Lecture Notes on Numerical Analysis

Virginia Tech · MATH/CS 5466 · Spring 2016

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

This course tells a marvelous success story. Through the use of clever algorithms, careful analysis, and speedy computers, we can construct *approximate* solutions to these otherwise intractable problems with remarkable speed. Nick Trefethen defines *numerical analysis* to be 'the study of algorithms for the problems of continuous mathematics'. This course takes a tour through many such algorithms, sampling a variety of techniques suitable across many applications. We aim to assess alternative methods based on both accuracy and efficiency, to discern well-posed problems from ill-posed ones, and to see these methods in action through computer implementation.

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

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

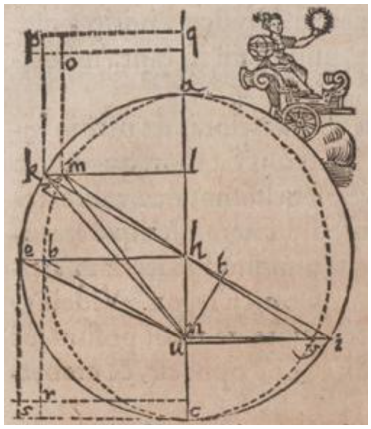


Image from Johannes Kepler's *Astronomia nova*, 1609, (ETH Bibliothek). In this text Kepler derives his famous equation that solves two-body orbital motion,

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

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

We highly recommend Trefethen's essay, 'The Definition of Numerical Analysis', (reprinted on pages 321–327 of Trefethen & Bau, *Numerical Linear Algebra*), which inspires our present manifesto.

learn how to evaluate algorithms according to these criteria.

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

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

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

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

Matrix computations play a fundamental role in numerical analysis. Discretization of continuous variables turns calculus into algebra. Algorithms for the fundamental problems in linear algebra are covered in MATH/CS 5465. If you have missed this beautiful content, your life will be poorer for it; when the methods we discuss this semester connect to matrix techniques, we will provide pointers.

These lecture notes were developed alongside courses that were supported by textbooks, such as *An Introduction to Numerical Analysis* by Süli and Mayers, *Numerical Analysis* by Gautschi, and *Numerical Analysis* by Kincaid and Cheney. These notes have benefited from this pedigree, and reflect certain hallmarks of these books. We have also been significantly influenced by G. W. Stewart's inspiring volumes, *Afternotes on Numerical Analysis* and *Afternotes Goes to Graduate School*. I am grateful for comments and corrections from past students, and welcome suggestions for further repair and amendment.

— Mark Embree

1
Interpolation

LECTURE 1: *Polynomial Interpolation in the Monomial Basis*

AMONG THE MOST FUNDAMENTAL problems in numerical analysis is the construction of a polynomial that approximates a continuous real function $f : [a, b] \to \mathbb{R}$. Of the several ways we might design such polynomials, we begin with *interpolation*: we will construct polynomials that exactly match f at certain fixed points in the interval $[a, b] \subset \mathbb{R}$.

1.1 *Polynomial interpolation: definitions and notation*

Definition 1.1. The set of continuous functions that map $[a, b] \subset \mathbb{R}$ to $\mathbb{R}$ is denoted by $C[a, b]$. The set of continuous functions whose first r derivatives are also continuous on $[a, b]$ is denoted by $C^r[a, b]$. (Note that $C^0[a, b] \equiv C[a, b]$.)

Definition 1.2. The set of polynomials of degree n or less is denoted by $\mathcal{P}_n$.

Note that $C[a, b]$, $C^r[a, b]$ (for any $a < b$, $r \geq 0$) and $\mathcal{P}_n$ are *linear spaces* of functions (since linear combinations of such functions maintain continuity and polynomial degree). Furthermore, note that $\mathcal{P}_n$ is an $n + 1$ dimensional subspace of $C[a, b]$.

The polynomial interpolation problem can be stated as:

> Given $f \in C[a, b]$ and $n + 1$ points $\{x_j\}_{j=0}^n$ satisfying
>
> $$a \leq x_0 < x_1 < \cdots < x_n \leq b,$$
>
> determine some $p_n \in \mathcal{P}_n$ such that
>
> $$p_n(x_j) = f(x_j) \qquad \text{for } j = 0, \ldots, n.$$

We freely use the concept of *vector spaces*. A set of functions $\mathcal{V}$ is a real vector space it is closed under vector addition and multiplication by a real number: for any $f, g \in \mathcal{V}$, $f + g \in \mathcal{V}$, and for any $f \in \mathcal{V}$ and $\alpha \in \mathbb{R}$, $\alpha f \in \mathcal{V}$. For more details, consult a text on linear algebra or functional analysis.

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

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

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

Regarding this last question, we should note that, in practice, we are not always able to choose the interpolation points as freely as we might like. For example, our 'continuous function $f \in C[a,b]$' could actually be a discrete list of previously collected experimental data, and we are stuck with the values $\{x_j\}_{j=0}^n$ at which the data was measured.

1.2 *Constructing interpolants in the monomial basis*

Of course, any polynomial $p_n \in \mathcal{P}_n$ can be written in the form

$$p_n(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_n x^n$$

for coefficients $c_0, c_1, \ldots, c_n$. We can view this formula as an expression for p_n as a linear combination of the *basis functions* 1, x, x^2, $\ldots$, x^n; these basis functions are called *monomials*.

