.

3. Periodic spline: Assume that the data is periodic, i.e. $y_0 = y_n$ $(S_1(a) = S_n(b))$.

$$S_1'(a) = S_n'(b)$$
$$S_1''(a) = S_n''(b)$$

4. Clamped spline: $S_1'(a) = v_0, \quad S_n'(b) = v_n$, where $v_0, v_n \in \mathbb{R}$ are given.

Theorem 5.5.
For each of the 4 cases in Remark 17.4, there exists a unique cubic spline that interpolates the data.

Remark 5.6.
Cubic splines are the "smoothest" interpolants among all interpolating functions $f \in C^2[a, b]$.

$$\|f\|_2 = \left(\int_a^b |f(x)|^2 \, dx \right)^{1/2}$$

Theorem 5.7.
Let S be a cubic spline that interpolates the data. Let $f \in C^2[a,b]$ be any function that also interpolates the data: $f(x_j) = y_j$, $j = 0, 1, \ldots, n$. Assume that

$$S''(x)\big[f'(x) - S'(x)\big]\Big|_{x=a}^{x=b} = 0.$$

Then $\|S''\|_2 \leq \|f''\|_2$.

Remark 5.8. ***Comment on Theorem 17.7***
Theorem 17.7 does not apply for the not-a-knot condition because the assumption does not hold, but it does apply for the other 3.

5.1.2 Filling in the Details: The Bookkeeping Behind $k-2$

Count carefully for order k on n subintervals:

free parameters: n pieces $\times$ k coefficients	nk
smoothness: C^{k-2} at $n-1$ interior knots, i.e. matching	
$S^{(i)}$, $i = 0, \ldots, k-2$ $\quad$ ($k-1$ equations per knot)	$(n-1)(k-1)$
interpolation at all $n+1$ nodes	$n+1$

$$nk - (n-1)(k-1) - (n+1) = nk - (nk - n - k + 1) - n - 1 = k - 2.$$

Sanity checks: $k = 2$ (linear): 0 extra conditions: connect the dots, uniquely. $k = 4$ (cubic): 2 extra conditions, hence the four standard recipes for spending them.

Choosing among the four cubic closures:

- *Natural* ($S'' = 0$ at both ends): simplest analysis, physical "free beam" ends; but it forces zero curvature the data may not have, costing accuracy near the boundary.
- *Not-a-knot*: uses no extra information; first two (and last two) pieces merge into single cubics. MATLAB's default. Best generic accuracy.
- *Clamped*: uses true endpoint slopes v_0, v_n if you know them; then it is the most accurate.
- *Periodic*: for closed curves / periodic data.

The name "spline": a draftsman's flexible ruler pinned at the data points. An elastic rod minimizes bending energy $\int (S'')^2$, precisely the quantity in the optimality theorem. The mathematics reproduces the physical instrument.

5.1.3 Reading the Optimality Theorem Correctly

Statement shape: among *all* C^2 interpolants f of the same data (subject to the boundary identity), the cubic spline minimizes $\|f''\|_2$.

- The competitor set is huge (any smooth curve through the points), yet a piecewise cubic wins.
- "Energy" $\|f''\|_2^2 = \int_a^b (f'')^2$ penalizes curvature: the spline is the *flattest* possible interpolant, which is why spline interpolants do not oscillate.
- The boundary hypothesis $S''(f' - S')\big|_a^b = 0$ holds for natural ($S'' = 0$ at ends), clamped ($f' = S'$ at ends), and periodic (terms cancel), but *not* for not-a-knot. One theorem, three closures.

Proof next lecture: one integration by parts plus a Pythagoras-style decomposition, worth following in detail, as the same "orthogonality" argument underlies least squares and Galerkin methods.

5.2 Interpolation Error; Cubic Spline Construction

Two constructions occupy this lecture. First we prove the minimum-curvature optimality theorem, a one-page argument whose two pillars reappear throughout applied mathematics: an integration by parts made possible by the piecewise-constant third derivative, and a Pythagoras-style orthogonality decomposition. Then we construct the natural cubic spline interpolant explicitly: eliminating parameters reduces the problem to a symmetric tridiagonal linear system for the knot curvatures, which is strictly diagonally dominant, hence uniquely solvable, and solvable in $O(n)$ operations without pivoting.

5.2.1 Proof of the Optimality Theorem

The optimality proof is short and instructive. Its two steps, an integration by parts and an orthogonality argument, recur in least-squares theory and in the analysis of Galerkin methods.

Proof.

$$\begin{aligned}
\int_a^b S''(x)\big[f''(x) - S''(x)\big]\,dx &\overset{\text{IBP}}{=} \underbrace{S''(f' - S')\Big|_a^b}_{=0} - \int_a^b S'''(x)\big[f'(x) - S'(x)\big]\,dx \\
&= -\sum_{j=1}^{n} c_j \int_{x_{j-1}}^{x_j} f'(x) - S'(x)\,dx \\
&\qquad \big(S'''(x) = c_j = \text{constant on } [x_{j-1}, x_j]\big) \\
&= -\sum_{j=1}^{n} c_j \big[f(x) - S(x)\big]\Big|_{x_{j-1}}^{x_j} \\
&= -\sum_{j=1}^{n} c_j \big(y_j - y_j - \cancel{y_{j-1}} + \cancel{y_{j-1}}\big) \\
&= 0
\end{aligned}$$

Next,

$$f''(x) = S''(x) + \big[f''(x) - S''(x)\big]$$

$$[f''(x)]^2 = [S''(x)]^2 + 2S''(x)\big[f''(x) - S''(x)\big] + \big[f''(x) - S''(x)\big]^2$$

$$\big\|f''\big\|_2^2 = \underbrace{\int_a^b [S''(x)]^2\,dx}_{=\|S''\|^2} + 2\underbrace{\int_a^b S''(x)\big[f''(x) - S''(x)\big]\,dx}_{=0} + \underbrace{\int_a^b \big[f''(x) - S''(x)\big]^2\,dx}_{\geq 0}$$

$$\big\|f''\big\| \geq \big\|S''\big\|_2$$

□

5.2.2 Construction of an Interpolating Cubic Spline

Optimality does not construct the spline; this section does, by direct elimination. We parameterize each piece, impose the conditions in a productive order, and watch the problem collapse to a small tridiagonal system. The computation is longer than anything so far, but every step is elementary.

Constructing splines boils down to solving a tridiagonal linear system ($\Rightarrow$ you never have to pivot). We will only cover the natural cubic spline; the other endpoint conditions are similar.

Recall:

$$a = x_0 < x_1 < x_2 < \cdots < x_n = b$$
$$S(x_j) = y_j, \qquad j = 0, 1, \ldots, n$$
$$S(x) = S_j(x) = \alpha_j + \beta_j(x - x_{j-1}) + \gamma_j(x - x_{j-1})^2 + \delta_j(x - x_{j-1})^3$$
$$\text{on } [x_{j-1}, x_j], \quad j = 1, 2, \ldots, n$$

Conditions:

1. $S_j(x_{j-1}) = y_{j-1}, \quad S_j(x_j) = y_j \quad \text{for } j = 1, 2, \ldots, n$
2. $S'_j(x_j) = S'_{j+1}(x_j), \quad j = 1, 2, \ldots, n-1$
3. $S''_j(x_j) = S''_{j+1}(x_j), \quad j = 1, 2, \ldots, n-1$
4. $S''_1(x_0) = 0, \quad S''_n(x_n) = 0 \quad$ (natural spline condition)

From these conditions, we get:

1. $\alpha_j = y_{j-1}$ and

$$y_j - y_{j-1} = \beta_j(x_j - x_{j-1}) + \gamma_j(x_j - x_{j-1})^2 + \delta_j(x_j - x_{j-1})^3$$
$$\Delta_j := y_j - y_{j-1}, \quad \xi_j := x_j - x_{j-1} \quad \text{(Notation)}$$

2. $\beta_j + 2\gamma_j\xi_j + 3\delta_j\xi_j^2 = \beta_{j+1}, \quad j = 1, 2, \ldots, n-1$
3. $2\gamma_j + 6\delta_j\xi_j = 2\gamma_{j+1}, \quad j = 1, 2, \ldots, n$
4. $2\gamma_1 = 0, \quad 2\gamma_n + 6\delta_n\xi_n = 0 \;\Leftrightarrow\; \gamma_1 = 0, \quad \gamma_{n+1} = 0 \quad$ where $\gamma_{n+1} = \gamma_n + 3\delta_n\xi_n$

3.
$$\delta_j = \frac{\gamma_{j+1} - \gamma_j}{3\xi_j}$$

1.
$$\beta_j = \frac{\Delta_j}{\xi_j} - \gamma_j\xi_j - \frac{\xi_j}{3}(\gamma_{j+1} - \gamma_j) = \frac{\Delta_j}{\xi_j} - \frac{\xi_j}{3}(\gamma_{j+1} + 2\gamma_j)$$

Insert the newest (3) and (1) into (2):

$$\xi_j\gamma_j + 2(\xi_j + \xi_{j+1})\gamma_{j+1} + \xi_{j+2}\gamma_{j+2} = 3\left(\frac{\Delta_{j+1}}{\xi_{j+1}} - \frac{\Delta_j}{\xi_j}\right), \qquad j = 1, 2, \ldots, n-1$$

(where $\gamma_1 = 0$, $\gamma_{n+1} = 0$).

$$\begin{bmatrix} 2(\xi_1+\xi_2) & \xi_2 & & & & \\ \xi_2 & 2(\xi_2+\xi_3) & \xi_3 & & & \\ & \ddots & \ddots & \ddots & & \\ & & \xi_{n-2} & 2(\xi_{n-2}+\xi_{n-1}) & \xi_{n-1} & \\ & & & \xi_{n-1} & 2(\xi_{n-1}+\xi_n) \end{bmatrix} \begin{bmatrix} \gamma_2 \\ \gamma_3 \\ \vdots \\ \gamma_n \end{bmatrix} = \begin{bmatrix} t_1 \\ t_2 \\ \vdots \\ t_{n-1} \end{bmatrix}$$

where

$$t_j := 3\left(\frac{\Delta_{j+1}}{\xi_{j+1}} - \frac{\Delta_j}{\xi_j}\right), \qquad j = 1, 2, \ldots, n-1.$$

Remark 5.9.
The coefficient matrix of this system is tridiagonal, symmetric, and strictly diagonally dominant:

$$2(\xi_j + \xi_{j+1}) > \xi_j + \xi_{j+1} \quad \text{for all } j.$$

$\Rightarrow$ There exists a unique solution $\gamma_2, \gamma_3, \ldots, \gamma_n$.

Furthermore, no pivoting is needed. The system can be solved in $O(n)$ flops.

5.2.3 Filling in the Details: The Two Pillars of the Proof

Pillar 1: the orthogonality relation. Why does $\int_a^b S''(f'' - S'')\,dx = 0$? Integrate by parts once ($u = S''$, $dv = (f'' - S'')dx$):

$$\int_a^b S''(f'' - S'') = \underbrace{S''(f' - S')\Big|_a^b}_{=0 \text{ (hypothesis)}} - \int_a^b S'''(f' - S').$$

Now S is piecewise cubic, so $S''' \equiv c_j$ (a constant!) on each $[x_{j-1}, x_j]$; integrate piece by piece:

$$\int_a^b S'''(f' - S') = \sum_j c_j \big[f - S\big]_{x_{j-1}}^{x_j} = 0,$$

because f and S *interpolate the same values* at every knot: $f(x_j) - S(x_j) = y_j - y_j = 0$. Every hypothesis got used exactly once.

Pillar 2: Pythagoras. With $g := f'' - S''$,

$$\big\|f''\big\|_2^2 = \big\|S''\big\|_2^2 + 2\underbrace{\langle S'', g\rangle}_{=0} + \|g\|_2^2 \geq \big\|S''\big\|_2^2.$$

S'' and the residual are orthogonal in L^2: the spline is the *orthogonal projection* flavor of optimality. Equality iff $f'' = S''$, i.e. $f = S$ (they share values and, integrating twice, the constants match).

Tracing the elimination to the tridiagonal system. Strategy of the derivation, so the algebra does not obscure it:

1. Parameters: $\alpha_j, \beta_j, \gamma_j, \delta_j$ per piece ($4n$ unknowns), with $\xi_j = x_j - x_{j-1}$, $\Delta_j = y_j - y_{j-1}$.
2. Interpolation at left ends $\Rightarrow \alpha_j = y_{j-1}$: eliminates n unknowns for free.
3. C^2 continuity (3.) $\Rightarrow \delta_j = \frac{\gamma_{j+1} - \gamma_j}{3\xi_j}$: expresses all δ's through γ's.
4. Interpolation at right ends (1.) $\Rightarrow$ solve for β_j in terms of γ_j, γ_{j+1}.
5. Only C^1 continuity (2.) remains: substituting 3.–4. into it yields, for $j = 1, \ldots, n-1$,

$$\xi_j \gamma_j + 2(\xi_j + \xi_{j+1})\gamma_{j+1} + \xi_{j+1}\gamma_{j+2} = 3\Big(\tfrac{\Delta_{j+1}}{\xi_{j+1}} - \tfrac{\Delta_j}{\xi_j}\Big),$$

with $\gamma_1 = \gamma_{n+1} = 0$ (natural ends): an $(n-1) \times (n-1)$ *tridiagonal* system for the interior γ's.

Note $\gamma_j = \frac{1}{2} S''(x_{j-1})$: we are really solving for the curvatures at the knots.

5.2.4 Why the System Is So Nice

Diagonal dominance, quantitatively: in every row,

$$|2(\xi_j + \xi_{j+1})| = 2(\xi_j + \xi_{j+1}) > \xi_j + \xi_{j+1} = \text{(sum of off-diagonal moduli)},$$

strictly, since all $\xi_j > 0$. Consequences:

- **Nonsingularity:** if $M\gamma = 0$, look at the index i where $|\gamma_i|$ is maximal; the ith equation gives $2(\xi_i + \xi_{i+1})\,|\gamma_i| \le (\xi_i + \xi_{i+1})\,|\gamma_i|$, forcing $\gamma = 0$. Unique spline!
- **No pivoting needed:** diagonal dominance is preserved during elimination, so pivots never vanish and element growth is controlled ($\rho \le 2$ for such systems).
- $O(n)$ **solve:** tridiagonal LU (the "Thomas algorithm") costs $\approx 8n$ flops (forward sweep plus back substitution). A million-point spline is instantaneous.

The same matrix structure (symmetric, tridiagonal, diagonally dominant) reappears in finite differences for ODE boundary value problems.

5.3 B-Splines

To compute with splines efficiently we need a good basis for the spline space $\mathcal{S}_{k,\Delta}$. This lecture determines the dimension of that space, $k+n-1$, and exhibits a first basis: monomials together with the truncated power functions $(x-x_j)_+^{k-1}$, whose $(k-1)$st derivative jumps at exactly one knot. The jump functionals make the linear-independence proof clean and instructive. Numerically, however, this basis fails on two counts, severe ill-conditioning and global support, and its failure sets the specification for the basis we actually want, the B-splines.

5.3.1 B-splines

For computation, the description of a spline matters as much as its existence. This section determines the dimension of the spline space and exhibits a first, mathematically convenient basis. Its numerical failure, documented at the end, motivates the B-splines that occupy the rest of the chapter.
Given: $\Delta = \{x_0, x_1, \ldots, x_n\}$, where $a = x_0 < x_1 < \cdots < x_n = b$ and $n \geq 1$ (so that $a < b$). Recall that for the spline S of degree $k-1$ (order k): $S \in C^{k-2}[a,b]$ and $S \in \prod_{k-1}$ on $[x_{j-1}, x_j]$, $j = 1, 2, \ldots, n$.

Definition 5.10. $\mathcal{S}_{k,\Delta}$
$\mathcal{S}_{k,\Delta}$ = the set of all splines of order k, $k \geq 2$.

It is easy to see that $\mathcal{S}_{k,\Delta}$ is a real vector space:

$$S, \tilde{S} \in \mathcal{S}_{k,\Delta},\ \alpha \in \mathbb{R} \ \Rightarrow\ S + \tilde{S} \in \mathcal{S}_{k,\Delta},\ \alpha S \in \mathcal{S}_{k,\Delta}.$$

Also, $\prod_{k-1} \subset \mathcal{S}_{k,\Delta}$.

Proposition 5.11. ***Dimension of*** $\mathcal{S}_{k,\Delta}$

- On $[x_0, x_1]$: k coefficients
- On $[x_1, x_2]$: k coefficients and $k-1$ conditions $\Rightarrow$ i.e. only 1 coefficient
- Similarly for $[x_{j-1}, x_j]$, $j = 1, 2, 3, \ldots, n$.

So $\dim(\mathcal{S}_{k,\Delta}) = k + n - 1$.