To construct the polynomial interpolant to f, we merely need to determine the proper values for the coefficients $c_0, c_1, \ldots, c_n$ in the above expansion. The interpolation conditions $p_n(x_j) = f(x_j)$ for

$j = 0, \ldots, n$ reduce to the equations

$$\begin{aligned} c_0 + c_1 x_0 + c_2 x_0^2 + \cdots + c_n x_0^n &= f(x_0) \\ c_0 + c_1 x_1 + c_2 x_1^2 + \cdots + c_n x_1^n &= f(x_1) \\ &\vdots \\ c_0 + c_1 x_n + c_2 x_n^2 + \cdots + c_n x_n^n &= f(x_n). \end{aligned}$$

Note that these $n+1$ equations are linear in the $n+1$ unknown parameters $c_0, \ldots, c_n$. Thus, our problem of finding the coefficients $c_0, \ldots, c_n$ reduces to solving the linear system

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ 1 & x_2 & x_2^2 & \cdots & x_2^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} f(x_0) \\ f(x_1) \\ f(x_2) \\ \vdots \\ f(x_n) \end{bmatrix}, \tag{1.1}$$

which we denote as $\mathbf{Ac} = \mathbf{f}$. Matrices of this form, called *Vandermonde* matrices, arise in a wide range of applications.[1] Provided all the interpolation points $\{x_j\}$ are distinct, one can show that this matrix is invertible.[2] Hence, fundamental properties of linear algebra allow us to confirm that there is exactly one degree-n polynomial that interpolates f at the given $n+1$ distinct interpolation points.

Theorem 1.1. Given $f \in C[a,b]$ and distinct points $\{x_j\}_{j=0}^n$, $a \leq x_0 < x_1 < \cdots < x_n \leq b$, there exists a unique $p_n \in \mathcal{P}_n$ such that $p_n(x_j) = f(x_j)$ for $j = 0, 1, \ldots, n$.

To determine the coefficients $\{c_j\}$, we could solve the above linear system with the Vandermonde matrix using some variant of Gaussian elimination (e.g., using MATLAB's \ command); this will take $\mathcal{O}(n^3)$ floating point operations. Alternatively, we could (and should) use a specialized algorithm that exploit the Vandermonde structure to determine the coefficients $\{c_j\}$ in only $\mathcal{O}(n^2)$ operations, a vast improvement.[3]

1.2.1 *Potential pitfalls of the monomial basis*

Though it is straightforward to see how to construct interpolating polynomials in the monomial basis, this procedure can give rise to some unpleasant numerical problems when we actually attempt to determine the coefficients $\{c_j\}$ on a computer. The primary difficulty is that the monomial basis functions 1, x, x^2, $\ldots$, x^n look increasingly alike as we take higher and higher powers. Figure 1.1 illustrates this behavior on the interval $[a,b] = [0,1]$ with $n = 5$ and $x_j = j/5$.

[1] Higham presents many interesting properties of Vandermonde matrices and algorithms for solving Vandermonde systems in Chapter 21 of *Accuracy and Stability of Numerical Algorithms*, 2nd ed., (SIAM, 2002). Vandermonde matrices arise often enough that MATLAB has a built-in command for creating them. If $\mathtt{x} = [x_0, \ldots, x_n]^T$, then $\mathbf{A} = \mathtt{fliplr(vander(x))}$.

[2] In fact, the determinant takes the simple form

$$\det(\mathbf{A}) = \prod_{j=0}^{n} \prod_{k=j+1}^{n} (x_k - x_j).$$

This is evident for $n = 1$; we will not prove if for general n, as we will have more elegant ways to establish existence and uniqueness of polynomial interpolants. For a clever proof, see p. 193 of Bellman, *Introduction to Matrix Analysis*, 2nd ed., (McGraw-Hill, 1970).

[3] See Higham's book for details and stability analysis of specialized Vandermonde algorithms.

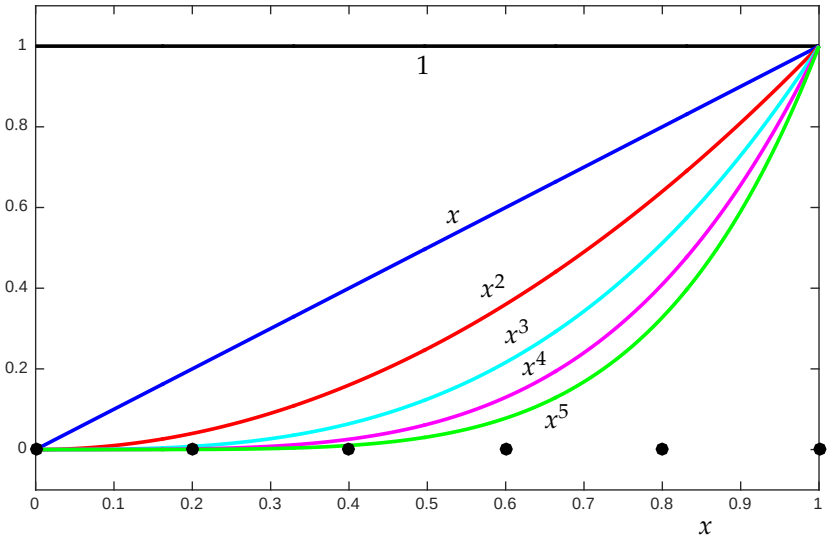


Figure 1.1: The six monomial basis vectors for $\mathcal{P}_5$, based on the interval $[a, b] = [0, 1]$ with $x_j = j/5$ (red circles). Note that the basis vectors increasingly align as the power increases: this basis becomes *ill-conditioned* as the degree of the interpolant grows.

Because these basis vectors become increasingly alike, one finds that the expansion coefficients $\{c_j\}$ in the monomial basis can become very large in magnitude even if the function $f(x)$ remains of modest size on $[a, b]$.