5.3.1.1 B-spline Basis (1)

We consider this kind of basis composed of monomials, that is, $p(x) = x^l$, $l = 0, 1, 2, \ldots$, and truncated powers of degree $k-1$:

$$f_{x_j}(x) = (x - x_{j-1})_+^{k-1} := \big(\max\{x - x_j, 0\}\big)^{k-1} = \begin{cases} (x-x_j)^{k-1} & x \geq x_j \\ 0 & x < x_j \end{cases}$$

$$\frac{d^l}{dx^l}(x-x_j)_+^{k-1} = (k-1)(k-2)\dots(k-1-l)(x-x_j)_+^{k-1-l}$$

So $(x-x_j)_+^{k-1} \in C^{k-2}[a,b]$, $(x-x_j)_+^{k-1} \in \mathcal{S}_{k,\Delta}$, and

$$\frac{d^{k-1}}{dx^{k-1}}(x-x_j)_+^{k-1} = \begin{cases} 0 & x < x_j \\ (k-1)! & x > x_j \end{cases}$$

Theorem 5.12.
The set

$$B = \{1, x, x^2, \dots, x^{k-1}, (x-x_1)_+^{k-1}, (x-x_2)_+^{k-1}, \dots, (x-x_{n-1})_+^{k-1}\}$$

is a basis for $\mathcal{S}_{k,\Delta}$.

Proof. We only need to show that the $k+n-1$ functions in B are linearly independent. Let

$$S(x) = \sum_{l=0}^{k-1} \alpha_l x^l + \sum_{i=1}^{n-1} \beta_i (x-x_i)_+^{k-1} = 0$$

for all $a \le x \le b$. For each $j = 1, 2, \dots, n-1$,

$$\begin{aligned} 0 &= S^{(k-1)}(x_j+0) - S^{(k-1)}(x_j-0) \\ &= \beta_j \left(\frac{d^{k-1}}{dx^{k-1}}(x-x_j)_+^{k-1}\Big|_{x=x_j+0} - \frac{d^{k-1}}{dx^{k-1}}(x-x_j)_+^{k-1}\Big|_{x=x_j-0} \right) \\ &= \beta_j (k-1)!. \end{aligned}$$

Thus, $\beta_j = 0$ for $j = 1, 2, \dots, n-1$, and

$$S(x) = \sum_{l=0}^{k-1} \alpha_l x^l = 0$$

for all $a \le x \le b$. So $\alpha_l = 0$ for $l = 0, 1, \dots, k-1$, and thus the functions in B are linearly independent. □

However, the basis B is useless in practice:

- Ill-conditioned
- Global support. $\operatorname{supp} x^l = [a,b]$, $\operatorname{supp}(x-x_j)_+^{n-1} = [x_j, b]$. It is more efficient to have a basis function with local support only.

5.3.2 Filling in the Details: Smoothness of Truncated Powers

Fix the knot x_j and let $\phi(x) = (x - x_j)_+^{k-1}$. Away from x_j, ϕ is a polynomial (either 0 or $(x - x_j)^{k-1}$). At x_j, differentiate each side l times:

$$\phi^{(l)}(x) = (k-1)(k-2)\cdots(k-l)\,(x - x_j)_+^{k-1-l},$$

For $l \le k-2$ the exponent $k-1-l \ge 1$ is positive, so both one-sided limits at x_j are 0: the derivative matches, $\phi \in C^{k-2}$. At $l = k-1$ the exponent hits 0 and

$$\phi^{(k-1)}(x_j^-) = 0, \qquad \phi^{(k-1)}(x_j^+) = (k-1)! \ :$$

a *jump of exactly* $(k-1)!$. So ϕ has precisely the spline smoothness: as much as possible without being a global polynomial.

That jump powers the basis proof. In the linear-independence argument, applying the jump functional $J_j[\,g\,] := g^{(k-1)}(x_j^+) - g^{(k-1)}(x_j^-)$ to $S = \sum_l \alpha_l x^l + \sum_i \beta_i (x - x_i)_+^{k-1} = 0$ kills every term except one: monomials are C^∞ (no jump), and $(x - x_i)_+^{k-1}$ with $i \neq j$ is smooth at x_j. What survives is $\beta_j (k-1)! = 0$. The functionals $J_1, \ldots, J_{n-1}$ pick off the β's one at a time; then the monomials' independence finishes it. **Pattern to remember:** to prove independence, find functionals that annihilate all basis elements but one.

Dimension check: k monomials $+(n-1)$ truncated powers $= k + n - 1$ $= \dim \mathcal{S}_{k,\Delta}$, matching the coefficient count (k on the first interval, one new freedom per additional knot).

5.3.3 Why This Basis Fails in Practice

- **Ill-conditioning.** For close knots $x_j \approx x_{j+1}$, the functions $(x - x_j)_+^{k-1}$ and $(x - x_{j+1})_+^{k-1}$ are nearly identical over most of $[a, b]$: columns of the collocation matrix become nearly parallel, κ explodes. (Compare: $\{1, x, x^2, \ldots\}$ on $[0, 1]$, all monotone increasing and similar-looking; the same disease.)
- **Global support.** $\operatorname{supp} x^l = [a, b]$ and $\operatorname{supp}(x - x_j)_+^{k-1} = [x_j, b]$: every evaluation of $S(x)$ touches $O(n)$ terms, every data change touches most coefficients, and interpolation matrices are dense (triangular-ish, but full).

Wish list for a good spline basis: nonnegative, supported on only k consecutive intervals, summing to 1, well-conditioned. All four wishes are granted by the *B-splines*, constructed recursively next lecture.

5.4 B-Splines of Order k

This lecture constructs the B-spline basis. After handling the interval endpoints by repeating the boundary knots k times, we define the B-splines N_{ik} by the Cox–de Boor recursion and run the recursion by hand from characteristic functions ($k = 1$) through hat functions ($k = 2$) to smooth bumps. We then establish the five structural properties (local support, nonnegativity, partition of unity, C^{k-2} smoothness, and piecewise degree $k - 1$) that make B-splines the universal representation of splines in computation.

5.4.1 B-Splines

This section sets up the knot sequence with repeated endpoints, the technical device that makes the B-spline family behave correctly at the boundary of the interval.

B-splines are basis functions for $\mathcal{S}_{k,\Delta}$ with desirable properties.

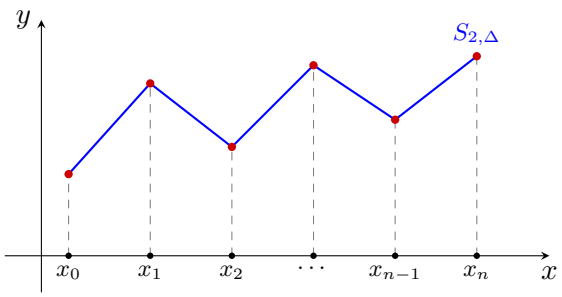


Figure 11: A piecewise linear spline $S_{2,\Delta}$ through the data points.

Example 5.13. $k = 2$ ***(Linear Splines)***
B-splines for $\mathcal{S}_{2,\Delta}$ are the *hat functions*.

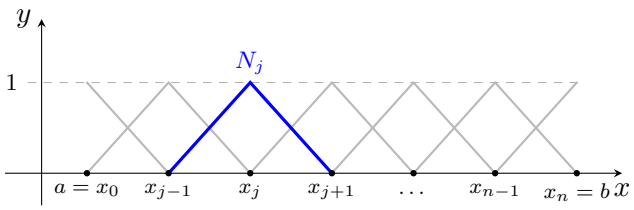


Figure 12: Hat functions on the knots; N_j is supported on $[x_{j-1}, x_{j+1}]$.

$$S(x) = \sum_{i=1}^{n+1} y_{i-1} N_{i2}(x)$$

$$S(x_{i-1}) = y_{i-1}, \qquad i = 1, 2, \ldots, n+1$$

Note: the end points x_0 and x_n are special: for $k = 2$, we count x_0 and x_n "twice."

For general $k \geq 2$:

$$\overbrace{\underbrace{x_0}_{\tau_1=\tau_2=\cdots=\tau_k}}^{k \text{ times}} < \underbrace{x_1}_{\tau_{k+1}} < \underbrace{x_2}_{\tau_{k+2}} < \cdots < \underbrace{x_{n-1}}_{\tau_{k+n-1}} < \overbrace{\underbrace{x_n}_{\tau_{k+n}=\tau_{k+n+1}=\cdots=\tau_{2k+n-1}}}^{k \text{ times}}$$

5.4.2 Construction of B-Splines of Order k (≥ 2)

The B-splines are defined by the Cox–de Boor recursion, one of the most useful recursions of approximation theory. This section states it and begins establishing the basic properties; de Boor's book develops the full theory, including derivative and integral formulas that we do not need.

$$\tau_1 \leq \tau_2 \leq \cdots \leq \tau_l, \qquad \tau_1, \tau_2, \ldots, \tau_l \in \mathbb{R} \tag{10}$$

$$\begin{aligned} l &= 2k + n - 1 \\ l - k &= k + n - 1 \\ &= \dim \mathcal{S}_{k,\Delta} \end{aligned}$$

Definition 5.14.
The B-splines N_{ik} of order k (≥ 2) (associated with (10)), $i = 1, 2, \ldots, l-k$, are defined recursively as follows:

$$N_{i1}(x) := \begin{cases} 1 & \tau_i \leq x < \tau_{i+1} \\ 0 & \text{otherwise} \end{cases}, \qquad i = 1, 2, \ldots, l-1$$

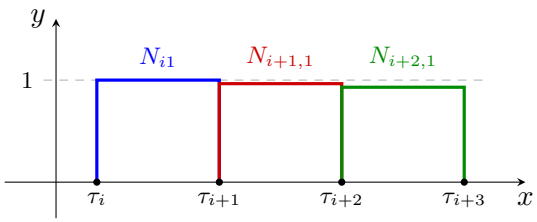


Figure 13: The piecewise constant B-splines N_{i1}, $N_{i+1,1}$, $N_{i+2,1}$ on the knots $\tau_i, \ldots, \tau_{i+3}$.

For $j = 2, 3, \ldots, k$:

$$N_{ij}(x) := \frac{x - \tau_i}{\tau_{i+j-1} - \tau_i} N_{i,j-1}(x) + \frac{\tau_{i+j} - x}{\tau_{i+j} - \tau_{i+1}} N_{i+1,j-1}(x), \qquad i = 1, 2, \ldots, l-j \tag{11}$$

Conventions:

1. $\frac{0}{0} = 0$ if $\tau_{i+j} - \tau_i = 0$ or $\tau_{i+j} - \tau_{i+1} = 0$. For example, if $\tau_i = \tau_{i+1} = \cdots = \tau_{i+k}$.

$$N_{i,j-1}(x) \equiv 0$$
$$N_{i+1,j-1}(x) \equiv 0$$
$$N_{ij}(x) \equiv 0$$

2. $N_{ik}(\tau_l) := \lim_{x \to \tau_l^-} N_{ik}(x)$

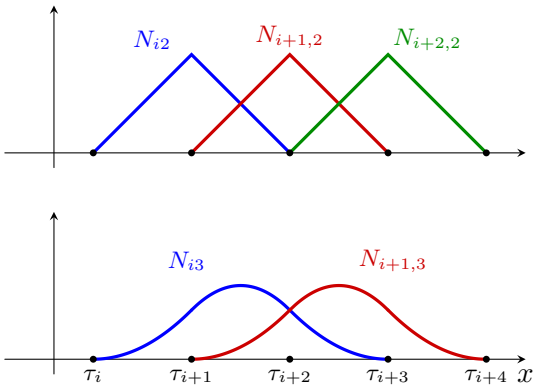


Figure 14: Order-2 hat functions (top) and order-3 quadratic bumps (bottom) on the knots $\tau_i, \ldots, \tau_{i+4}$.

Note:

$$N'_{i4}(\tau_i) = N''_{i4}(\tau_i) = 0$$
$$N'_{i4}(\tau_{i+4}) = N''_{i4}(\tau_{i+4}) = 0$$

Properties

(a) $\operatorname{supp} N_{ik} \subset [\tau_i, \tau_{i+k}]$

(b) $N_{ik}(x) \geq 0$ for all $x \in [\tau_n, \tau_l]$

(c) $\sum_{i=1}^{l-k} N_{ik}(x) = 1$ for all $x \in [\tau_1, \tau_l]$, i.e. the N_{ik}'s form a *partition of unity* on $[\tau_1, \tau_l]$

(d) $N_{ik} \in C^{k-2}[\tau_1, \tau_l]$

(e) $N_{ik} \in \prod_{k-1}$ on $[\tau_j, \tau_{j+1}]$, $\quad j = 1, 2, \ldots, l-1$

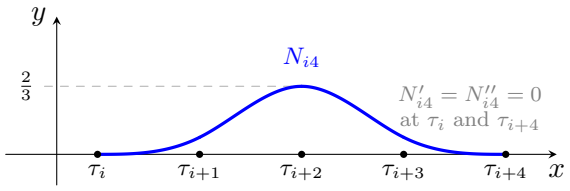


Figure 15: The cubic B-spline N_{i4} supported on $[\tau_i, \tau_{i+4}]$.

5.4.3 Filling in the Details: Running the Recursion by Hand

Take equally spaced knots $\tau_i = i$ and build upward. Order 1: boxes, $N_{i1} = \mathbf{1}_{[i,i+1)}$. Order 2:

$$N_{i2}(x) = \frac{x-i}{1} N_{i1}(x) + \frac{(i+2)-x}{1} N_{i+1,1}(x) = \begin{cases} x-i & i \leq x < i+1 \\ i+2-x & i+1 \leq x < i+2 \\ 0 & \text{else} \end{cases}$$

which is exactly the *hat function*: up-ramp times the left box plus down-ramp times the right box. Order 3 blends two hats into a C^1 piecewise-quadratic bump on $[i, i+3]$; order 4 gives the C^2 cubic bump on $[i, i+4]$.

Reading the recursion: each weight $\frac{x-\tau_i}{\tau_{i+j-1}-\tau_i}$ rises linearly from 0 to 1 across the support of its factor, and the two weights are *not* required to sum to 1, yet the resulting family still sums to 1 (property (c)).

Two of the properties, proved by induction on j:

(a) Support. N_{i1} lives on $[\tau_i, \tau_{i+1}]$. If $N_{i,j-1}$ lives on $[\tau_i, \tau_{i+j-1}]$ and $N_{i+1,j-1}$ on $[\tau_{i+1}, \tau_{i+j}]$, their combination lives on the union $[\tau_i, \tau_{i+j}]$: each order gains one interval. Hence cubic B-splines ($k = 4$) touch only 4 intervals: *locality*.

(c) Partition of unity. Sum the recursion over i and reindex the second term ($i + 1 \to i$):

$$\sum_i N_{ij} = \sum_i \left[\frac{x - \tau_i}{\tau_{i+j-1} - \tau_i} + \frac{\tau_{i+j-1} - x}{\tau_{i+j-1} - \tau_i} \right] N_{i,j-1} = \sum_i N_{i,j-1} = \cdots = \sum_i N_{i1} = 1.$$

The two weights multiplying the *same* $N_{i,j-1}$ (one from N_{ij}, one from $N_{i-1,j}$) sum to exactly 1; this identity carries the induction.

Why repeat boundary knots k times? The $0/0 := 0$ convention then makes the leftmost B-spline start at height 1 at $x = a$ (it degenerates to a one-sided function), so interpolation at a and b works and the basis "clamps" the interval ends. Interior knots stay simple, preserving C^{k-2} there.

5.4.4 The Five Properties as a User's Contract

For $S(x) = \sum_i \gamma_i N_{ik}(x)$:

- **(a) local support** $\Rightarrow$ at any x only k terms are nonzero: evaluation in $O(k^2)$, banded interpolation matrices, local data edits.
- **(b)** $N_{ik} \geq 0$ **and (c)** $\sum_i N_{ik} = 1 \Rightarrow S(x)$ is a *convex combination* of nearby coefficients: $\min_i \gamma_i \leq S(x) \leq \max_i \gamma_i$. No overshoot, no wild oscillation, and excellent conditioning ($\|S\|_\infty \leq \max |\gamma_i|$).
- **(d)** C^{k-2} **and (e) piecewise** $\prod_{k-1} \Rightarrow$ each $N_{ik} \in \mathcal{S}_{k,\Delta}$: we are inside the right space, and (next lecture) they form a *basis* of it.

This contract is why B-splines power CAD, computer graphics (NURBS), and finite elements (isogeometric analysis).

5.5 B-Splines: Properties and Evaluation

We complete the B-spline story by turning the recursion into algorithms. The triangular evaluation scheme computes all B-spline values at a point in $O(k^2)$ operations, independent of the number of knots, using only convex combinations of nonnegative numbers; the computation is unconditionally

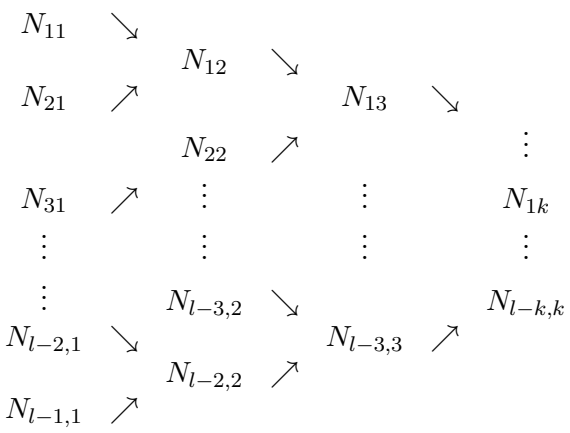
stable. Choosing knots with multiplicity k at the boundary makes the N_{ik} a genuine basis of $\mathcal{S}_{k,\Delta}$, and spline interpolation becomes a banded linear system solvable in $O(n)$ time. The coefficients (de Boor points) acquire a geometric meaning that underlies computer-aided design.

5.5.1 B-Splines (Continued)

This section completes the structural properties and turns to algorithms: evaluating a spline from its B-spline coefficients, and solving the interpolation problem.

$$\tau_1 \le \tau_2 \le \cdots \le \tau_l \tag{12}$$

For $j = 2, 3, \ldots, k:\ N_{ij}(x) = \cdots N_{i,j-1}(x) + \cdots N_{i+1,j-1}(x), \qquad i = 1, 2, \ldots, l-j$ (the recursion (11



5.5.1.1 Efficient Evaluation for $\tau_1 \le x < \tau_l$

Determine j such that $\tau_j \le x < \tau_{j+1} \quad (1 \le j < l,\ j \text{ is unique})$.

$$N_{j1}(x) = 1$$
$$N_{ji}(x) = 0 \quad \text{if } i \ne j$$

$$
\begin{array}{lllll}
 & & & & N_{1,j}(x) \quad \text{(column } j\text{)} \\
 & & & & \vdots \\
 & & N_{j-2,3}(x) & \nearrow & \\
 & N_{j-1,2}(x) & \diagdown\!\diagup & & \\
1 = N_{j1}(x) & \diagdown\!\diagup & N_{j-1,3}(x) & & \\
 & N_{j,2}(x) & \diagdown\!\diagup & & \\
 & & N_{j,3}(x) & \searrow & \\
 & & & \ddots & \\
 & & & & N_{j,l-j}(x) \quad \text{(column } l-j\text{)}
\end{array}
$$

5.5.1.2 Back to $\mathcal{S}_{k,\Delta}$

Choose the τ_j's (12) as follows:

$$
\underbrace{a}_{\substack{=\tau_1=\tau_2=\cdots=\tau_k \\ k \text{ times}}} = x_0 < \underbrace{x_1}_{=\tau_{k+1}} < \underbrace{x_2}_{=\tau_{k+2}} < \cdots < \underbrace{x_{n-1}}_{=\tau_{k+n-1}} < \underbrace{x_n = b}_{\substack{=\tau_{k+n}=\tau_{k+n+1}=\cdots=\tau_{n+2k-1} \\ k \text{ times}}}
$$

Set $l := n + 2k - 1 \quad (\Rightarrow l - k = n + k - 1 = \dim \mathcal{S}_{k,\Delta})$.

Corresponding B-splines of order k:

$$
N_{1k}, N_{2k}, \ldots, N_{n+k-1,k}
$$

Theorem 5.15.
The B-splines N_{ik}, $i = 1, 2, \ldots, n + k - 1$ form a basis of $\mathcal{S}_{k,\Delta}$.

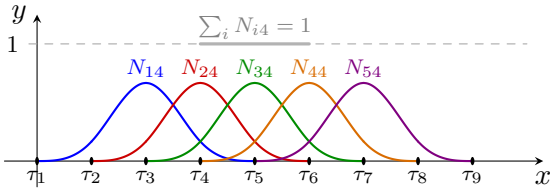


Figure 16: $\operatorname{supp} N_{i4} \subseteq [\tau_i, \tau_{i+4}]$. Also, $\sum_i N_{i4}(x) = 1$.

Notes:

1. The representation
$$
S(x) = \sum_{i=1}^{n+k-1} \gamma_i N_{ik}(x)
$$
of any $S \in \mathcal{S}_{k,\Delta}$ is well-conditioned.

2. The coefficients γ_i, $i = 1, 2, \ldots, n+k-1$ are called the *de Boor points* of S.

3. Since $N_{ik}(x) \geq 0$ and $\sum_{i=1}^{n+k-1} N_i(x) = 1$ for all $a \leq x \leq b$, the $S(x) = \sum_{i=1}^{n+k-1} \gamma_i N_{ik}(x)$ is a convex combination of the de Boor points $\gamma_1, \gamma_2, \ldots, \gamma_{l-k}$ for any fixed $x \in [a, b]$.

Example 5.16. ***Cubic Spline with Not-A-Knot Condition***
$k = 4$, $n \geq 3$.

$$\underbrace{x_0}_{=\tau_1=\tau_2=\tau_3=\tau_4} < x_1 < \underbrace{x_2}_{=\tau_5} < \cdots < \underbrace{x_{n-2}}_{=\tau_{n+1}} < x_{n-1} < \underbrace{x_n}_{=\tau_{n+2}=\tau_{n+3}=\tau_{n+4}=\tau_{n+5}}$$

$l = n + 5$. Corresponding B-splines $N_{14}, N_{24}, \ldots, N_{n+1,4}$.

$$S(x) = \sum_{i=1}^{n+1} \gamma_i N_{i4}(x)$$

Interpolating Conditions:

$$S(x_{j-1}) = y_{j-1}, \qquad j = 1, \ldots, n+1$$

$$M \begin{bmatrix} \gamma_1 \\ \gamma_2 \\ \vdots \\ \gamma_{n+1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}$$

where $M = [m_{ji}]_{j,i=1,2,\ldots,n+1}$ with $m_{ji} = N_{i4}(x_{j-1})$.

5.5.2 Filling in the Details: The Evaluation Scheme, Quantified

To evaluate at one point x with $\tau_j \leq x < \tau_{j+1}$: only $N_{j1}(x) = 1$ is nonzero at order 1, and by the support property only $N_{j-i+1,i}, \ldots, N_{j,i}$ can be nonzero at order i. So the triangular table has $1 + 2 + \cdots + k = \frac{k(k+1)}{2}$ entries, each computed from two below it with ≈ 5 flops:

$$\text{cost} \approx \tfrac{5}{2}k^2 \text{ flops, independent of } n.$$

For cubics: a few dozen flops per point, ever, no matter how many knots. Moreover every entry is a *convex combination of nonnegative numbers*: no

subtractions, no cancellation; the algorithm is unconditionally stable. (This is de Boor's algorithm in its simplest form.)

The interpolation matrix is banded. With $m_{ji} = N_{ik}(x_{j-1})$: at the node x_{j-1} at most k B-splines are nonzero, and they are *consecutive* in i. So M has bandwidth k (cubics: at most 4 nonzeros per row, in fact 3 at interior nodes since one B-spline vanishes at each knot). Consequences:

- storage $O(kn)$, factorization and solve $O(k^2 n)$, *linear* in the data size;
- M is "totally nonnegative"; solving with elimination without pivoting is provably safe here;
- compare: the truncated-power basis gives a dense, exponentially ill-conditioned matrix. Same space, same spline; the **basis** makes the computation.

5.5.3 De Boor Points: Geometry of the Coefficients

Because $N_{ik} \geq 0$ and $\sum_i N_{ik} \equiv 1$, at each fixed x,

$$S(x) = \sum_i \gamma_i N_{ik}(x)$$

is a *convex combination* of the de Boor points γ_i (weights: nonnegative, summing to 1). Hence:

- $\min_i \gamma_i \leq S(x) \leq \max_i \gamma_i$: the spline lies in the "control polygon's" shadow; coefficients *look like* the curve.
- Perturbing the coefficients moves the spline by at most the same amount: $\|\Delta S\|_\infty \leq \max_i |\Delta\gamma_i|$; the representation is perfectly conditioned in this sense.
- Changing one γ_i changes S only on $[\tau_i, \tau_{i+k}]$: *local control*, the property CAD designers need when dragging one control point.

This is the payoff of the whole B-spline development: a basis in which splines are stable to compute, cheap to evaluate, and geometrically intuitive to manipulate.

5.6 Notes and References

De Boor's *A Practical Guide to Splines* is the standard reference for everything in this chapter, including the B-spline recursion, the evaluation algorithms, and the total-nonnegativity results mentioned in passing. Deuflhard and Hohmann covers interpolation and splines in the style of these notes. The minimum-curvature property goes back to Holladay; the physical spline analogy predates the mathematics.

5.7 Exercises

1. Construct the Lagrange interpolating polynomial through the points $(-1, 2)$, $(0, 1)$, $(2, 7)$, simplify it to the monomial form, and verify the interpolation conditions.

2. (Computing.) Interpolate the Runge function $f(x) = 1/(1 + 25x^2)$ on $[-1, 1]$ at $n + 1$ equally spaced nodes for $n = 5, 10, 20$, and plot the interpolants against f. Describe the behavior near the endpoints as n grows.

3. Carry out the degrees-of-freedom count for quadratic splines ($k = 3$) and show that interpolation determines the spline up to exactly one extra condition. Propose a reasonable choice for that condition.

4. Show that the hat functions form a basis of $\mathcal{S}_{2,\Delta}$: they are linearly independent, they lie in the space, and their number equals its dimension.

5. On the uniform knots $\tau_i = i$, compute the quadratic B-spline N_{03} explicitly from the recursion, obtaining its three parabolic pieces. Verify that it is C^1 at the interior knots and vanishes outside $[0, 3]$.

6. Specialize the tridiagonal system for the natural cubic spline to equally spaced nodes $\xi_j = h$ and show that the coefficient matrix has rows $(h,\ 4h,\ h)$. Solve the system by hand for the data $(0, 0)$, $(1, 1)$, $(2, 0)$, $(3, 1)$ and write out the spline on the first subinterval.

7. Complete the equality case of the minimum-curvature theorem: if f is a competing interpolant with $\|f''\|_2 = \|S''\|_2$, show that $f = S$.

8. (Computing.) Compare `spline` (not-a-knot) and `interp1(...,'linear')` in MATLAB, or the SciPy equivalents, on $f(x) = \sin x$ sampled at 9 points on $[0, 2\pi]$: plot both interpolants and the pointwise errors.

Chapter 6

Numerical Integration

We construct and analyze quadrature rules: the interpolatory Newton–Cotes rules and their compound versions, Gaussian quadrature with its optimal degree of exactness, the nested Gauss–Kronrod pairs used in software, and adaptive algorithms that concentrate work where the integrand is difficult.

6.1 Numerical Integration; Quadrature Rules

We open the quadrature unit. After framing the general form of a quadrature rule (nodes, weights, and a remainder), we derive the trapezoidal and Simpson rules by integrating interpolants, obtain their error terms, and observe the bonus degree of exactness that symmetric rules enjoy. Compounding the rules over subintervals gives methods of orders h^2 and h^4, and comparing a rule with its refinement yields a computable error estimate at almost no extra cost. This idea is the seed of Romberg integration and of the adaptive algorithms to come.

6.1.1 Numerical Integration

We turn to the approximation of definite integrals. The general form of a quadrature rule is fixed in this section; the rest of the chapter concerns the choice of nodes and weights. Davis and Rabinowitz is the encyclopedic reference for the subject.

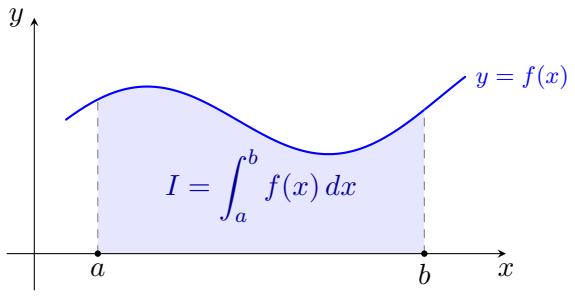


Figure 17: The integral $I = \int_a^b f(x)\,dx$ as the area under the graph of f.

Problem:

Evaluate

$$I = \int_a^b f(x)\,dx$$

where $f : (a, b) \to \mathbb{R}$ is integrable on $[a, b]$.

Quadrature Rules

$$I \approx \sum_{i=1}^{n} w_i f(x_i)$$

where $a \le x_1 < x_2 < \cdots < x_n \le b$ and $w_i \in \mathbb{R}$.

$$R_n := \sum_{i=1}^{n} w_i f(x_i) - I = (\underline{\text{remainder}} \text{ or } \underline{\text{error}})$$

6.1.2 Examples of Quadrature Rules

The two classical rules, trapezoidal and Simpson, arise from integrating interpolants. This section derives both with their error terms and assembles the compound versions used in computation.

1. Trapezoidal Rule. The simplest useful rule comes from the simplest useful approximation: replace f on $[a, b]$ by the straight line (chord) through its endpoint values, and integrate the line instead. The area under the chord is a trapezoid, whence the name:

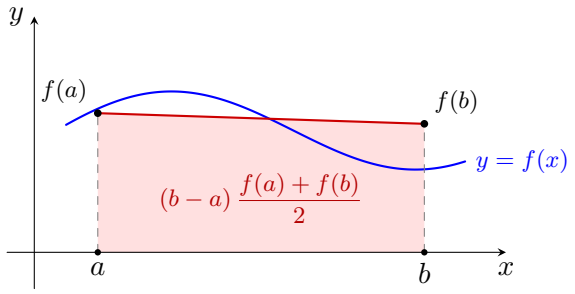


Figure 18: The trapezoidal rule approximates I by the area of the trapezoid under the chord.

$$I \approx (b-a)\frac{f(a)+f(b)}{2} = \frac{b-a}{2}f(a) + \frac{b-a}{2}f(b)$$

$n = 2, \quad x_1 = a, \quad x_2 = b, \quad w_1 = w_2 = \frac{b-a}{2}.$

$$R_2 = (b-a)^3 \frac{1}{12} f^{(2)}(\xi) \qquad \text{for some } \xi \in (a,b)$$

Thus, the trapezoid rule is exact for polynomials of degree ≤ 1, as it must be: a polynomial of degree ≤ 1 *is* its own chord. The error term involves f'', the quantity that measures how far f bends away from any straight line.

2. Simpson's Rule. To do better, interpolate at three points instead of two, the endpoints and the midpoint, and integrate the resulting quadratic:

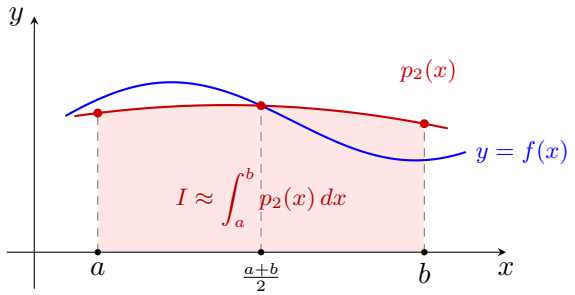


Figure 19: Interpolate the 3 points with a quadratic function.

$$I \approx \frac{b-a}{6}\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right)$$

$n = 3, \quad x_1 = a, \quad x_2 = \frac{a+b}{2}, \quad x_3 = b, \quad w_1 = w_3 = \frac{b-a}{6}, \quad w_2 = \frac{2}{3}(b-a).$

$$R_3 := (b-a)^5 \frac{1}{2880} f^{(4)}(\xi) \qquad \text{for some } \xi \in (a,b)$$

Thus, Simpson's rule is exact for polynomials of degree ≤ 3, one degree more than the quadratic interpolation used to build it would suggest. Where does this extra degree of accuracy come from? The answer is symmetry: for a cubic, the error committed on the left half of the interval exactly cancels the error on the right half, as the shaded regions in the next figure show for the model case $y = x^3$.

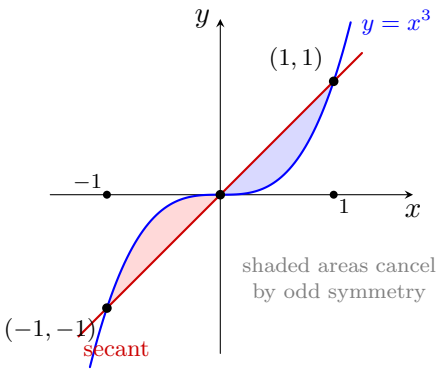


Figure 20: Simpson's rule for $y = x^3$ on $[-1, 1]$ is exact.

3. Compound trapezoidal rule. A single trapezoid across $[a, b]$ is crude, but the error scales like the *cube* of the interval length, so cutting the interval into small pieces pays off rapidly: apply the rule on each piece and add. Let $s \geq 1$ be an integer and set $t_i = a + ih$, $i = 0, 1, \ldots, s$, where $h := \frac{b-a}{s}$.