Consider the following analogy from linear algebra. The vectors

$$\begin{bmatrix} 1 \\ 10^{-10} \end{bmatrix}, \qquad \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

form a basis for $\mathbb{R}^2$. However, both vectors point in *nearly* the same direction, though of course they are *linearly independent*. We can write the vector $[1, 1]^T$ as a unique linear combination of these basis vectors:

$$\begin{bmatrix} 1 \\ 1 \end{bmatrix} = 10,000,000,000 \begin{bmatrix} 1 \\ 10^{-10} \end{bmatrix} - 9,999,999,999 \begin{bmatrix} 1 \\ 0 \end{bmatrix}. \tag{1.2}$$

Although the vector we are expanding and the basis vectors themselves are all have modest size (norm), the expansion coefficients are enormous. Furthermore, small changes to the vector we are expanding will lead to huge changes in the expansion coefficients. This is a recipe for disaster when computing with finite-precision arithmetic.

This same phenomenon can occur when we express polynomials in the monomial basis. As a simple example, consider interpolating $f(x) = 2x + x\sin(40x)$ at uniformly spaced points ($x_j = j/n$, $j = 0, \ldots, n$) in the interval $[0, 1]$. Note that $f \in C^\infty[0, 1]$: this f is a 'nice' function with infinitely many continuous derivatives. As seen in Figures 1.2–1.3, f oscillates modestly on the interval $[0, 1]$, but it

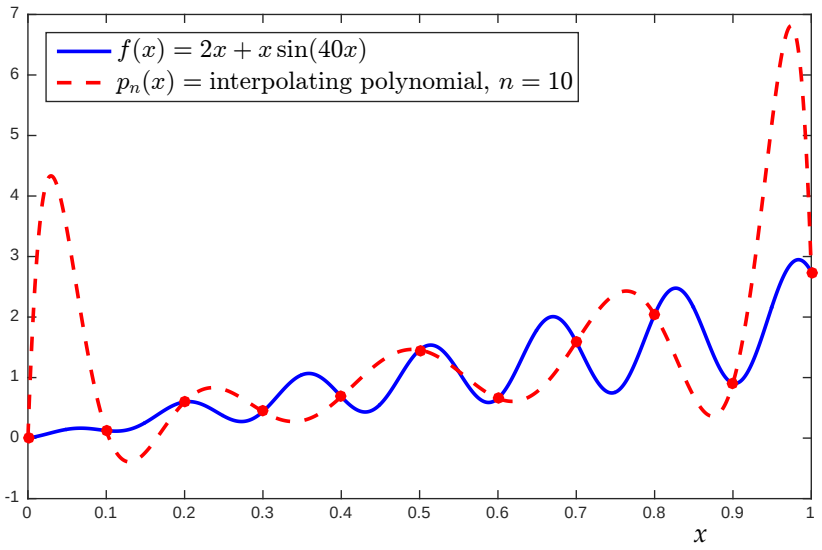


Figure 1.2: Degree $n = 10$ interpolant $p_{10}(x)$ to $f(x) = 2x + x\sin(40x)$ at the uniformly spaced points $x_0, \ldots, x_{10}$ for $x_j = j/10$ over $[a, b] = [0, 1]$. Even though $p_{10}(x_j) = f(x_j)$ at the eleven points $x_0, \ldots, x_n$ (red circles), the interpolant gives a poor approximation to f at the ends of the interval.

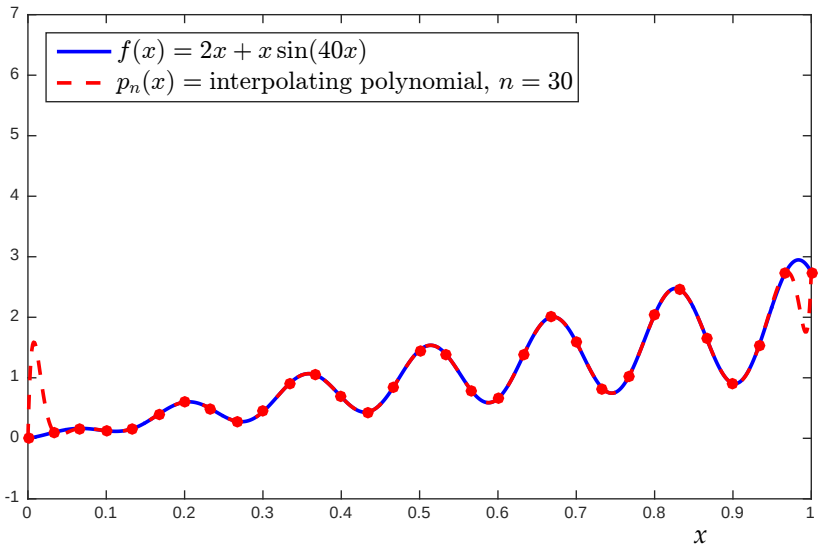


Figure 1.3: Repetition of Figure 1.2, but now with the degree $n = 30$ interpolant at uniformly spaced points $x_j = j/30$ on $[0, 1]$. The polynomial still overshoots f near $x = 0$ and $x = 1$, though by less than for $n = 10$; for this example, the overshoot goes away as n is increased further.

certainly does not grow excessively large in magnitude or exhibit any nasty singularities.

Comparing the interpolants with $n = 10$ and $n = 30$ between the two figures, it appears that, in some sense, $p_n \to f$ as n increases. Indeed, this is the case, in a manner we shall make precise in future lectures.

However, we must address a crucial question:

> *Can we accurately compute the coefficients $c_0, \ldots, c_n$ that specify the interpolating polynomial?*

Use MATLAB's basic Gaussian elimination algorithm to solve the Vandermonde system $\mathbf{A}\mathbf{c} = \mathbf{f}$ for $\mathbf{c}$ via the command `c = A\f`, then evaluate

$$p_n(x) = \sum_{j=0}^{n} c_j x^j$$

e.g., using MATLAB's `polyval` command.

Since p_n was constructed to interpolate f at the points $x_0, \ldots, x_n$, we might (at the very least!) expect

$$f(x_j) - p_n(x_j) = 0, \qquad j = 0, \ldots, n.$$

Since we are dealing with numerical computations with a finite precision floating point system, we should instead be well satisfied if our numerical computations only achieve $|f(x_j) - p_n(x_j)| = \mathcal{O}(\varepsilon_{\text{mach}})$, where $\varepsilon_{\text{mach}}$ denotes the precision of the floating point arithmetic system.[4]

More precisely, we might expect

$$|f(x_j) - p_n(x_j)| \approx \varepsilon_{\text{mach}} \|f\|_{L_\infty},$$

where $\|f\|_{L_\infty} := \max_{x \in [a,b]} |f(x)|$.

Instead, the results of our numerical computations are *remarkably inaccurate* due to the magnitude of the coefficients $c_0, \ldots, c_n$ and the ill-conditioning of the Vandermonde matrix.

Recall from numerical linear algebra that the accuracy of solving the system $\mathbf{A}\mathbf{c} = \mathbf{f}$ depends on the *condition number* $\|\mathbf{A}\| \|\mathbf{A}^{-1}\|$ of $\mathbf{A}$.[5] Figure 1.4 shows that this condition number grows *exponentially* as n increases.[6] Thus, we should expect the computed value of $\mathbf{c}$ to have errors that scale like $\|\mathbf{A}\| \|\mathbf{A}^{-1}\| \varepsilon_{\text{mach}}$. Moreover, consider the entries in $\mathbf{c}$. For $n = 10$ (a typical example), we have

j	c_j
0	0.00000
1	363.24705
2	−10161.84204
3	113946.06962
4	−679937.11016
5	2411360.82690
6	−5328154.95033
7	7400914.85455
8	−6277742.91579
9	2968989.64443
10	−599575.07912

The entries in $\mathbf{c}$ *grow in magnitude and oscillate in sign*, akin to the simple $\mathbb{R}^2$ vector example in (1.2). The sign-flips and magnitude of the coefficients would make

$$p_n(x) = \sum_{j=0}^{n} c_j x^j$$

[4] For the double-precision arithmetic used by MATLAB, $\varepsilon_{\text{mach}} \approx 2.2 \times 10^{-16}$.

[5] For information on conditioning and the accuracy of solving linear systems, see, e.g.., Lecture 12 of Trefethen and Bau, *Numerical Linear Algebra* (SIAM, 1997).

[6] The curve has very regular behavior up until about $n = 20$; beyond that point, where $\|\mathbf{A}\| \|\mathbf{A}^{-1}\| \approx 1/\varepsilon_{\text{mach}}$, the computation is sufficiently unstable that the condition number is no longer computed accurately! For $n > 20$, take all the curves in Figure 1.4 with a healthy dose of salt.

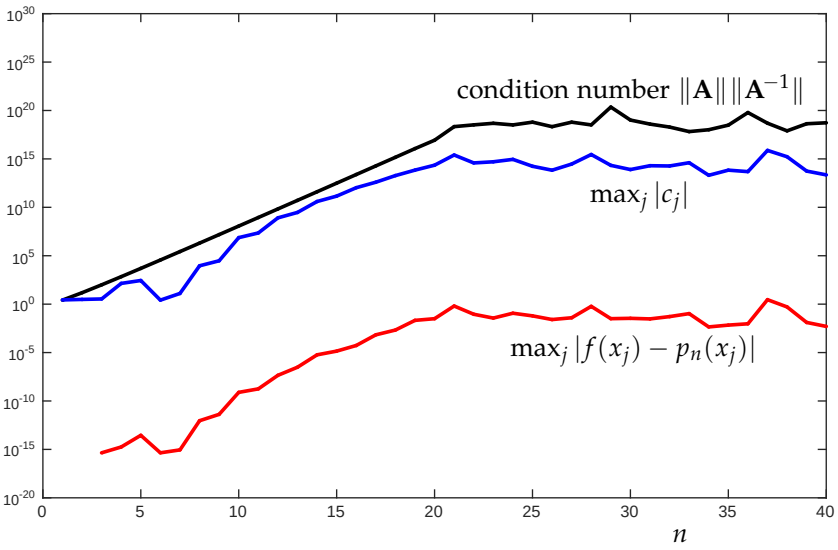


Figure 1.4: Illustration of some pitfalls of working with interpolants in the monomial basis for large n: (a) the condition number of $\mathbf{A}$ grows large with n; (b) as a result, some coefficients c_j are large in magnitude (blue line) and inaccurately computed; (c) consequently, the computed 'interpolant' p_n is far from f at the interpolation points (red line): *this red curve should be zero*!

difficult to compute accurately for large n, even if all the coefficients $c_0, \ldots, c_n$ were given exactly. Figure 1.4 shows how the largest computed value in $\mathbf{c}$ grows with n. Finally, this figure also shows the quantity we began discussing,

$$\max_{0 \le j \le n} |f(x_j) - p_n(x_j)|.$$

Rather than being nearly zero, this quantity grows with n, until the computed 'interpolating' polynomial differs from f at some interpolation point by roughly $1/10$ for the larger values of n: we must have higher standards!

This is an example where a simple problem formulation quickly yields an algorithm, but that algorithm gives unacceptable numerical results.

Perhaps you are now troubled by this entirely reasonable question: If the computations of p_n are as unstable as Figure 1.4 suggests, why should we put any faith in the plots of interpolants for $n = 10$ and, especially, $n = 30$ in Figures 1.2–1.3?

You should trust those plots because I computed them using a much better approach, about which we shall next learn.