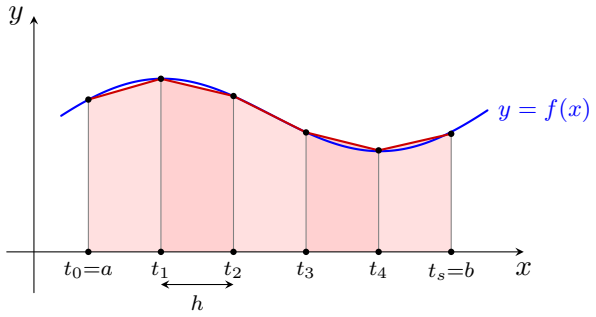


Figure 21: The compound trapezoidal rule on subintervals of width h at $t_0 = a, t_1, \ldots, t_s = b$.

$$
\begin{aligned}
I = \int_a^b f(x)\,dx &= \sum_{i=0}^{s-1} \int_{t_i}^{t_{i+1}} f(x)\,dx \\
&\approx \sum_{i=0}^{s-1} \frac{h}{2}\big[f(t_i) + f(t_{i+1})\big] \\
&= \frac{h}{2}\big[f(a) + 2f(a+h) + 2f(a+2h) + \cdots + 2f(b-h) + f(b)\big] \\
&= h\left[\frac{1}{2}f(a) + f(a+h) + f(a+2h) + \cdots + f(b-h) + \frac{1}{2}f(b)\right] \\
&=: T_{s+1}
\end{aligned}
$$

where $s+1$ is the number of points used. Error:

$$
\begin{aligned}
R_{s+1} = T_{s+1} - I = sh^3\frac{1}{12}f^{(2)}(\xi) \qquad &\text{for some } \xi \in (a,b) \\
= (b-a)\frac{1}{12}h^2 f^{(2)}(\xi)&
\end{aligned}
$$

T_{s+1} involves $s+1$ points $\quad (h = \frac{b-a}{s})$.
T_{2s+1} involves $2s+1$ points $\quad (\hat{h} = \frac{b-a}{2s} = \frac{h}{2})$.

$$
\begin{aligned}
T_{2s+1} = \underbrace{\hat{h}}_{\frac{h}{2}}\Big[&\frac{1}{2}f(a) + f(a+\hat{h}) \\
&+ \underbrace{f(a+2\hat{h})}_{f(a+h)} + \cdots + \underbrace{f(b-2\hat{h})}_{f(b-h)} + f(b-\hat{h}) + \frac{1}{2}f(b)\Big] \\
= \frac{1}{2}T_{s+1} &\\
&+ \hat{h}\Big[f(a+\hat{h}) + f(a+3\hat{h}) + \cdots + f(b-3\hat{h}) + f(b-\hat{h})\Big]
\end{aligned}
$$

Consequence: Once we have T_{s+1}, the approximation T_{s+1} can be obtained with s additional function evaluations! We can also estimate the error of T_{2s+1}:

$$
\begin{aligned}
R_{s+1} = T_{s+1} - I &= (b-a)\frac{1}{12}h^2 f^{(2)}(\xi) \\
R_{2s+1} = T_{2s+1} - I &= (b-a)\frac{1}{12}\frac{h^2}{4}f^{(2)}(\hat{\xi})
\end{aligned}
$$

Assume that $f^{(2)}(\xi) \approx f^{(2)}(\hat{\xi})$. Then

$$R_{2s+1} \approx \frac{1}{4} R_{s+1}.$$

But:

$$\begin{aligned} |T_{2s+1} - T_{s+1}| &= |R_{2s+1} - R_{s+1}| \\ &= \frac{3}{4} |R_{s+1}| \end{aligned}$$

So an error estimate for $T_{s+1} \approx |R_{s+1}| \approx \frac{4}{3} |T_{2s+1} - T_{s+1}|$. Approximate integral:

$$I \approx T_{2s+1}.$$

Conservative error estimate:

$$\frac{4}{3} |T_{2s+1} - T_{s+1}| .$$

6.1.3 Filling in the Details: Where the Error Terms Come From

Trapezoid. Interpolate f linearly at a, b and integrate the interpolant. The interpolation error formula gives

$$f(x) - p_1(x) = \frac{f''(\xi_x)}{2} (x-a)(x-b),$$

and integrating (the factor $(x-a)(x-b) \leq 0$ has one sign, so the mean value theorem for integrals applies):

$$I - (b-a)\tfrac{f(a)+f(b)}{2} = \frac{f''(\xi)}{2} \int_a^b (x-a)(x-b)\, dx = -\frac{(b-a)^3}{12} f''(\xi).$$

Sign says: for convex f ($f'' > 0$) the trapezoid *over*estimates; draw the chord to see it.

Simpson's bonus degree. Simpson integrates its quadratic interpolant, so degree ≤ 2 exactness is built in. Degree 3 comes free by *symmetry*: for $f(x) = (x - \frac{a+b}{2})^3$, both the exact integral and the rule give 0 (odd function about the midpoint; the weights $\frac{1}{6}, \frac{4}{6}, \frac{1}{6}$ are symmetric). Hence the error involves $f^{(4)}$, not $f^{(3)}$: order jumps from h^4 to h^5 per panel. Midpoint rule enjoys the same symmetry bonus. *Symmetric rules gain a degree.*

The error-estimate algebra (Richardson's idea), cleanly: with $R_{s+1} = Ch^2$ and $R_{2s+1} = C(h/2)^2 = \frac{1}{4}Ch^2$ (same C if f'' barely varies),

$$T_{2s+1} - T_{s+1} = R_{2s+1} - R_{s+1} = -\tfrac{3}{4}R_{s+1}$$
$$\implies \quad |R_{s+1}| = \tfrac{4}{3}\,|T_{2s+1} - T_{s+1}|\,, \quad |R_{2s+1}| \approx \tfrac{1}{3}\,|T_{2s+1} - T_{s+1}|\,.$$

So the *difference of two computable numbers estimates the error of each.* One can go further: $\frac{4T_{2s+1} - T_{s+1}}{3}$ cancels the h^2 term entirely, and the result is exactly compound Simpson (Romberg integration iterates this.)

Reuse bookkeeping: halving h only adds the *new midpoints*; old function values are all recycled. Cost of the refined estimate $= s$ new evaluations, not $2s{+}1$.

6.1.4 Choosing h: a Worked Budget

Compound trapezoid, $|R| \le \frac{b-a}{12}h^2 \max|f''|$. Suppose $[a,b] = [0,1]$, $\max|f''| \le 10$, target 10^{-8}:

$$\tfrac{10}{12}h^2 \le 10^{-8} \;\Rightarrow\; h \le 1.1 \times 10^{-4} \;\Rightarrow \approx 9{,}000 \text{ points.}$$

Same target with compound Simpson ($|R| \le \frac{b-a}{180}h^4 \max\left|f^{(4)}\right|$, say $\max\left|f^{(4)}\right| \le 100$):

$$\tfrac{100}{180}h^4 \le 10^{-8} \;\Rightarrow\; h \le 0.012 \;\Rightarrow \approx 90 \text{ points.}$$

Two orders of magnitude cheaper: higher order pays off once f is smooth. If f is *not* smooth (kinks, singularities), the rates degrade and adaptivity (Section 6.4) becomes the right tool.

6.2 Quadrature Examples; Gaussian Quadrature

After consolidating the bookkeeping of the compound Simpson rule, we ask how much accuracy n function evaluations can buy. Counting parameters suggests that n nodes and n weights might integrate polynomials of degree $2n-1$ exactly, twice the naive expectation. Realizing this bound is the achievement of Gaussian quadrature: the nodes must be the zeros of the nth orthogonal polynomial for the interval, objects we construct by Gram–Schmidt orthogonalization and whose zeros we show to be real, simple, and interior. The weights are then determined by exactness on polynomials of degree $\le n-1$.

6.2.1 Examples of Quadratures (Continued)

This section completes the bookkeeping of the compound Simpson rule and begins Gaussian quadrature, which answers the question of how much accuracy a fixed number of function evaluations can buy.

4. Compound Simpson's Rule. $s \geq 1, \quad h = \frac{b-a}{2s}$

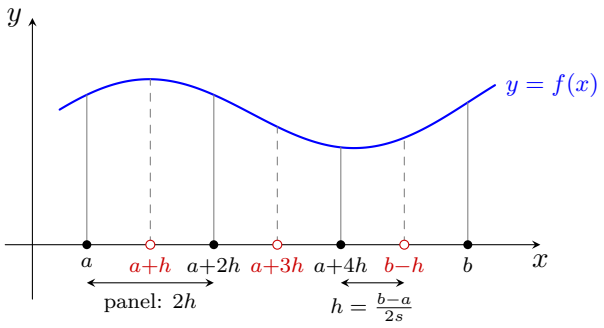


Figure 22: Nodes of the compound Simpson rule: panel endpoints (solid) and panel midpoints (open), with $h = \frac{b-a}{2s}$.

$$
\begin{aligned}
I = \int_a^b f(x)\,dx &= \sum_{i=0}^{s-1} \int_{a+2ih}^{a+2(i+1)h} f(x)\,dx \\
&\approx \frac{h}{3}\Big[\underbrace{f(a) + 4f(a+h) + f(a+2h)}\Big. \\
&\quad + \underbrace{f(a+2h) + 4f(a+3h) + f(a+4h)} + \cdots + f(b-2h) \\
&\quad \Big. + \underbrace{f(b-2h) + 4f(b-h) + f(b)}\Big] \\
&\approx \frac{h}{3}\big[f(a) + 4f(a+h) + 2f(a+2h) \\
&\quad + 4f(a+3h) + 2f(a+4h) + \cdots + 2f(b-2h) \\
&\quad + 4f(b-h) + f(b)\big] \\
&= S_{2s+1}
\end{aligned}
$$

This uses $2s+1$ points. Remainder:

$$R_{2s+1} = S_{2s+1} - I = (b-a)\frac{h^4}{180} f^{(4)}(\xi) \qquad \text{for some } \xi \in (a,b)$$

If we double s:

$$R_{4s+1} \approx \frac{1}{16} R_{2s+1}$$

Approximate integral: $I \approx S_{4s+1}$
Conservative error estimate: $\frac{16}{15} |S_{4s+1} - S_{2s+1}|$

Efficient Implementation

$$S_{2s+1} = \frac{h}{3} \left[f(a) + 4 \sum_{j=1}^{s} f\big(a + (2j-1)h\big) + f(b) + 2 \sum_{k=1}^{s-1} \sum f(a + 2kh) \right],$$

$$h = \frac{b-a}{2s}$$

$$= \frac{h}{3} \Bigg[\underbrace{f(a) + 2 \sum_{i=1}^{2s-1} f(a + ih) + f(b)}_{=:A_{2s+1}} + \underbrace{2 \sum_{j=1}^{s} f\big(a + (2j-1)h\big)}_{=:B_s} \Bigg]$$

$$S_{4s+1} = \frac{h}{6} \big[A_{4s+1} + B_{2s} \big]$$

$$\text{where} \qquad A_{4s+1} = f(a) + 2 \sum_{i=1}^{4s-1} f\left(a + i\frac{h}{2}\right) + f(b)$$

$$\text{and} \qquad B_{2s} = 2 \sum_{j=1}^{2s} f\left(a + (2j-1)\frac{h}{2}\right)$$

We get A_{4s+1} for free because

$$A_{4s+1} = \underbrace{f(a) + 2 \sum_{\substack{k=1 \\ (2k=i)}}^{2s-1} f(a + kh) + f(b)}_{=A_{2s+1}} + 2 \sum_{\substack{j=1 \\ (2j-1=i)}}^{2s} f\left(a + (2j-1)\frac{h}{2}\right)$$

$$= A_{2s+1} + B_{2s}$$

Consequence: To obtain S_{4s+1} from $S_{2s+1} = \frac{b-a}{6s}(A_{2s+1} + B_s)$, we only need to compute

$$B_{2s} = 2\sum_{j=1}^{2s} f\big(a + (2j-1)\hat{h}\big), \qquad \hat{h} = \frac{h}{2}$$

and set

$$\begin{aligned} A_{4s+1} &= A_{2s+1} + B_{2s} \\ S_{4s+1} &= \frac{b-a}{12s}(A_{4s+1} + B_{2s}) \end{aligned}$$

5. Gaussian Integration. So far all the formulas we've seen have been of the form

$$I = \int_a^b f(x)\,dx \approx \sum_{i=1}^{n} w_i f(x_i) \tag{13}$$

but with the nodes fixed in advance (endpoints, midpoints, uniform grids), leaving only the n weights to tune. Now release the nodes too. The expression then has $2n$ degrees of freedom, n nodes and n weights, and since $\dim \prod_{2n-1} = 2n$, we may hope to integrate exactly all polynomials of degree $\leq 2n-1$, roughly *twice* the degree the same number of function evaluations achieved before. The remarkable fact is that this hope is realized, by one specific choice of nodes. For Gaussian integration, we choose $a < x_1 < x_2 < \cdots < x_n < b$ to be the zeros of the nth orthogonal polynomial $p_n(x) = x^n + \cdots \in \prod_n$ defined by

$$\int_a^b p(x)p_n(x)\,dx = 0 \quad \text{for all } p \in \prod_n$$

($a = -1$, $b = 1$, $p_n(x) = \gamma_n \cdot (n\text{th Legendre polynomial})$.)

The w_i's are chosen such that (13) is exact for all $f \in \prod_{n-1}$, i.e.

$$\int_a^b f(x)\,dx = \sum_{i=1}^{n} w_i f(x_i), \qquad f \in \prod_{n-1}.$$

(For example: $f(x) = x^j$, $j = 0, 1, \ldots, n-1$.)

In this case, we get a linear system:

$$\begin{bmatrix} 1 & 1 & \cdots & 1 \\ x_1 & x_2 & \cdots & x_n \\ x_1^2 & x_2^2 & \cdots & x_n^2 \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{n-1} & x_2^{n-1} & \cdots & x_n^{n-1} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} = \begin{bmatrix} b-a \\ \frac{1}{2}(b^2-a^2) \\ \vdots \\ \frac{1}{n}(b^n-a^n) \end{bmatrix}$$

6.2.2 Filling in the Details: Orthogonal Polynomials from Scratch

Equip $\prod_\infty$ with the inner product $\langle p, q\rangle := \int_a^b p(x)q(x)\,dx$. Run Gram–Schmidt on $1, x, x^2, \ldots$:

$$p_0 = 1, \qquad p_n = x^n - \sum_{j=0}^{n-1} \frac{\langle x^n, p_j\rangle}{\langle p_j, p_j\rangle}\, p_j.$$

By construction p_n is monic, and $\langle p_n, q\rangle = 0$ for every $q \in \prod_{n-1}$. On $[-1,1]$ these are (monic multiples of) the *Legendre polynomials*: $p_1 = x$, $p_2 = x^2 - \frac{1}{3}$, $p_3 = x^3 - \frac{3}{5}x$, ...

The nodes are usable: the zeros of p_n are real, simple, and lie in (a,b). Sketch: let $x_1, \ldots, x_m$ be the sign changes of p_n in (a,b) and $q(x) = \prod(x - x_i) \in \prod_m$. Then $p_n q \geq 0$ and $\not\equiv 0$, so $\langle p_n, q\rangle > 0$, but orthogonality forces this to vanish if $m \leq n-1$. Hence $m = n$: all n zeros are inside and distinct. (For $n = 2$: nodes $\pm 1/\sqrt{3}$, weights $1, 1$; try it on $\int_{-1}^{1} x^2\,dx$.)

Two ways to get the weights.

1. *Moment equations* (the Vandermonde system above): impose exactness on $1, x, \ldots, x^{n-1}$; the matrix is Vandermonde in the (distinct) nodes, hence nonsingular, so the w_i exist and are unique.

2. *Lagrange shortcut:* exactness on $\prod_{n-1}$ means $w_i = \int_a^b \ell_i(x)\,dx$, the integral of the ith Lagrange basis polynomial. Same numbers, no linear solve.

Bonus fact (from the theory): all Gaussian weights are *positive*, crucial for stability, since positive weights mean no cancellation and $\sum_i |w_i| = b - a$ stays bounded as n grows. (Newton–Cotes rules lose exactly this property for large n, which is why they are never used at high order.)

In practice, nodes and weights are not computed from these definitions but from an eigenvalue problem for a tridiagonal "Jacobi matrix" (Golub–Welsch), a preview of Chapter 7.

6.2.3 The Degrees-of-Freedom Heuristic, Audited

Counting: n nodes + n weights = $2n$ parameters; $\dim \prod_{2n-1} = 2n$. The count *suggests* degree $2n-1$ might be reachable, but counting alone guarantees nothing (the equations are nonlinear in the nodes!). The achievement of Gauss:

- the specific choice "nodes = zeros of p_n" *decouples* the nonlinearity: with those nodes fixed, the weights come from a linear problem, and exactness on $\prod_{2n-1}$ follows by the division argument (next lecture's theorem);
- degree $2n-1$ is also *optimal*: no rule with n nodes is exact on $\prod_{2n}$ (we will see the one-line proof next time).

Efficiency comparison at $n = 5$: Gauss integrates degree-9 polynomials exactly; Newton–Cotes with 5 points only degree 4 (5 with symmetry).

6.3 Gaussian Integration

This lecture proves the two main theorems of the quadrature unit: Gaussian quadrature is exact on polynomials of degree $2n-1$ (by a division-with-remainder argument in which orthogonality and the choice of nodes each contribute one line), and no rule with n nodes can do better (test the square of the node polynomial). We then weigh the practical pros and cons of Gaussian rules; their one drawback, that refining the rule discards all previous function values, is repaired by the Gauss–Kronrod construction, which extends an n-point Gaussian rule to a nested $(2n+1)$-point rule and provides the error estimator used in modern software.

6.3.1 Gaussian Integration (Continued)

This section proves the two theorems that characterize Gaussian quadrature: exactness on polynomials of degree $2n-1$, and the optimality of that degree. It then addresses the practical drawback of Gaussian rules and its resolution, the Gauss–Kronrod pairs.

$$I = \int_a^b f(x)\,dx \approx \sum_{i=1}^{n} w_i f(x_i) \tag{14}$$

Choose the x_i as the zeros of the nth orthogonal polynomial $p_n(x)$: $a < x_1 < x_2 < \cdots < x_n < b$. The w_i's are chosen so that

$$\int_a^b f(x)\,dx = \sum_{i=1}^{n} w_i f(x_i)$$

for all $f \in \prod_{n-1}$.