LECTURE 2: *Superior Bases for Polynomial Interpolants*

1.3 *Polynomial interpolants in a general basis*

THE MONOMIAL BASIS may seem like the most natural way to write down the interpolating polynomial, but it can lead to numerical problems, as seen in the previous lecture. To arrive at more stable expressions for the interpolating polynomial, we will derive several different bases for $\mathcal{P}_n$ that give superior computational properties: the expansion coefficients $\{c_j\}$ will typically be smaller, and it will be simpler to determine those coefficients. This is an instance of a general principle of applied mathematics: to promote stability, express your problem in a well-conditioned basis.

Suppose we have some *basis* $\{b_j\}_{j=0}^n$ for $\mathcal{P}_n$. We seek the polynomial $p \in \mathcal{P}_n$ that interpolates f at $x_0, \ldots, x_n$. Write p in the basis as

$$p(x) = c_0 b_0(x) + c_1 b_1(x) + \cdots + c_n b_n(x).$$

Recall that $\{b_j\}_{j=0}^n$ is a *basis* if the functions *span* $\mathcal{P}_n$ and are *linearly independent*. The first requirement means that for any polynomial $p \in \mathcal{P}_n$ we can find constants $c_0, \ldots, c_n$ such that

$$p = c_0 b_0 + \cdots + c_n b_n,$$

while the second requirement means that if

$$0 = c_0 b_0 + \cdots + c_n b_n$$

then we must have $c_0 = \cdots = c_n = 0$.

We seek the coefficients $c_0, \ldots, c_n$ that express the interpolant p in this basis. The interpolation conditions are

$$\begin{aligned}
p(x_0) &= c_0 b_0(x_0) + c_1 b_1(x_0) + \cdots + c_n b_n(x_0) = f(x_0) \\
p(x_1) &= c_0 b_0(x_1) + c_1 b_1(x_1) + \cdots + c_n b_n(x_1) = f(x_1) \\
&\vdots \\
p(x_n) &= c_0 b_0(x_n) + c_1 b_1(x_n) + \cdots + c_n b_n(x_n) = f(x_n).
\end{aligned}$$

Again we have $n+1$ equations that are linear in the $n+1$ unknowns $c_0, \ldots, c_n$, hence we can arrange these in the matrix form

$$\begin{bmatrix} b_0(x_0) & b_1(x_0) & \cdots & b_n(x_0) \\ b_0(x_1) & b_1(x_1) & \cdots & b_n(x_1) \\ \vdots & \vdots & \ddots & \vdots \\ b_0(x_n) & b_1(x_n) & \cdots & b_n(x_n) \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} f(x_0) \\ f(x_1) \\ \vdots \\ f(x_n) \end{bmatrix}, \tag{1.3}$$

which can be solved via Gaussian elimination for $c_0, \ldots, c_n$.

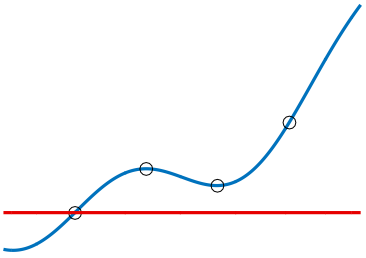
Notice that the linear system for the monomial basis in (1.1) is a special case of the system in (1.3), with the choice $b_j(x) = x^j$. Next we will look at two superior bases that give more stable expressions for the interpolant. We emphasize that when the basis changes, so to do the values of $c_0, \ldots, c_n$, *but the interpolating polynomial p remains the same, regardless of the basis we use to express it*.

1.4 *Constructing interpolants in the Newton basis*

To derive our first new basis for $\mathcal{P}_n$, we describe an alternative method for constructing the polynomial $p_n \in \mathcal{P}_n$ that interpolates $f \in C[a,b]$ at the distinct points $\{x_0, \ldots, x_n\} \subset [a,b]$. This approach, called the *Newton form* of the interpolant, builds p_n up from lower degree polynomials that interpolate f at only some of the data points.

Begin by constructing the polynomial $p_0 \in \mathcal{P}_0$ that interpolates f at x_0: $p_0(x_0) = f(x_0)$. Since p_0 is a zero-degree polynomial (i.e., a constant), it has the simple form

$$p_0(x) = c_0.$$



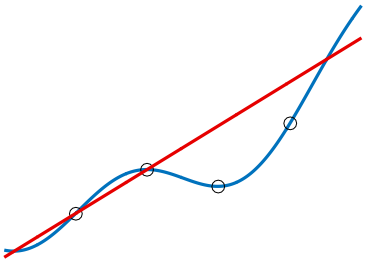
To satisfy the interpolation condition at x_0, set $c_0 = f(x_0)$. (We emphasize again: this c_0, and the c_j below, will be different from the c_j's obtained in Section 1.2 for the monomial basis.)

Next, use p_0 to build the polynomial $p_1 \in \mathcal{P}_1$ that interpolates f at both x_0 and x_1. In particular, we will require p_1 to have the form

$$p_1(x) = p_0(x) + c_1 q_1(x)$$

for some constant c_1 and some $q_1 \in \mathcal{P}_1$. Note that

$$\begin{aligned} p_1(x_0) &= p_0(x_0) + c_1 q_1(x_0) \\ &= f(x_0) + c_1 q_1(x_0). \end{aligned}$$



Since we require that $p_1(x_0) = f(x_0)$, the above equation implies that $c_1 q_1(x_0) = 0$. Either $c_1 = 0$ (which can only happen in the special case $f(x_0) = f(x_1)$, and we seek a basis that works for *any* f) or $q_1(x_0) = 0$, i.e., $q_1(x_0)$ has a root at x_0. Thus, we deduce that $q_1(x) = x - x_0$. It follows that

$$p_1(x) = c_0 + c_1(x - x_0),$$

where c_1 is still undetermined. To find c_1, use the interpolation condition at x_1:

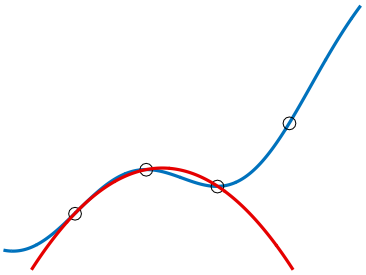
$$f(x_1) = p_1(x_1) = c_0 + c_1(x_1 - x_0).$$

Solving for c_1,

$$c_1 = \frac{f(x_1) - c_0}{x_1 - x_0}.$$

Next, find the $p_2 \in \mathcal{P}_2$ that interpolates f at x_0, x_1, and x_2, where p_2 has the form

$$p_2(x) = p_1(x) + c_2 q_2(x).$$



Similar to before, the first term, now $p_1(x)$, 'does the right thing' at the first two interpolation points, $p_1(x_0) = f(x_0)$ and $p_1(x_1) = f(x_1)$.