Theorem 6.1.
(14) is exact for all $p \in \prod_{2n-1}$.

Proof. Let $p \in \prod_{2n-1}$. $0 = \int_a^b p_n(x)q(x)\,dx$ for all $q \in \prod_{n-1}$. Polynomial division:

$$p(x) = p_n(x)q(x) + r(x) \qquad \text{where } q, r \in \prod_{n-1}$$

$$\begin{aligned}
\int_a^b p(x)\,dx &= \underbrace{\int_a^b p_n(x)q(x)\,dx}_{\substack{=0 \text{ b/c} \\ p_n = n\text{th orthogonal polynomial}}} + \int_a^b r(x)\,dx \\
&= \sum_{i=1}^{n} w_i r(x_i) \\
&= \sum_{i=1}^{n} w_i [\overbrace{p_n(x_i)}^{=0} q(x_i) + r(x_i)] \\
&= \sum_{i=1}^{n} w_i p(x_i)
\end{aligned}$$

□

6.3.1.1 Pros and Cons of Gaussian Integration

Pros	Cons
• Optimal Accuracy • Can be used for functions with singularities at $x = a$ or $x = b$ (since $a < x_1 < x_2 < \cdots < x_n < b$)	• When n is increased, old values $f(x_1), f(x_2), \ldots, f(x_n)$ cannot be reused

6.3.1.2 Gauss-Kronrod Rules

Given: Gaussian rule with n points,

$$G_n = \sum_{i=1}^{n} w_i f(x_i).$$

Gauss-Kronrod:

$$K_{2n+1} = \sum_{i=1}^{n} a_i f(x_i) + \sum_{j=1}^{n+1} b_j f(y_j)$$

where the y_j's are chosen as the zeros of the polynomial $q_{n+1}(x) = x^{n+1} + \cdots \in \prod_{n+1}$ satisfying

$$\int_a^b p_n(x)p(x)q_{n+1}(x)\,dx = 0 \quad \text{for all } p \in \prod_n.$$

Thus, q_{n+1} is the $(n+1)$st orthogonal polynomial with respect to the "inner product" $(p,q) := \int_a^b p(x)q(x)p_n(x)\,dx$.

The remaining parameters a_i, $i = 1, 2, \ldots, n$ and b_j, $j = 1, 2, \ldots, n+1$ are determined such that

$$\int_a^b p(x)\,dx = \sum_{i=1}^{n} a_i p(x_i) + \sum_{j=1}^{n+1} b_j p(y_j) \quad \text{for all } p \in \prod_{2n}.$$

Theorem 6.2.
K_{2n+1} is exact for all $p \in \prod_{3n+1}$.

Proof. For $p \in \prod_{3n+1}$:

$$p = (p_n q_{n+1})t + r, \qquad r \in \prod_{2n},\ t \in \prod_n$$

$$\begin{aligned}
\int_a^b p(x)\,dx &= \underbrace{\int_a^b p_n(x)q_{n+1}(x)t(x)\,dx}_{=0} + \int_a^b r(x)\,dx \\
&= \sum_{i=1}^{n} a_i r(x_i) + \sum_{j=1}^{n+1} b_j r(y_j) \\
&= \sum_{i=1}^{n} a_i p(x_i) + \sum_{j=1}^{n+1} b_j p(y_j)
\end{aligned}$$

□

6.3.1.3 Practical Use

Pair (G_n, K_{2n+1}). Approximate the integral:

$$K_{2n+1} \approx \int_a^b f(x)\,dx.$$

Heuristic error estimate:

$$\big(200\,|G_n - K_{2n+1}|\big)^{3/2}.$$

If we start with n reasonably large (e.g. $n = 10$), then K_{2n+1} will probably be accurate enough. However, if we need to double n then we have to start from scratch.

6.3.2 Filling in the Details: The Division Argument, Annotated

Given $p \in \prod_{2n-1}$, divide by the monic p_n:

$$p = p_n q + r, \qquad \deg q \le n-1,\ \deg r \le n-1.$$

Now compute both sides of the quadrature identity:

$$\int_a^b p = \underbrace{\int_a^b p_n q}_{=0\ \text{(orthogonality, } q\in\prod_{n-1}\text{)}} + \int_a^b r \overset{(*)}{=} \sum_i w_i r(x_i) \overset{(**)}{=} \sum_i w_i p(x_i).$$

$(*)$: the weights were *constructed* to be exact on $\prod_{n-1}$, and $r \in \prod_{n-1}$. $(**)$: at each node, $p(x_i) = p_n(x_i)q(x_i) + r(x_i) = r(x_i)$ because the nodes are the *zeros* of p_n. Both special choices (orthogonal p_n for the integral, its zeros for the nodes) click into place in one line each.

Optimality in one line. Test $p = p_n^2 \in \prod_{2n}$:

$$\int_a^b p_n^2 > 0 \qquad \text{but} \qquad \sum_i w_i\, p_n(x_i)^2 = 0.$$

No n-point rule (whatever its nodes) can be exact on all of $\prod_{2n}$: just build the same counterexample from its own node polynomial. Degree $2n-1$ is a hard ceiling.

Gauss–Kronrod, and why it exists. The pain point: G_n and G_{2n} share no nodes: doubling n wastes all previous f-evaluations, and $|G_{2n} - G_n|$ as an error estimate costs a full extra rule. Kronrod's fix: *keep* the n Gauss

nodes, add $n+1$ new nodes (zeros of the Stieltjes polynomial q_{n+1}, orthogonal w.r.t. the signed weight $p_n(x)\,dx$), and re-tune all $2n+1$ weights. Budget check:

$$\underbrace{2n+1}_{\text{free weights}} + \underbrace{n+1}_{\text{new nodes}} = 3n+2 \text{ parameters} \quad \leftrightarrow \quad \dim \prod_{3n+1} = 3n+2. \checkmark$$

The pair (G_n, K_{2n+1}) then delivers: high-order answer K_{2n+1}, error estimate from $|G_n - K_{2n+1}|$, at only $n+1$ evaluations beyond G_n. MATLAB's `quadgk` and the QUADPACK library are built on exactly the (G_7, K_{15}) pair applied adaptively, which is where we go next.

6.3.3 Summary Table: Rules and Their Exactness

Rule	points	exact on	error order
Trapezoid	2	$\prod_1$	$(b-a)^3 f''$
Simpson	3	$\prod_3$	$(b-a)^5 f^{(4)}$
Compound trapezoid	$s+1$	$\prod_1$ each panel	$O(h^2)$
Compound Simpson	$2s+1$	$\prod_3$ each panel	$O(h^4)$
Gauss G_n	n	$\prod_{2n-1}$	$\propto f^{(2n)}$
Kronrod K_{2n+1}	$2n+1$	$\prod_{3n+1}$	–

Reading: for smooth integrands, exactness degree translates directly into convergence rate; Gauss buys the most accuracy per function evaluation. For rough integrands, all rates collapse, motivating adaptivity.

6.4 Adaptive Quadrature; Eigenvalue Problems

The first half of this lecture assembles adaptive quadrature: equipped with a rule and an error estimate, the algorithm bisects exactly those subintervals whose estimated error exceeds a budget proportional to their length, concentrating work where the integrand is difficult. This completes the integration unit. The second half opens the final unit of the course, eigenvalue problems: definitions, the characteristic polynomial, and a structural fact: by Abel's theorem, no finite algorithm can compute eigenvalues of general matrices of size $n \geq 5$, so every eigenvalue algorithm must be iterative.

6.4.1 Adaptive Quadrature

Fixed rules spend function evaluations uniformly; adaptive methods spend them where the integrand is difficult. This section assembles the standard recursive algorithm from a rule and an error estimate. The design, essentially that of the QUADPACK library, underlies most library integrators in use today.

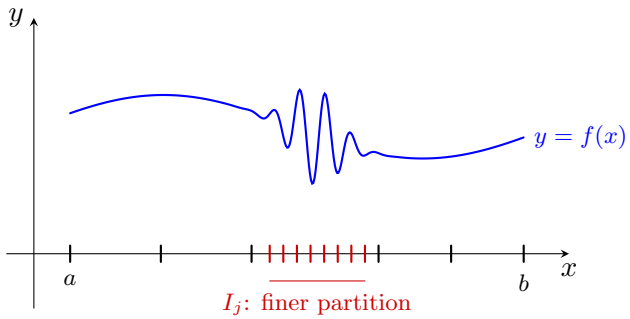


Figure 23: Adaptive quadrature: the partition is refined on the subinterval I_j where f varies rapidly.

Idea: Use more function values $f(x_j)$ where f "varies more."

Ingredients

- Quadrature rule Q with error estimate E.
- Tolerance $\epsilon > 0$.

Goal: Approximate the integral $Q_{[a,b]}$ such that

$$\left| Q_{[a,b]} - \int_a^b f(x)\,dx \right| \lessapprox E_{[a,b]} < \epsilon.$$

6.4.1.1 Basic Adaptive Procedure

The algorithm maintains, at every stage, a partition of the interval into nonoverlapping pieces,

$$[a,b] = I_1 \cup I_2 \cup \cdots \cup I_l, \qquad I_j = [a_j, b_j], \quad h_j = b_j - a_j,$$

so that $h_1 + h_2 + \cdots + h_l = b - a$, together with, for each piece, an approximation and a local error estimate:

$$Q_j \approx \int_{I_j} f(x)\,dx, \qquad \left| Q_j - \int_{I_j} f(x)\,dx \right| \lessapprox E_j.$$

Each piece is accepted when its estimated error fits within an allowance proportional to its length; pieces that fail are bisected and re-examined.

Algorithm 7 Adaptive Quadrature

Require: quadrature rule Q with error estimate E; tolerance $\epsilon > 0$; minimum length $h_{\min} > 0$

Ensure: $Q_{[a,b]} \approx \int_a^b f(x)\,dx$ with estimated error $< \epsilon$

$l \leftarrow 1, \quad I_1 \leftarrow [a,b], \quad h_1 \leftarrow b-a$
apply Q to I_1, obtaining Q_1, E_1
while $E_j \geq \frac{h_j}{b-a}\,\epsilon$ for some $j \in \{1,\dots,l\}$ **do**
    for all $I_j = [a_j, b_j]$ with $E_j \geq \frac{h_j}{b-a}\,\epsilon$ **do**
        if $h_j/2 < h_{\min}$ **then**
            stop: tolerance not attainable ▷ safeguard against singular or noisy f
        end if
        bisect: $I_j \leftarrow \left[a_j, \frac{a_j+b_j}{2}\right], \quad I_{l+1} \leftarrow \left[\frac{a_j+b_j}{2}, b_j\right]$
        $h_j \leftarrow h_{l+1} \leftarrow \frac{b_j-a_j}{2}, \quad l \leftarrow l+1$
        apply Q to both halves, obtaining Q_j, E_j and Q_l, E_l
    end for
end while
return $Q_{[a,b]} = \sum_{j=1}^{l} Q_j$ ▷ $E_{[a,b]} = \sum_j E_j < \frac{\epsilon}{b-a}\sum_j h_j = \epsilon$

On termination every piece satisfies its budget, and because the budgets are proportional to the lengths, they sum to exactly ϵ, as the final comment records. In library implementations Q is a Gauss–Kronrod pair (typically (G_7, K_{15})), the interval with the *largest* error estimate is bisected first, and additional safeguards bound the total number of subdivisions.

6.4.2 Eigenvalue Problems

The final chapter of the course concerns the computation of eigenvalues. This section fixes the definitions and records the basic facts. Golub and Van Loan and Wilkinson's treatise are the standard references for the entire chapter.

Definition 6.3. ***Eigenvalue, Eigenvector***
Let $A \in \mathbb{C}^{n\times n}$. A number $\lambda \in \mathbb{C}$ is called an *eigenvalue* of A if there exists a nonzero $x \in \mathbb{C}^n$, $x \neq 0$, such that

$$Ax = \lambda x.$$

Such an x is called an *eigenvector* of A.

Definition 6.4. ***Spectrum***
The set

$$\Lambda(A) := \{\lambda \in \mathbb{C} \mid \lambda \text{ is an eigenvalue of } A\}$$

is called the *spectrum* of A.

$$\begin{aligned} & Ax = \lambda x, \quad x \neq 0 \\ \Rightarrow\ & (\lambda I - A)x = 0, \quad x \neq 0 \\ \Rightarrow\ & \text{the matrix } \lambda I - A \text{ is singular} \\ \Rightarrow\ & \det(\lambda I - A) = 0 \end{aligned}$$

Definition 6.5. ***Characteristic Polynomial***
The polynomial

$$p(z) = p_A(z) := \det(zI - A) = z^n + \alpha_{n-1}z^{n-1} + \cdots + \alpha_1 z + \alpha_0$$

is called the *characteristic polynomial* of A.

Notes:

1. The eigenvalues of A are the zeros of the characteristic polynomial, p_A. In particular, $A \in \mathbb{C}^{n\times n}$ has n eigenvalues, but they are not necessarily distinct.

2. For $n \geq 5$, any algorithm for computing eigenvalues of general $A \in \mathbb{C}^{n\times n}$ has to be iterative!

    - Even if this was not true, working with polynomials is not great.

6.4.3 Filling in the Details: Why the Budget Is Proportional to Length

The acceptance test $E_j < \frac{h_j}{b-a}\,\epsilon$ hands each subinterval an error allowance *proportional to its length.* Then, on termination,

$$E_{[a,b]} = \sum_j E_j < \frac{\epsilon}{b-a}\sum_j h_j = \epsilon :$$

the local budgets provably add up to the global tolerance, because the lengths add up to $b - a$. A *uniform* per-interval budget ($E_j < \epsilon/l$) would

fail: l changes as we subdivide, and thousands of tiny intervals would each claim the same allowance.

Why bisection concentrates work correctly: for a rule with local error $\sim h^{p+1}$, halving an offending interval cuts its error by $\approx 2^{p+1}$ while its budget only halves, so a few bisections fix any smooth region, and the recursion drills down *only* near kinks, spikes, or singularities. Smooth parts terminate at coarse h; total work concentrates exactly where f is hard. Safeguards ($h_j \geq h_{\min}$, maximum interval count) guard against nonintegrable singularities and noisy f, where estimates lie.

From polynomials to matrices and back. Two directions:

- eigenvalues → polynomial roots: $\Lambda(A) =$ zeros of $\det(zI - A)$;
- polynomial roots → eigenvalues: every monic p is the characteristic polynomial of its *companion matrix* (next lecture's example).

The two problems are *equivalent*. Abel–Ruffini says degree-≥ 5 polynomial roots admit no closed formula in radicals; hence no finite algorithm can compute eigenvalues of a general $n \geq 5$ matrix: every eigen-solver must iterate. (Direction: modern practice goes polynomial → companion matrix → QR iteration (`roots` in MATLAB), the *opposite* of the naive char-poly route, because polynomial root-finding from coefficients is ill-conditioned while QR on the matrix is backward stable.)

6.4.4 A 2×2 Eigenvalue Warm-Up

$$A = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}: \qquad p_A(z) = \det \begin{bmatrix} z-2 & -1 \\ -1 & z-2 \end{bmatrix} = (z-2)^2 - 1 = (z-1)(z-3).$$

$\Lambda(A) = \{1, 3\}$; eigenvectors $(1, -1)^T$ and $(1, 1)^T$ (orthogonal; A is symmetric). Check the invariants: $\operatorname{tr} A = 4 = 1 + 3$, $\det A = 3 = 1 \cdot 3$.

Now perturb: $\hat{A} = \begin{bmatrix} 2 & 1 \\ 1 + 10^{-10} & 2 \end{bmatrix}$ has eigenvalues $2 \pm \sqrt{1 + 10^{-10}}$; they move by $\sim 5 \times 10^{-11}$: benign here. But for the Jordan-type matrix $\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$, adding 10^{-10} in the $(2, 1)$ entry moves the double eigenvalue 0 to $\pm 10^{-5}$: a *square-root* response. Eigenvalue conditioning varies wildly with the matrix, one reason the robust algorithms (Schur, QR) avoid fragile objects like the Jordan form entirely.

6.5 Notes and References

Deuflhard and Hohmann treats quadrature in the order followed here. Davis and Rabinowitz, *Methods of Numerical Integration*, is the classical comprehensive reference. Gaussian nodes and weights are computed in practice by the Golub–Welsch algorithm from the eigenvalues of the Jacobi matrix, an application of the next chapter's methods. The (G_7, K_{15}) pair and its adaptive driver come from QUADPACK, whose design underlies most library integrators.

6.6 Exercises

1. Verify by direct computation that Simpson's rule integrates $f(x) = x^3$ exactly on $[0, 1]$, and that it fails to be exact for $f(x) = x^4$. What error does it commit in the latter case, and how does this compare with the error-term formula?

2. Derive the midpoint rule error $I - (b-a) f(m) = \frac{(b-a)^3}{24} f''(\xi)$, where m is the midpoint, by expanding f in a Taylor series about m and integrating.

3. Set up the moment equations for the two-point Gaussian rule on $[-1, 1]$ and solve them to obtain the nodes $\pm 1/\sqrt{3}$ and weights $1, 1$. Check exactness on x^2 and x^3, and compute the error for x^4.

4. Prove that all Gaussian weights are positive. (Apply the exactness of the rule on $\prod_{2n-1}$ to $p = \ell_i^2$, where ℓ_i is the ith Lagrange basis polynomial of the nodes.)

5. Compute the monic orthogonal polynomials p_2 and p_3 on $[-1, 1]$ by Gram–Schmidt orthogonalization of $1, x, x^2, x^3$, and verify that their zeros are real, simple, and interior.

6. (Computing.) Apply the compound trapezoidal rule to $\int_0^1 e^x \, dx$ with $s = 2, 4, 8, \dots, 256$ panels. Verify the $O(h^2)$ convergence, and check that the Richardson combination $(4T_{2s+1} - T_{s+1})/3$ converges at order h^4.

7. (Computing.) Integrate $\int_0^1 \sqrt{x} \, dx$ with the compound Simpson rule at tolerance 10^{-8} using uniform refinement, and then with a recursive adaptive Simpson method at the same tolerance. Report the number of function evaluations each requires and explain the difference.

Chapter 7

Eigenvalue Problems

The final unit develops the modern eigenvalue algorithm. Since no finite algorithm can exist, the method is iterative: reduce to Hessenberg form by Householder similarities, then apply the shifted QR iteration with Givens rotations and deflation: in essence, the algorithm inside production eigenvalue libraries.

7.1 Computing Eigenvalues; Unitary Similarity

We first make the equivalence between eigenvalue problems and polynomial root-finding precise via the companion matrix, and then explain why the two classical textbook routes, computing roots of the characteristic polynomial and computing the Jordan canonical form, are both numerically unstable. The constructive path forward is the similarity transformation: similar matrices share a spectrum, and among all similarities the unitary ones neither amplify errors ($\kappa_2(U) = 1$) nor require inversion. Schur's theorem, stated here and proved next lecture, guarantees that unitary similarities suffice to reach triangular form, where the eigenvalues sit on the diagonal.

7.1.1 Eigenvalue Problems (Continued)

This section makes the connection between eigenvalues and polynomial roots exact, in both directions, and draws the structural conclusion: eigenvalue algorithms must iterate.

3. Any polynomial $p(z) = z^n + \alpha_{n-1}z^{n-1} + \alpha_{n-2}z^{n-2} + \cdots + \alpha_1 z + \alpha_0$ is the characteristic polynomial of a matrix $A \in \mathbb{C}^{n\times n}$. For example,

$$A = \begin{bmatrix} 0 & 0 & \cdots & 0 & -\alpha_0 \\ 1 & \ddots & & \vdots & -\alpha_1 \\ 0 & \ddots & \ddots & 0 & \vdots \\ \vdots & \ddots & \ddots & 0 & -\alpha_{n-2} \\ 0 & \cdots & 0 & 1 & -\alpha_{n-1} \end{bmatrix}$$

has the characteristic polynomial

$$\det(zI - A) = \det \begin{bmatrix} z & & & 0 & \alpha_0 \\ -1 & z & & & \alpha_1 \\ & -1 & \ddots & & \vdots \\ & & \ddots & z & \alpha_{n-2} \\ 0 & & & -1 & z + \alpha_{n-1} \end{bmatrix}$$
$$= z^n + \alpha_{n-1} z^{n-1} + \cdots + \alpha_1 z + \alpha_0.$$

For some classes of matrices, the eigenvalues can be found trivially. For example, an upper triangular matrix:

$$A = \begin{bmatrix} a_{11} & * & \cdots & * \\ & a_{22} & \ddots & \vdots \\ & & \ddots & * \\ 0 & & & a_{nn} \end{bmatrix}, \qquad \Lambda(A) = \{a_{11}, a_{22}, \ldots, a_{nn}\}$$

7.1.2 Computation of Eigenvalues

Knowing that iteration is unavoidable, we must decide what to iterate toward. This section rules out the two classical answers, the characteristic polynomial and the Jordan canonical form, on grounds of conditioning, and identifies the right target: the Schur form.

7.1.2.1 Bad Ideas

1. Forming p_A and computing $\lambda \in \Lambda(A)$ as zeros of p_A.
2. Attempting to compute the Jordan canonical form of A:

$$X^{-1} A X = \begin{bmatrix} J_1 & & & 0 \\ & J_2 & & \\ & & \ddots & \\ 0 & & & J_l \end{bmatrix},$$

where each

$$J_i = \begin{bmatrix} \lambda_i & 1 & & 0 \\ & \lambda_i & 1 & \\ & & \ddots & \ddots \\ & & & \ddots & 1 \\ 0 & & & & \lambda_i \end{bmatrix}, \qquad i = 1, 2, \ldots, l$$

are Jordan blocks.

```
[J,X] = jordan(A)
```

7.1.2.2 Better Ideas

1. Use the Schur factorization of A.
2. Use unitary similarity transformations to first transform A to a "simpler form."

Definition 7.1. ***Similarity Transformations***
Let $A \in \mathbb{C}^{n\times n}$. Let $U \in \mathbb{C}^{n\times n}$ be nonsingular. The map

$$A \mapsto U^{-1}AU$$

is called a *similarity transformation* of A.

Lemma 7.2.

$$\Lambda(A) = \Lambda(U^{-1}AU)$$

Proof.

$$\begin{gathered} Ax = \lambda x, \quad x \neq 0 \\ U^{-1}AU(\underbrace{U^{-1}x}_{\tilde{x}}) = \lambda(\underbrace{U^{-1}x}_{\tilde{x}}), \quad \tilde{x} \neq 0 \\ U^{-1}AU\tilde{x} = \lambda\tilde{x}, \quad \tilde{x} \neq 0 \end{gathered}$$

□

Use matrices U that are easy to invert and that are numerically well-behaved:

Definition 7.3. ***Unitary***
$U \in \mathbb{C}^{n\times n}$ is said to be *unitary* if

$$U^H U = I$$

$(U = [u_{jk}], \quad U^H = [\overline{u_{kj}}] = (\overline{U})^T.)$

Notes:

1. U unitary $\Rightarrow U$ is nonsingular and $U^{-1} = U^H$.

2. U unitary $\Rightarrow U^H$ is unitary and $(U^H)^{-1} = (U^H)^H = U$.

3. U unitary $\Rightarrow \|Ux\|_2 = \|x\|_2$ for all $x \in \mathbb{C}^n$. This is because $\|Ux\|_2^2 = (Ux)^H Ux = x^H U^H Ux = x^H x = \|x\|_2^2$ and $\|U\|_2 = \max_{x \neq 0} \frac{\|Ux\|_2}{\|x\|_2} = 1$.

4. Unitary matrices U have the best possible Euclidean condition numbers:
$$\kappa_2(U) = \underbrace{\|U\|_2}_{=1} \underbrace{\left\| \overbrace{U^{-1}}^{=U^H} \right\|_2}_{=1} = 1.$$

7.1.3 Unitary Similarity Transformations

Similarity transformations preserve eigenvalues; unitary similarity transformations also preserve error bounds. This section proves both facts. The preference for unitary transformations is the organizing principle of practical eigenvalue computation, a point Trefethen and Bau emphasize throughout their treatment.

$$A \mapsto U^{-1}AU = U^H AU =: T$$

where U is unitary.

Question: What is the "simplest" T we can achieve?

Theorem 7.4. ***Schur***
For any $A \in \mathbb{C}^{n \times n}$, there exists a unitary $U \in \mathbb{C}^{n \times n}$ such that

$$U^H AU = T = \begin{bmatrix} t_{11} & * & \cdots & * \\ & t_{22} & \ddots & \vdots \\ & & \ddots & * \\ 0 & & & t_{nn} \end{bmatrix}$$

is upper-triangular. In particular, $\Lambda(A) = \Lambda(T) = \{t_{11}, t_{22}, \ldots, t_{nn}\}$.

We will prove this by induction on n. For $n = 1$: $A = [a]$, $U = [1]$, $T = [a]$.

7.1.4 Filling in the Details: Why the Bad Ideas Are Bad

Bad idea 1 (roots of p_A): the map coefficients $\rightarrow$ roots is severely ill-conditioned. Classic example (Wilkinson): the polynomial with roots

$1, 2, \ldots, 20$; perturbing *one coefficient* in its 10th digit moves several roots by $O(1)$ and sends some into the complex plane. Yet those same numbers, as eigenvalues of a symmetric matrix, are perfectly conditioned. Passing through the coefficient representation *destroys* conditioning so we must work on the matrix directly.

Bad idea 2 (Jordan form): the Jordan structure is *discontinuous* in the entries:

$$\begin{bmatrix}\lambda & 1\\ 0 & \lambda\end{bmatrix} \text{ (one } 2\times 2 \text{ block)} \quad \text{vs} \quad \begin{bmatrix}\lambda & 1\\ \varepsilon & \lambda\end{bmatrix} \text{ (two distinct eigenvalues } \lambda \pm \sqrt{\varepsilon}).$$

Arbitrarily small ε (e.g. one rounding error) changes the block structure, and the transformation X becomes arbitrarily ill-conditioned. No stable algorithm can compute a discontinuous object.

Why unitary similarity is the cure. For the transformations $A \mapsto U^H A U$ we will use:

- spectra are preserved (today's lemma), so we never lose the answer;
- $\kappa_2(U) = 1$: transforming by U amplifies *no* errors. Concretely, if we compute $\tilde{T} = U^H(A + E)U$ with backward error E, then $\|E\|_2$ is unchanged by U: errors are *moved around, never magnified*;
- $U^{-1} = U^H$ is available for free: no inversion, no growth.

Contrast general similarity $X^{-1}AX$: it also preserves spectra, but $\kappa(X)$ multiplies every error bound. The whole modern eigenvalue technology = "do everything with unitary matrices," and Schur's theorem says unitary matrices *suffice* to reach triangular form, which exposes the eigenvalues on the diagonal. That existence proof is next lecture.

7.1.5 Verifying the Companion Matrix Claim ($n = 3$)

$$A = \begin{bmatrix}0 & 0 & -\alpha_0\\ 1 & 0 & -\alpha_1\\ 0 & 1 & -\alpha_2\end{bmatrix}, \qquad zI - A = \begin{bmatrix}z & 0 & \alpha_0\\ -1 & z & \alpha_1\\ 0 & -1 & z+\alpha_2\end{bmatrix}.$$

Expand along the first row:

$$\det(zI - A) = z\big(z(z+\alpha_2) + \alpha_1\big) + \alpha_0 = z^3 + \alpha_2 z^2 + \alpha_1 z + \alpha_0. \checkmark$$

(The general case follows by the same expansion / induction: each subdiagonal -1 shifts the powers up by one.)

So *any* monic polynomial is a characteristic polynomial, so eigenvalue computation is at least as hard as polynomial root-finding, completing the equivalence from last lecture.

7.2 Schur's Theorem; Householder Reflectors

Two constructions fill this lecture. First we prove Schur's theorem by deflation: an eigenpair supplies the first column of a unitary similarity that splits off one eigenvalue, and induction does the rest. Second, we introduce the Householder reflectors, the unitary building blocks of practical algorithms: for any vector x we construct a reflector mapping x to a multiple of e_1, choosing the sign in the construction to avoid cancellation. Reflectors are Hermitian, unitary, cheap to apply, and are exactly the tool that turns the existence proof of Schur's theorem into computation.

7.2.1 Proof of Schur's Theorem

The proof is by deflation: one eigenpair reduces the problem size by one, and induction finishes the argument. The proof is constructive except for a single step, extending a unit vector to a unitary matrix, which the Householder reflectors of the next section make computational.

Recall:

$$U^H A U = T = \text{upper-triangular}$$

Proof. By induction. For $n = 1$, it is trivial. Assume it is true for $1, \ldots, n-1$. Let $\lambda \in \Lambda(A)$ with eigenvector x, $\|x\|_2 = 1$. Choose a unitary matrix

$$U_1 = \begin{bmatrix} x & * & * & \cdots & * \end{bmatrix} \in \mathbb{C}^{n \times n}$$

with x as the first column. Then:

$$AU_1 = \begin{bmatrix} \lambda x & * & * & \cdots & * \end{bmatrix}$$

$$U_1^H A U_1 = \begin{bmatrix} \lambda & * & \cdots & * \\ 0 & & & \\ \vdots & & \tilde{A} & \\ 0 & & & \end{bmatrix}, \qquad \text{where } \tilde{A} \in \mathbb{C}^{n-1 \times n-1}$$

Induction hypothesis:

$$\tilde{U}^H \tilde{A} \tilde{U} = \tilde{T} = \begin{bmatrix} * & * & \cdots & * \\ & * & \cdots & * \\ & & \ddots & \vdots \\ 0 & & & * \end{bmatrix}, \qquad \tilde{U} \in \mathbb{C}^{n-1 \times n-1} \text{ is unitary}$$

Set

$$U := U_1 \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & & & \\ \vdots & & \tilde{U} & \\ 0 & & & \end{bmatrix} \in \mathbb{C}^{n \times n}$$

Then U is unitary and

$$\begin{aligned} U^H A U &= \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & & & \\ \vdots & & \tilde{U}^H & \\ 0 & & & \end{bmatrix} U_1^H A U_1 \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & & & \\ \vdots & & \tilde{U} & \\ 0 & & & \end{bmatrix} \\ &= \begin{bmatrix} \lambda & * & \cdots & * \\ 0 & & & \\ \vdots & & \tilde{U}^H \tilde{A} \tilde{U} & \\ 0 & & & \end{bmatrix} =: T = \text{upper-triangular} \end{aligned}$$

□

7.2.2 Two Simple Unitary Matrices: Householder Reflectors

Householder reflectors are basic building blocks of dense matrix computation, used for QR factorization and least-squares problems as well as for the eigenvalue algorithms of this chapter; Golub and Van Loan devote a chapter to their uses. This section constructs them and settles the sign choice that keeps the construction numerically stable.

Definition 7.5. ***Householder Reflectors***
$Q \in \mathbb{C}^{n \times n}$ of the form $Q = I - 2vv^H$, where $v \in \mathbb{C}^n$ with $\|v\|_2 = 1$, is a *Householder reflector* matrix. Q is *Hermitian*:

$$Q^H = I - 2(vv^H)^H = Q.$$

Q is unitary:

$$Q^H Q = Q^2 = I - 4vv^H + 4v \underbrace{v^H v}_{=\|v\|_2^2 = 1} v^H = I.$$

For any $x \in \mathbb{C}^n$:

$Qx =$ reflection of x w.r.t. the $(n-1)$-dimensional plane $\{w \in \mathbb{C}^n \mid w^H v = 0\}$

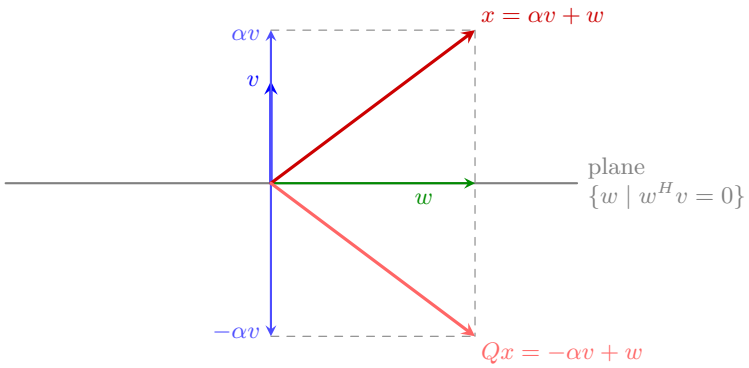


Figure 24: The Householder reflection: $x = \alpha v + w$ is mapped to $Qx = -\alpha v + w$, the mirror image of x in the plane $\{w \mid w^H v = 0\}$.