We require that q_2 not interfere with p_1 at x_0 and x_1, i.e., $q_2(x_0) = q_2(x_1) = 0$. Thus, we take q_2 to have the form

$$q_2(x) = (x - x_0)(x - x_1).$$

The interpolation condition at x_2 gives an equation where c_2 is the only unknown,

$$f(x_2) = p_2(x_2) = p_1(x_2) + c_2 q_2(x_2),$$

which we can solve for

$$c_2 = \frac{f(x_2) - p_1(x_2)}{q_2(x_2)} = \frac{f(x_2) - c_0 - c_1(x_2 - x_0)}{(x_2 - x_0)(x_2 - x_1)}.$$

Follow the same pattern to bootstrap up to p_n, which takes the form

$$p_n(x) = p_{n-1}(x) + c_n q_n(x),$$

where

$$q_n(x) = \prod_{j=0}^{n-1} (x - x_j),$$

and, setting $q_0(x) = 1$, we have

$$c_n = \frac{f(x_n) - \sum_{j=0}^{n-1} c_j q_j(x_n)}{q_n(x_n)}.$$

Finally, the desired polynomial takes the form

$$p_n(x) = \sum_{j=0}^{n} c_j q_j(x).$$

The polynomials q_j for $j = 0, \ldots, n$ form a basis for $\mathcal{P}_n$, called the *Newton basis*. The c_j we have just determined are the expansion coefficients for this interpolant in the Newton basis. Figure 1.5 shows the Newton basis functions q_j for $[a, b] = [0, 1]$ with $n = 5$ and $x_j = j/5$, which look considerably more distinct than the monomial basis polynomials illustrated in Figure 1.1.

This entire procedure for constructing p_n can be condensed into a system of linear equations with the coefficients $\{c_j\}_{j=0}^n$ unknown:

$$\begin{bmatrix} 1 & & & & \\ 1 & (x_1 - x_0) & & & \\ 1 & (x_2 - x_0) & (x_2 - x_0)(x_2 - x_1) & & \\ \vdots & \vdots & \vdots & \ddots & \\ 1 & (x_n - x_0) & (x_n - x_0)(x_n - x_1) & \cdots & \prod_{j=0}^{n-1}(x_n - x_j) \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} f(x_0) \\ f(x_1) \\ f(x_2) \\ \vdots \\ f(x_n) \end{bmatrix}, \tag{1.4}$$

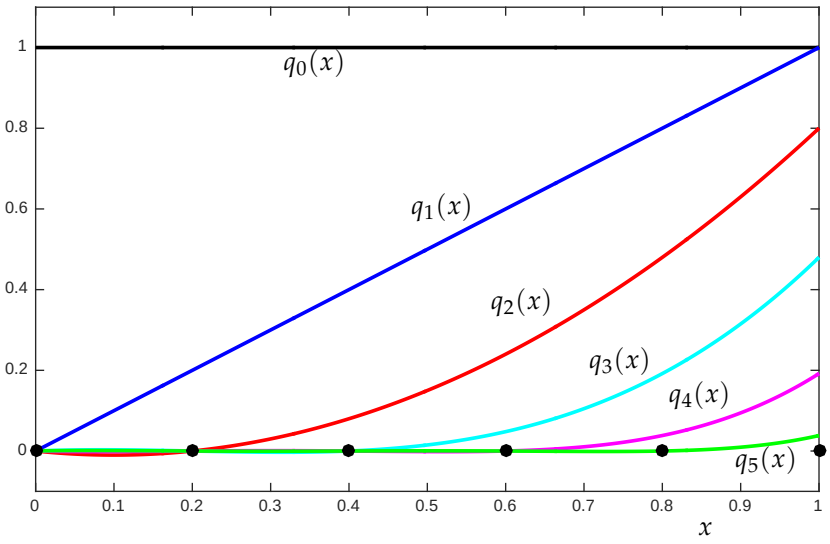


Figure 1.5: The six Newton basis polynomials $q_0, \ldots, q_5$ for $\mathcal{P}_5$, based on the interval $[a, b] = [0, 1]$ with $x_j = j/5$ (black dots). Compare these to the monomial basis polynomials in Figure 1.1: these vectors look far more distinct from one another than the monomials.

again a special case of (1.3) but with $b_j(x) = q_j(x)$. (The unspecified entries above the diagonal are zero, since $q_j(x_k) = 0$ when $k < j$.) The system (1.4) involves a triangular matrix, which is simple to solve. Clearly $c_0 = f(x_0)$, and once we know c_0, we can solve for

$$c_1 = \frac{f(x_1) - c_0}{x_1 - x_0}.$$

With c_0 and c_1, we can solve for c_2, and so on. This procedure, *forward substitution*, requires roughly n^2 floating point operations once the entries are formed.

With this Newton form of the interpolant, one can easily update p_n to p_{n+1} in order to incorporate a new data point $(x_{n+1}, f(x_{n+1}))$, as such a change affects neither the previous values of c_j nor q_j. The new data $(x_{n+1}, f(x_{n+1}))$ simply adds a new row to the bottom of the matrix in (1.4), which preserves the triangular structure of the matrix and the values of $\{c_0, \ldots, c_n\}$. If we have already found these coefficients, we easily obtain c_{n+1} through one more step of forward substitution.

1.5 *Constructing interpolants in the Lagrange basis*

The monomial basis gave us a linear system (1.1) of the form $\mathbf{Ac} = \mathbf{f}$ in which $\mathbf{A}$ was a *dense* matrix: all of its entries are nonzero. The Newton basis gave a simpler system (1.4) in which $\mathbf{A}$ was a *lower triangular* matrix. Can we go one step further, and find a set of basis functions for which the matrix in (1.3) is *diagonal*?

For the matrix to be diagonal, the jth basis function would need to have roots at all the other interpolation points x_k for $k \neq j$. Such func-

tions, denoted ℓ_j for $j = 0, \ldots, n$, are called *Lagrange basis polynomials*, and they result in the *Lagrange form* of the interpolating polynomial.

We seek to construct $\ell_j \in \mathcal{P}_n$ with $\ell_j(x_k) = 0$ if $j \neq k$, but $\ell_j(x_k) = 1$ if $j = k$. That is, ℓ_j takes the value one at x_j and has roots at all the other n interpolation points.

What form do these basis functions $\ell_j \in \mathcal{P}_n$ take? Since ℓ_j is a degree-n polynomial with the n roots $\{x_k\}_{k=0,k\neq j}^{n}$, it can be written in the form

$$\ell_j(x) = \prod_{k=0,k\neq j}^{n} \gamma_k (x - x_k)$$

for appropriate constants γ_k. We can force $\ell_j(x_j) = 1$ if all the terms in the above product are one when $x = x_j$, i.e., when $\gamma_k = 1/(x_j - x_k)$, so that

$$\ell_j(x) = \prod_{k=0,k\neq j}^{n} \frac{x - x_k}{x_j - x_k}.$$

This form makes it clear that $\ell_j(x_j) = 1$. With these new basis functions, the constants $\{c_j\}$ can be written down immediately. The interpolating polynomial has the form

$$p_n(x) = \sum_{k=0}^{n} c_k \ell_k(x).$$

When $x = x_j$, all terms in this sum will be zero except for one, the $k = j$ term (since $\ell_k(x_j) = 0$ except when $j = k$). Thus,

$$p_n(x_j) = c_j \ell_j(x_j) = c_j,$$

so we can directly write down the coefficients, $c_j = f(x_j)$.

As desired, this approach to constructing basis polynomials leads to a diagonal matrix $\mathbf{A}$ in the equation $\mathbf{Ac} = \mathbf{f}$ for the coefficients. Since we also insisted that $\ell_j(x_j) = 1$, the matrix $\mathbf{A}$ is actually just the *identity* matrix:

$$\begin{bmatrix} 1 & & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} f(x_0) \\ f(x_1) \\ \vdots \\ f(x_n) \end{bmatrix}.$$

Now the coefficient matrix is simply the identity.

A forthcoming exercise will investigate an important flexible and numerically stable method for constructing and evaluating Lagrange interpolants known as *barycentric interpolation*.

Figure 1.6 shows the Lagrange basis functions for $n = 5$ with $[a, b] = [0, 1]$ and $x_j = j/5$, the same parameters used in the plots of the monomial and Newton bases earlier.

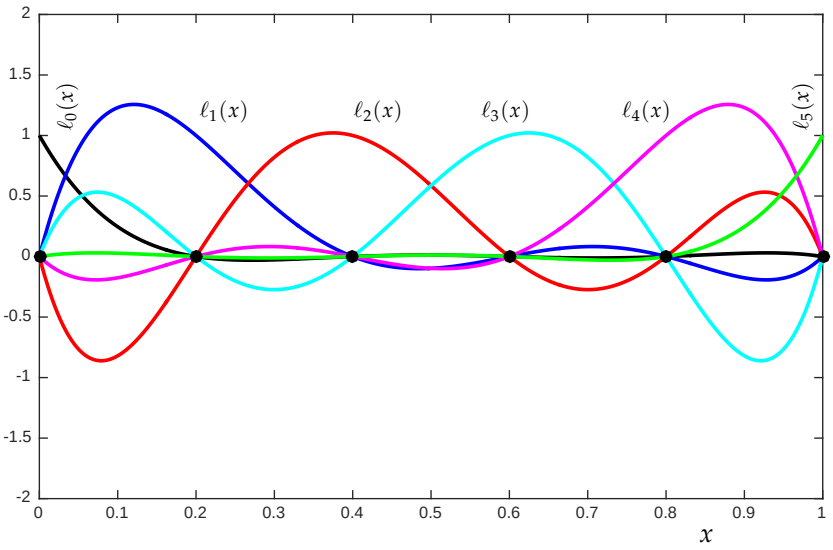


Figure 1.6: The six Lagrange basis polynomials $\ell_0, \ldots, \ell_5$ for $\mathcal{P}_5$, based on the interval $[a, b] = [0, 1]$ with $x_j = j/5$ (black dots). Note that each Lagrange polynomial has roots at n of the interpolation points. Compare these polynomials to the monomial and Newton basis polynomials in Figures 1.1 and 1.5 (but note the different vertical scale): these basis vectors look most independent of all.