Proof.

$$\begin{aligned} x &= \alpha v + w, \qquad \text{where } \alpha \in \mathbb{C},\ w^H v = 0 \\ Qx &= (I - 2vv^H)(\alpha v + w) \\ &= \alpha v + w - 2v \underbrace{v^H \alpha v}_{=\alpha v^H v = \alpha} - 2v \underbrace{v^H w}_{=0} \\ &= -\alpha v + w \end{aligned}$$

□

Given: $x \in \mathbb{C}^n$.
We can construct a Householder reflector $Q = I - 2vv^H$ such that

$$\begin{gathered} Qx = \gamma e_1 \qquad \text{for some } \gamma \in \mathbb{C} \text{ with } |\gamma| = \|x\|_2 \\ x = \alpha v + w, \qquad w^H v = 0 \\ Qx = -\alpha v + w = \gamma e_1 \\ x - \gamma e_1 = \alpha v + w - \gamma e_1 = 2\alpha v \end{gathered}$$

Set

$$\tilde{v} := x - \gamma e_1 = \begin{bmatrix} x_1 - \gamma \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \qquad \text{and} \qquad v := \frac{\tilde{v}}{\|\tilde{v}\|_2}$$

Numerically best choice of γ:

$$\gamma := -(\operatorname{sgn} x_1)\, \|x\|_2$$

where

$$\operatorname{sgn} x_1 := \begin{cases} \dfrac{x_1}{\|x_1\|} & x \neq 0 \\ 1 & x_1 = 0 \end{cases}.$$

Summary:

$$\begin{aligned} Q &= I - 2vv^H, \qquad \text{where } v = \frac{\tilde{v}}{\|\tilde{v}\|_2} \\ \tilde{v} &= x + (\operatorname{sgn} x_1)\, \|x\|_2\, e_1 \\ &= \begin{bmatrix} x_1 + (\operatorname{sgn} x_1)\, \|x\|_2 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \\ Qx &= \gamma e_1, \qquad \text{where } \gamma = -(\operatorname{sgn} x_1)\, \|x\|_2 \end{aligned}$$

7.2.2.1 Reduction of A to Hessenberg Form

Definition 7.6. ***Hessenberg Matrix***

`http://en.wikipedia.org/wiki/Hessenberg_matrix`

An upper Hessenberg matrix has zero entries below the first subdiagonal, and a lower Hessenberg matrix has zero entries above the first superdiagonal. For example:

$$\begin{bmatrix} 1 & 4 & 2 & 3 \\ 3 & 4 & 1 & 7 \\ 0 & 2 & 3 & 4 \\ 0 & 0 & 1 & 3 \end{bmatrix} \quad \text{is upper Hessenberg.}$$

$$\begin{bmatrix} 1 & 2 & 0 & 0 \\ 5 & 2 & 3 & 0 \\ 3 & 4 & 3 & 7 \\ 5 & 6 & 1 & 1 \end{bmatrix} \quad \text{is lower Hessenberg.}$$

Schur factorization:

$$U^H A U = T = \begin{bmatrix} * & \cdots & * \\ & \ddots & \vdots \\ 0 & & * \end{bmatrix}$$

Next "best" thing:

$$U^H AU = H = \begin{bmatrix} * & * & \cdots & \cdots & * \\ * & * & & & \vdots \\ & * & \ddots & & \vdots \\ & & \ddots & \ddots & * \\ 0 & & & * & * \end{bmatrix}$$

Can be achieved with $n-2$ Householder reflections

$$U = U_1 U_2 \dots U_{n-2}, \qquad \text{where } U_j = \begin{bmatrix} I_j & 0 \\ 0 & Q_{n-j} \end{bmatrix},$$
$$Q_{n-j} \in \mathbb{C}^{(n-j)\times(n-j)} \text{ a Householder reflector.}$$

Example 7.7. $n = 5$
Let us watch the first step for $n = 5$. Partition

$$A = \begin{bmatrix} a_{11} & * \\ x & * \end{bmatrix},$$

where $x \in \mathbb{C}^4$ is the part of the first column *below* the diagonal. Build a Householder reflector $Q_1 \in \mathbb{C}^{4\times 4}$ with $Q_1 x = \gamma_1 e_1$, and embed it as $U_1 = \begin{bmatrix} 1 & 0 \\ 0 & Q_1 \end{bmatrix} = U_1^H$. The crucial design decision is the leading 1: because U_1 leaves row 1 and column 1 alone, the left multiplication compresses x onto $\gamma_1 e_1$ (zeroing entries $3, 4, 5$ of column 1), and the subsequent right multiplication touches only columns $2, \dots, 5$, so those hard-won zeros survive:

$$U_1^H AU_1 = U_1 AU_1 = (U_1 A)U_1 = \begin{bmatrix} a_{11} & * \\ \gamma_1 e_1 & * \end{bmatrix} U_1$$
$$= \begin{bmatrix} a_{11} & \# & \# & \# & \# \\ \gamma_1 & \# & \# & \# & \# \\ 0 & \# & \# & \# & \# \\ 0 & \# & \# & \# & \# \\ 0 & \# & \# & \# & \# \end{bmatrix}$$

Repeat with U_2 (leading I_2 block, Householder on the bottom 3-vector of column 2) and U_3 (leading I_3, Householder on the bottom 2-vector of column

3). After $n-2=3$ similarities the matrix is upper Hessenberg:

$$U_3^H U_2^H U_1^H A U_1 U_2 U_3 = \begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & 0 & * & * \end{bmatrix}$$

One subdiagonal remains, and it must: eliminating it too would triangularize A in finitely many operations, which we know is impossible for $n \geq 5$. The reduction stops exactly where the theory says finite algorithms must stop.

7.2.3 Filling in the Details: The Deflation Step Under a Microscope

Why does $U_1^H A U_1$ have the claimed block form? Column by column: the first column of AU_1 is $Ax = \lambda x$ (eigenpair!), so

$$U_1^H A U_1 e_1 = U_1^H (\lambda x) = \lambda\, U_1^H x = \lambda e_1,$$

since the columns of U_1 are orthonormal with first column x ($U_1^H x$ picks out the coefficients of x in that basis: $(1, 0, \ldots, 0)^T$). A first column equal to λe_1 is exactly "λ on top, zeros below": one eigenvalue is *deflated*, leaving the $(n-1)$-size block $\tilde{A}$ for the induction hypothesis.

Two ingredients the proof quietly used:

- *every* matrix has at least one eigenpair over $\mathbb{C}$ (fundamental theorem of algebra applied to p_A); this is why Schur works over $\mathbb{C}$ but real matrices need 2×2 bumps (real Schur form);
- a unit vector x extends to a unitary $U_1 = [x * \cdots *]$. Existence: Gram–Schmidt. *Stable construction*: a Householder reflector with $Qx = \gamma e_1$; then $U_1 := Q^H$ (columns orthonormal, first $= x$ up to phase). Today's two topics are one story.

The sign choice, and why it matters. We need $\tilde{v} = x - \gamma e_1$ with $|\gamma| = \|x\|_2$. Both signs of γ work in exact arithmetic; numerically, choosing $\gamma = -(\operatorname{sgn} x_1)\, \|x\|_2$ makes the first entry

$$\tilde{v}_1 = x_1 + (\operatorname{sgn} x_1)\, \|x\|_2$$

an *addition of same-signed quantities*, with no cancellation. The wrong sign would compute $x_1 - \|x\|_2\, (\operatorname{sgn} x_1)$, which for $x \approx e_1$ subtracts nearly equal

numbers and can lose all significant digits in v (the cancellation disease of Chapter 3, avoided by design).

Cost accounting: never form Q as a matrix. Apply it as

$$QM = M - 2v(v^H M),$$

one matrix–vector product + one rank-1 update: $O(n^2)$ per application instead of $O(n^3)$. Storage: just the vector v. This convention is what makes Householder-based algorithms (Hessenberg reduction, QR factorization) efficient.

7.2.4 Schur vs. Diagonalization: What We Can and Cannot Have

- **Diagonal** $U^H AU = D$ with unitary U: possible iff A is *normal* ($A^H A = AA^H$); this class includes Hermitian and unitary matrices. For Hermitian A, Schur's T is automatically diagonal with real entries: the spectral theorem falls out as a corollary ($T = U^H AU$ is both triangular and Hermitian).
- **Triangular** T: always achievable (Schur). The diagonal carries $\Lambda(A)$; the strict upper part measures non-normality.
- **Nothing better is stable:** Jordan form (last lecture) is the cautionary tale.

The computational plan for the rest of the course: reach (nearly) the Schur form by an *iteration* of cheap unitary similarities: Hessenberg first (finite, $O(n^3)$, next lecture), then QR iterations (to convergence).

7.3 Hessenberg Reduction; The QR Algorithm; Givens Rotations

This lecture assembles the modern eigenvalue algorithm. Householder similarities cannot reach triangular form in finitely many steps, because the right-hand multiplication would destroy the zeros, but they can reach upper Hessenberg form, one subdiagonal away, in about $\frac{10}{3}n^3$ operations. On Hessenberg matrices we then run the QR iteration with shifts, implementing each step with Givens rotations, plane rotations that annihilate one entry at a time. Two structural facts make the method practical: each QR step is a unitary similarity (the shift cancels exactly), and the Hessenberg form is preserved from step to step, so each iteration costs only $O(n^2)$.

7.3.1 Reduction of A to Hessenberg Form (Continued)

This section completes the Hessenberg reduction and records its cost.

$$U^H A U = H, \qquad U = U_1 U_2 \dots U_{n-2}, \qquad A \in \mathbb{C}^{n\times n}$$

Notes

1. $\Lambda(A) = \Lambda(H)$. Eigenvectors transform as follows:

$$Ax = \lambda x,\ x \neq 0 \ \Leftrightarrow\ \underbrace{U^H A U}_{H} U^H x = \lambda U^H x$$

$$y = U^H x \ \Leftrightarrow\ Hy = \lambda y,\ y \neq 0$$

   Here $y = U^H x$ and $x = Uy = U_1 U_2 \dots U_{n-2} y$,

$$U_j = \begin{bmatrix} I_j & 0 \\ 0 & Q_{n-j} \end{bmatrix}, \qquad Q_{n-j} = I - 2 v_{n-j} v_{n-j}^H, \quad v_{n-j} \in \mathbb{C}^{n-j}$$

2. Reduction of $A \in \mathbb{C}^{n\times n}$ to upper-Hessenberg form requires $\sim \frac{10}{3} n^3$ flops.

Left to do: Compute the eigenvalues of H.

7.3.2 The QR Algorithm

Published independently by Francis and by Kublanovskaya around 1961, the QR algorithm remains the standard method for the dense eigenvalue problem. This section states the shifted iteration and proves that each step is a unitary similarity of the previous iterate.

Definition 7.8. ***QR Factorization***
QR factorization of $M \in \mathbb{C}^{n\times n}$:

$$M = QR, \qquad \text{where } Q \in \mathbb{C}^{n\times n} \text{ is unitary and } R \text{ is upper-triangular}$$

This is done via Gram-Schmidt.

Algorithm 8 Shifted QR Iteration (for eigenvalue computation)

Require: upper-Hessenberg matrix $H \in \mathbb{C}^{n\times n}$

1: $A^{(0)} \leftarrow H$
2: **for** $k = 1, 2, \ldots$ **do**
3: choose a suitable shift $\mu_k \in \mathbb{C}$ ▷ e.g. $\mu_k = a_{nn}^{(k-1)}$; see the shift strategies below
4: compute a QR factorization $A^{(k-1)} - \mu_k I = Q^{(k)} R^{(k)}$ ▷ $n-1$ Givens rotations, $O(n^2)$ flops
5: $A^{(k)} \leftarrow R^{(k)} Q^{(k)} + \mu_k I$ ▷ multiply the factors in *reverse* order, add the shift back
6: **end for**

Note the shape of one step: factor, swap the factors, restore the shift. The lemma below shows that this seemingly odd recipe is a unitary similarity in disguise, so the eigenvalues are preserved at every step.

Lemma 7.9. 1. $A^{(k)} = (Q^{(k)})^H A^{(k-1)} Q^{(k)}, \quad k = 1, 2, \ldots$

2. $\Lambda(H) = \Lambda(A^{(k)}), \quad k = 0, 1, 2, \ldots$

Proof. 2. By (1), $A^{(k)}, A^{(k-1)}, A^{(k-2)}, \ldots, A^{(1)}, A^{(0)} = H$ are all similar.

1.
$$A^{(k)} = R^{(k)} Q^{(k)} + \mu_k I = (Q^{(k)})^H (\underbrace{Q^{(k)} R^{(k)} + \mu_k I}_{=A^{(k-1)}}) Q^{(k)}$$

□

Note: All the matrices $A^{(k)}$ are upper-Hessenberg!

7.3.3 Two Simple Unitary Matrices: Givens Rotations

Householder reflectors modify many entries at once; Givens rotations act on two rows and can zero a single chosen entry. For Hessenberg matrices, whose subdiagonal has only $n-1$ entries, they are the right tool. This section defines them and applies them to the QR step.

7.3.3.1 2×2 Case

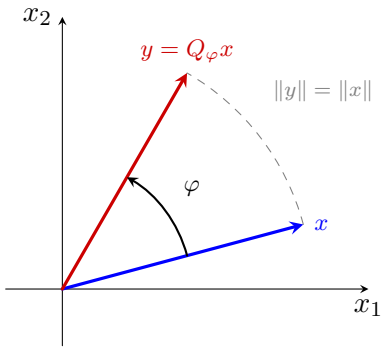


Figure 25: A Givens rotation maps x to $y = Q_\varphi x$, preserving length.

$$Q = \begin{bmatrix} c & s \\ -\overline{s} & \overline{c} \end{bmatrix} \in \mathbb{C}^{2\times 2}, \qquad \text{where } |c|^2 + |s|^2 = 1$$

Q is unitary:

$$Q^H Q = \begin{bmatrix} \overline{c} & -s \\ \overline{s} & c \end{bmatrix} \begin{bmatrix} c & s \\ -\overline{s} & \overline{c} \end{bmatrix} = \begin{bmatrix} |c|^2 + |s|^2 & 0 \\ 0 & |c|^2 + |s|^2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Householder:

$$Q \begin{bmatrix} * \\ * \\ \vdots \\ * \end{bmatrix} = \begin{bmatrix} * \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

Givens:

$$Q \begin{bmatrix} * \\ * \end{bmatrix} = \begin{bmatrix} * \\ 0 \end{bmatrix}$$

Use of Q: to "zero out" the entry x_2 of any given $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \in \mathbb{C}^2, \quad x \neq 0.$

Indeed: Set

$$c = e^{i\alpha} \frac{\overline{x_1}}{\|x\|_2}, \qquad s = e^{i\alpha} \frac{\overline{x_2}}{\|x\|_2}, \qquad \text{where } \alpha \in \mathbb{C}$$

Then

$$|c|^2 + |s|^2 = \frac{|x_1|^2 + |x_2|^2}{\|x\|_2^2} = 1$$

$$Q \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \frac{1}{\|x\|_2} \begin{bmatrix} e^{i\alpha}\overline{x_1} & e^{i\alpha}\overline{x_2} \\ -e^{-i\alpha}x_2 & e^{-i\alpha}x_1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} e^{i\alpha}\|x\|_2 \\ 0 \end{bmatrix}$$

7.3.3.2 General Case

$$Q = \begin{bmatrix} 1 & & & & & & & & 0 \\ & \ddots & & & & & & & \\ & & 1 & & & & & & \\ & & & c & & & s & & \\ & & & & 1 & & & & \\ & & & & & \ddots & & & \\ & & & -\overline{s} & & & \overline{c} & & \\ & & & & & & & 1 & \\ & & & & & & & & \ddots \\ 0 & & & & & & & & 1 \end{bmatrix}$$

Notes:

1. Left-multiplication with Q only changes the jth and kth rows.
2. Right-multiplication with Q only changes the jth and kth columns.
3. Q is a Givens rotation $\Rightarrow Q^H$ is a Givens rotation:

$$Q^H = \begin{bmatrix} \overline{c} & -s \\ \overline{s} & c \end{bmatrix}$$

7.3.3.3 Use in QR Algorithm

Example 7.10. $n = 5$

Here is one QR factorization inside the iteration, carried out with Givens rotations for $n = 5$. The matrix to factor is Hessenberg, so only the four subdiagonal entries stand between it and triangular form:

$$A^{(k-1)} - \mu_k I = \begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & 0 & * & * \end{bmatrix}$$

We eliminate them one at a time, top to bottom. The first rotation acts on rows 1 and 2 and is tuned to zero the $(2, 1)$ entry: choose

$$Q_1 = \begin{bmatrix} c_1 & s_1 & & & \\ -\overline{s_1} & \overline{c_1} & & & \\ & & 1 & & \\ & & & 1 & \\ & & & & 1 \end{bmatrix} \text{ such that}$$

$$Q_1^H(A^{(k-1)} - \mu_k I) = \begin{bmatrix} \# & \# & \# & \# & \# \\ 0 & \# & \# & \# & \# \\ 0 & * & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & 0 & * & * \end{bmatrix}$$

$$\text{Choose } Q_2 = \begin{bmatrix} 1 & & & & \\ & c_2 & s_2 & & \\ & -\overline{s_2} & \overline{c_2} & & \\ & & & 1 & \\ & & & & 1 \end{bmatrix} \text{ such that}$$

$$Q_2^H Q_1^H(A^{(k-1)} - \mu_k I) = \begin{bmatrix} * & * & * & * & * \\ 0 & \# & \# & \# & \# \\ 0 & 0 & \# & \# & \# \\ 0 & 0 & * & * & * \\ 0 & 0 & 0 & * & * \end{bmatrix}$$

Note that Q_2 mixes only rows 2 and 3: it zeros the $(3,2)$ entry without disturbing the zero just created in column 1. Two more rotations, on rows $(3,4)$ and $(4,5)$, finish the job:

$$\vdots$$

$$Q_4^H Q_3^H Q_2^H Q_1^H(A^{(k-1)} - \mu_k I) = \begin{bmatrix} * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & 0 & \# & \# \\ 0 & 0 & 0 & 0 & \# \end{bmatrix} =: R^{(k)}$$

Four rotations, each touching two rows of length at most n: the whole factorization costs $O(n^2)$, against $O(n^3)$ for a full matrix. This is the payoff of the Hessenberg preprocessing, and (as shown in the "Filling in the Details" section) the form is restored when $R^{(k)}$ is multiplied back by the rotations, so the same cheap step repeats forever.

7.3.4 Filling in the Details: Why Hessenberg and Not Triangular?

Tempting: use a Householder Q_0 on *all* of column 1 to zero entries $2, \ldots, n$; would $n-1$ such steps not triangularize A? The catch is the **similarity**: we must multiply by U_1^H on the left *and* U_1 on the right. If U_1 mixes row 1 with the others, the right-multiplication mixes *column* 1 correspondingly and refills the zeros we just created. (And it must fail: a finite exact triangularization would compute eigenvalues in finitely many operations, impossible for $n \geq 5$.)

The fix: let $U_1 = \begin{bmatrix} 1 & 0 \\ 0 & Q_1 \end{bmatrix}$ act only on rows/columns $2, \ldots, n$. Left multiplication zeros column-1 entries $3, \ldots, n$ (using $x = A_{2:n,1}$); right multiplication touches only columns $2, \ldots, n$, so *column 1's zeros survive*. Iterating on columns $2, \ldots, n-2$ leaves exactly one subdiagonal: upper-Hessenberg, reached in finitely many ($\sim \frac{10}{3}n^3$ flops) steps. Hessenberg is the closest to triangular that a finite unitary similarity process can reach.

Why the QR iteration preserves Hessenberg + why Givens fits. For Hessenberg $A^{(k-1)} - \mu_k I$: only one subdiagonal to annihilate, and Givens rotation Q_j (acting on rows $j, j{+}1$) kills the single entry $(j{+}1, j)$ without touching anything below; $n-1$ rotations give $R^{(k)}$ in $O(n^2)$ flops (vs $O(n^3)$ for a full matrix!). Then

$$A^{(k)} = R^{(k)}\, Q_1 Q_2 \cdots Q_{n-1} + \mu_k I :$$

right-multiplying the triangular $R^{(k)}$ by each Q_j mixes columns $j, j{+}1$, adding back exactly one subdiagonal, so $A^{(k)}$ is Hessenberg again. The structure is a fixed point of the iteration, so *every* QR step costs only $O(n^2)$. Householder for the one-time reduction, Givens for the surgical strikes inside the loop: each tool where it is cheapest.

The similarity identity, re-derived: from $A^{(k-1)} - \mu_k I = Q^{(k)} R^{(k)}$ we get $R^{(k)} = (Q^{(k)})^H (A^{(k-1)} - \mu_k I)$, hence

$$\begin{aligned} A^{(k)} = R^{(k)} Q^{(k)} + \mu_k I = (Q^{(k)})^H A^{(k-1)} Q^{(k)} - \mu_k (Q^{(k)})^H Q^{(k)} + \mu_k I \\ = (Q^{(k)})^H A^{(k-1)} Q^{(k)}. \end{aligned}$$

The shift cancels: every QR step is a *unitary similarity*, so $\Lambda(A^{(k)}) = \Lambda(H) = \Lambda(A)$ for all k; we can iterate forever without corrupting the answer.

7.3.5 Cost Sheet for the Eigenvalue Pipeline

Stage	flops
$A \to H$ (Householder, once)	$\sim \frac{10}{3}n^3$
one QR step on Hessenberg (Givens)	$O(n^2)$
QR steps needed in practice	$O(n)$ total ($\approx$ 2–3 per eigenvalue)
eigenvalues only	$\sim 10n^3$
+ eigenvectors (accumulate U)	$\sim 27n^3$

Without the Hessenberg reduction, each QR step would cost $O(n^3)$ and the whole algorithm $O(n^4)$; the preprocessing pays for itself immediately. Why so few steps per eigenvalue? Because of *shifts*, the convergence theory of the final lecture.

7.4 Convergence of the QR Algorithm; Shift Strategies

The final substantive lecture explains why the QR algorithm converges and how it is used. The unshifted iteration is simultaneous power iteration in disguise: subdiagonal entries decay geometrically with ratios of adjacent eigenvalue moduli, which is unacceptably slow when eigenvalues are close. Shifting repairs this: choosing the shift near an eigenvalue turns the relevant ratio into a rapidly shrinking one, and the standard shift strategies achieve quadratic (for Hermitian matrices, cubic) convergence of the trailing subdiagonal entry. Deflation then splits converged eigenvalues off the problem. Assembled with the Hessenberg reduction, this is, in essence, the algorithm inside production eigenvalue libraries.

7.4.1 End-of-Semester Notes

- Final project presentations: last week of classes / final examination period (Dec. 14–18); written reports due during finals week.
- Project topics must be approved by the instructor.

7.4.2 QR Factorization (Continued)

We assemble the complete algorithm for a general matrix.

For general $n \geq 2$
Convert A to a Hessenberg matrix and feed this to the QR algorithm: $A \to H = A^{(0)}$.

$$A^{(k-1)} - \mu_k I = Q^{(k)} R^{(k)} \qquad \text{where } Q^{(k)} = \underbrace{Q_1 Q_2 \dots Q_{n-1}}_{n-1 \text{ Givens rotations}}$$

$$A^{(k)} = R^{(k)} Q^{(k)} + \mu_k I \text{ is upper-Hessenberg}$$

7.4.3 Convergence of the QR Algorithm

Why the iteration converges, and how fast, is the subject of this section. The complete theory is beyond the course; we state the essential result for the unshifted iteration and explain its mechanism, the connection to the power method, which Trefethen and Bau develop in detail.

Let $\Lambda(A) = \{\lambda_1, \lambda_2, \dots, \lambda_n\}$ and assume that $|\lambda_1| > |\lambda_2| > \cdots > |\lambda_n|$. (Special case: $\mu_k = 0$ for all k.) Then

$$A^{(k)} = \begin{bmatrix} * & * & \cdots & \cdots & * \\ a_{21}^{(k)} & * & \cdots & \cdots & * \\ & a_{32}^{(k)} & \ddots & & \vdots \\ & & \ddots & \ddots & \vdots \\ 0 & & & a_{n,n-1}^{(k)} & * \end{bmatrix} \xrightarrow{k\to\infty} R = \begin{bmatrix} r_{11} & * & \cdots & \cdots & * \\ 0 & r_{22} & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \ddots & * \\ 0 & \cdots & \cdots & 0 & r_{nn} \end{bmatrix}$$

where $\Lambda(R) = \{r_{11}, r_{22}, \dots, r_{nn}\} = \Lambda(A)$.

Speed of convergence: If $|\lambda_n| > 0$, then

$$a_{j,j-1}^{(k)} = O\left(\left| \frac{\lambda_j}{\lambda_{j-1}} \right|^k \right), \qquad j = 2, 3, \dots, \quad k = 1, 2, \dots$$

$\Rightarrow$ slow convergence if $|\lambda_j| \lessapprox |\lambda_{j-1}|$. Shifts μ_k are used to speed up convergence. Suppose $\mu_k = \mu =$ constant and $|\lambda_1 - \mu| > |\lambda_2 - \mu| > \cdots > |\lambda_n - \mu| > 0$. Speed of convergence:

$$a_{j,j-1}^{(k)} = O\left(\left| \frac{\lambda_j - \mu}{\lambda_{j-1} - \mu} \right|^k \right), \qquad j = 2, 3, \dots, n, \quad k = 1, 2, \dots$$

$\Rightarrow a_{j,j-1}^{(k)} \xrightarrow{k\to\infty} 0$ fast if $\mu \approx \lambda_j$.

7.4.4 Strategy for Choosing μ_k

Shifts convert the slow linear convergence of the previous section into fast local convergence. This section presents the standard strategy, the deflation test, and the cost accounting for the assembled algorithm. Wilkinson's book contains the full analysis of the shift strategies.

At the beginning of the kth iteration of the QR algorithm,

$$A^{(k-1)} = [a_{ij}^{(k-1)}] = \begin{bmatrix} * & \cdots & \cdots & \cdots & * \\ a_{21}^{(k)} & * & & & \vdots \\ 0 & \ddots & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & a_{n,n-1}^{(k)} & * \end{bmatrix};$$

for $j = 2, 3, \ldots, n$, check if $a_{j,j-1}^{(k-1)} \approx 0$.

$$\begin{matrix} a_{j-1,j-1}^{(k)} & \\ a_{j,j-1}^{(k-1)} & a_{jj}^{(k)} \end{matrix}$$

$$\left|a_{j,j-1}^{k-1}\right| \leq \epsilon \left|a_{j-1,j-1}^{(k-1)}\right| + \epsilon \left|a_{jj}^{(k-1)}\right|, \qquad \text{where } \epsilon = O(\texttt{eps})$$

Set $a_{j,j-1}^{(k-1)} = 0$.

7.4.4.1 3 Cases

1. Case 1: $j = n$

$$A^{(k-1)} = \begin{bmatrix} * & * & \cdots & \cdots & * & * \\ * & \ddots & & & \vdots & \vdots \\ & \ddots & \ddots & & \vdots & \vdots \\ & & \ddots & \ddots & * & \vdots \\ & & & * & * & * \\ 0 & \cdots & \cdots & 0 & 0 & a_{nn}^{(k-1)} \end{bmatrix} = \begin{bmatrix} \tilde{A}^{(k-1)} & * \\ 0 & a_{nn}^{(k-1)} \end{bmatrix},$$

where $\tilde{A}^{(k-1)} \in \mathbb{C}^{(n-1)\times(n-1)}$. $\Rightarrow a_{nn}^{(k-1)} \in \Lambda(A^{(k-1)}) = \Lambda(A)$.

Accept $a_{nn}^{(k-1)}$ as an eigenvalue of A and continue the QR algorithm on $\tilde{A}^{(k-1)}$. Typical choice: $\mu_k = a_{n-1,n-1}^{(k-1)}$.

2. Case 2: $j = 2$

$$A^{(k-1)} = \begin{bmatrix} a_{11}^{(k-1)} & * & \cdots & \cdots & * \\ 0 & & & & \\ 0 & & & & \\ \vdots & & \tilde{A}^{(k-1)} & & \\ 0 & & & & \end{bmatrix}$$

$a_{11}^{(k-1)}$ is an eigenvalue of A.

3. Case 3: $2 < j < n$

$$A^{(k-1)} = \begin{bmatrix} * & * & \cdots & \cdots & * & * & \cdots & \cdots & \cdots & * \\ * & \ddots & & & \vdots & \vdots & & & & \vdots \\ & \ddots & \ddots & & \vdots & \vdots & & & & \vdots \\ & & \ddots & \ddots & * & \vdots & & & & \vdots \\ & & & * & * & * & \cdots & \cdots & \cdots & * \\ & & & & 0 & * & * & \cdots & \cdots & * \\ & & & & & * & \ddots & & & \vdots \\ & & & & 0 & & \ddots & \ddots & & \vdots \\ & & & & & & & \ddots & \ddots & * \\ & & & & 0 & & & & * & * \end{bmatrix} = \begin{bmatrix} A_1^{(k-1)} & * \\ 0 & A_2^{(k-1)} \end{bmatrix}$$

$\Lambda(A) = \Lambda(A^{(k-1)}) = \Lambda(A_1^{(k-1)}) \cup \Lambda(A_2^{(k-1)})$. Continue QR on $A_1^{(k-1)} \in \mathbb{C}^{j-1 \times j-1}$ and $A_2^{(k-1)} \in \mathbb{C}^{n-j+1 \times n-j+1}$.

Typical flop count for such a practical QR algorithm:

- $\sim 10n^3$ if only the eigenvalues are computed
- $\sim 27n^3$ if eigenvalues and eigenvectors are computed

7.4.5 Filling in the Details: Where the Ratios Come From

Unshifted QR is *power iteration in disguise*. Sketch for intuition: the first column of $Q^{(1)} \cdots Q^{(k)}$ equals the normalized $A^k e_1$, the classical power method, converging to the dominant eigenvector at rate $|\lambda_2/\lambda_1|$. Simultaneously, the QR iteration runs power iteration on *all* leading subspaces $\text{span}(e_1, \ldots, e_j)$ at once; the jth subdiagonal entry measures how far the jth

subspace still is from invariant, decaying like $|\lambda_j/\lambda_{j-1}|^k$. Close eigenvalue moduli $\Rightarrow$ ratio $\approx 1 \Rightarrow$ stagnation. (It also explains why the eigenvalues appear *sorted* by modulus on the limit diagonal.)

Shifts as inverse iteration. The identity $(A-\mu I)^{-1}$ has eigenvalues $(\lambda_j-\mu)^{-1}$: whichever λ_j is closest to μ becomes strongly dominant. The shifted QR step secretly performs one step of inverse iteration with shift μ_k on the *last* row; with $\mu_k = a_{nn}^{(k-1)}$ (a Rayleigh-quotient-like estimate of λ_n) the relevant ratio

$$\left|\frac{\lambda_n-\mu_k}{\lambda_{n-1}-\mu_k}\right|$$

shrinks at every step; the convergence of $a_{n,n-1}^{(k)} \to 0$ becomes *quadratic* (cubic for Hermitian A). Two or three iterations per eigenvalue in practice, matching the cost sheet.

Deflation is what makes it an algorithm. The stopping test

$$\left|a_{j,j-1}^{(k-1)}\right| \le \epsilon\Big(\left|a_{j-1,j-1}^{(k-1)}\right| + \left|a_{jj}^{(k-1)}\right|\Big), \qquad \epsilon = O(\epsilon_{\text{mach}}),$$

compares each subdiagonal to its diagonal *neighbors* (a relative, scale-invariant test). Setting such an entry to zero is a backward perturbation of size $O(\epsilon_{\text{mach}}\,\|A\|)$, which is harmless, and makes the matrix block triangular:

$$\Lambda\begin{bmatrix} A_1 & * \\ 0 & A_2 \end{bmatrix} = \Lambda(A_1)\cup\Lambda(A_2).$$

Case $j=n$: accept a_{nn} as an eigenvalue, continue on the leading $(n-1)$-block. Case $2<j<n$: recurse on two smaller Hessenberg problems. The problem literally falls apart as it converges.

The assembled "Francis" algorithm (what `eig` runs): balance $\to$ Householder Hessenberg $\to$ shifted QR with Givens + deflation $\to$ (optionally) accumulate transformations for eigenvectors. Backward stable: the computed Λ is exact for some $A+E$, $\|E\| = O(\epsilon_{\text{mach}}\,\|A\|)$: the eigenvalue story ends where the course began, with backward error analysis certifying the answer.

7.4.6 Course Capstone: One Idea Three Times

Notice the same design pattern in our three big algorithms:

1. **LU with pivoting:** control element growth ($|l_{jk}| \le 1$) $\Rightarrow$ backward stable linear solves.

2. **B-spline evaluation:** convex combinations of nonnegative quantities $\Rightarrow$ no cancellation, perfectly stable.

3. **QR eigenvalue algorithm:** unitary transformations only ($\kappa_2 = 1$) $\Rightarrow$ errors never amplified.

In each case, *the algorithm is organized so that no intermediate quantity can dwarf the data*; this is the single most important design principle this course has to offer. Next lecture: full review.

7.5 Notes and References

Golub and Van Loan is the standard reference for the material of this chapter, from the Schur decomposition through the practical shifted QR algorithm; Trefethen and Bau presents the same pipeline with an emphasis on the connection to power iteration used in the convergence discussion. The QR algorithm is due to Francis and Kublanovskaya; Wilkinson's *The Algebraic Eigenvalue Problem* contains the classical analysis, including the shift strategy bearing his name.

7.6 Exercises

1. Expand $\det(zI - A)$ for the 4×4 companion matrix of $p(z) = z^4 + \alpha_3 z^3 + \alpha_2 z^2 + \alpha_1 z + \alpha_0$ and verify that it equals $p(z)$.

2. Show that similar matrices have the same characteristic polynomial, and hence the same eigenvalues with the same algebraic multiplicities. Give an example showing that the converse fails: two matrices with the same characteristic polynomial that are not similar.

3. For $x = (3, 4)^T$, construct the Householder reflector Q with $Qx = \gamma e_1$, using the sign choice of the text. Write out Q as a 2×2 matrix, and verify $Q = Q^H$, $Q^H Q = I$, and $|\gamma| = \|x\|_2$.

4. Let $Q = I - 2vv^H$ with $\|v\|_2 = 1$. Show that $Qv = -v$ and $Qw = w$ for every w orthogonal to v, and interpret Q geometrically as a reflection. Conclude that $\Lambda(Q) = \{-1, 1, \ldots, 1\}$.

5. Reduce $A = \begin{bmatrix} 2 & 1 & 1 \\ 1 & 3 & 1 \\ 1 & 1 & 4 \end{bmatrix}$ to tridiagonal (Hessenberg) form with one Householder similarity acting on rows and columns $2, 3$. Why does symmetry make the result tridiagonal rather than merely Hessenberg?

6. Perform one shifted QR step on $A = \begin{bmatrix} 2 & \epsilon \\ \epsilon & 1 \end{bmatrix}$ with the shift $\mu = a_{22} = 1$, for small $\epsilon > 0$. Show that the new off-diagonal entry is $O(\epsilon^3)$, illustrating the cubic convergence of the shifted iteration for Hermitian matrices.

7. Prove the spectral theorem as a corollary of Schur's theorem: if $A = A^H$, then the Schur factor $T = U^H A U$ is diagonal with real entries, so A has an orthonormal basis of eigenvectors.

8. (Computing.) Implement the unshifted QR iteration for the symmetric matrix with eigenvalues $4, 2, 1$ obtained as $A = Q\,\mathrm{diag}(4,2,1)\,Q^T$ for a random orthogonal Q. Plot $\left|a_{32}^{(k)}\right|$ and $\left|a_{21}^{(k)}\right|$ against k on a log scale, and compare the slopes with $|\lambda_3/\lambda_2|$ and $|\lambda_2/\lambda_1|$.

Index