The fact that these basis functions are not as closely aligned as the previous ones has interesting consequences on the size of the coefficients $\{c_j\}$. For example, if we have $n + 1 = 6$ interpolation points for $f(x) = \sin(10x) + \cos(10x)$ on $[0, 1]$, we obtain the following coefficients:

	monomial	Newton	Lagrange
c_0	1.0000000e+00	1.0000000e+00	1.0000000e+00
c_1	4.0861958e+01	-2.5342470e+00	4.9315059e-01
c_2	-3.8924180e+02	-1.7459341e+01	-1.4104461e+00
c_3	1.0775024e+03	1.1232385e+02	6.8075479e-01
c_4	-1.1683645e+03	-2.9464687e+02	8.4385821e-01
c_5	4.3685881e+02	4.3685881e+02	-1.3830926e+00

We emphasize that all three approaches (in exact arithmetic) must yield the same unique polynomial, but they are expressed in different bases. The behavior in floating point arithmetic varies significantly with the choice of basis; the monomial basis is the clear loser.

LECTURE 3: *Interpolation Error Bounds*

1.6 Convergence theory for polynomial interpolation

Interpolation can be used to generate low-degree polynomials that approximate a complicated function over the interval $[a, b]$. One might assume that the more data points that are interpolated (for a fixed $[a, b]$), the more accurate the resulting approximation. In this lecture, we address the behavior of the maximum error

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

as the number of interpolation points—hence, the degree of the interpolating polynomial—is increased. We begin with a theoretical result.

Theorem 1.2 (Weierstrass Approximation Theorem)**.**
Suppose $f \in C[a, b]$. For any $\varepsilon > 0$ there exists some polynomial p_n of finite degree n such that $\max_{x\in[a,b]} |f(x) - p_n(x)| \leq \varepsilon$.

Unfortunately, we do not have time to prove this in class.[7] As stated, this theorem gives no hint about what the approximating polynomial looks like, whether p_n interpolates f at $n + 1$ points, or merely approximates f well throughout $[a, b]$, nor does the Weierstrass theorem describe the accuracy of a polynomial for a specific value of n (though one could gain insight into such questions by studying the constructive proof).

[7] The typical proof is a construction based on Bernstein polynomials; see, e.g., Kincaid and Cheney, *Numerical Analysis*, 3rd edition, pages 320–323. This result can be generalized to the Stone–Weierstrass Theorem, itself a special case of Bishop's Theorem for approximation problems in operator algebras; see e.g., §5.6–§5.8 of Rudin, *Functional Analysis*, 2nd ed. (McGraw Hill, 1991).

On the other hand, for the interpolation problem studied in the preceding lectures, we can obtain a specific error formula that gives a bound on $\max_{x\in[a,b]} |f(x) - p_n(x)|$. From this bound, we can deduce if interpolating f at increasingly many points will eventually yield a polynomial approximation to f that is accurate to any specified precision.

For any $\widehat{x} \in [a, b]$ that is *not* of the interpolation points, we seek to measure the error

$$f(\widehat{x}) - p_n(\widehat{x}),$$

where $p_n \in \mathcal{P}_n$ is the interpolant to f at the distinct points $x_0, \ldots, x_n \in [a, b]$. We can get a grip on this error from the following perspective. Extend p_n by one degree to give a new polynomial that additionally interpolates f at $\widehat{x}$. This is easy to do with the Newton form of the interpolant; write the new polynomial as

$$p_n(x) + \lambda \prod_{j=0}^{n} (x - x_j),$$

for constant λ chosen so that

$$f(\widehat{x}) = p_n(\widehat{x}) + \lambda \prod_{j=0}^{n} (\widehat{x} - x_j).$$

For convenience, we write

$$w(x) := \prod_{j=0}^{n} (x - x_j),$$

so we could solve for λ as

$$\lambda = \frac{f(\widehat{x}) - p_n(\widehat{x})}{w(\widehat{x})}. \tag{1.5}$$

Now notice that

$$0 = f(\widehat{x}) - \big(p_n(\widehat{x}) + \lambda w(\widehat{x})\big)$$

implies

$$f(\widehat{x}) - p_n(\widehat{x}) = \lambda w(\widehat{x}), \tag{1.6}$$

which is an expression for the desired error $f(\widehat{x}) - p_n(\widehat{x})$. Unfortunately, the formula (1.5) does not give much insight into how the error behaves as a function of f and the interpolation points.

Theorem 1.3 (Interpolation Error Formula)**.**
Suppose $f \in C^{n+1}[a,b]$ and let $p_n \in \mathcal{P}_n$ denote the polynomial that interpolates $\{(x_j, f(x_j))\}_{j=0}^{n}$ for distinct points $x_j \in [a,b]$, $j = 0, \ldots, n$. Then for every $x \in [a,b]$ there exists $\xi \in [a,b]$ such that

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

From this formula follows a bound for the worst error over $[a,b]$:

$$\max_{x\in[a,b]} |f(x) - p_n(x)| \le \left(\max_{\xi\in[a,b]} \frac{|f^{(n+1)}(\xi)|}{(n+1)!} \right) \left(\max_{x\in[a,b]} \prod_{j=0}^{n} |x - x_j| \right). \tag{1.7}$$

We shall carefully prove this essential result; it will repay the effort, for this theorem becomes the foundation upon which we shall build the convergence theory for piecewise polynomial approximation and interpolatory quadrature rules for definite integrals.

Proof. Consider some arbitrary point $\widehat{x} \in [a,b]$. We seek a descriptive expression for the error $f(\widehat{x}) - p_n(\widehat{x})$. If $\widehat{x} = x_j$ for some $j \in \{0, \ldots, n\}$, then $f(\widehat{x}) - p_n(\widehat{x}) = 0$ and there is nothing to prove. Thus, suppose for the rest of the proof that $\widehat{x}$ is not one of the interpolation points.

To describe $f(\widehat{x}) - p_n(\widehat{x})$, we shall build the polynomial of degree $n+1$ that interpolates f at $x_0, \ldots, x_n$, and also $\widehat{x}$. Of course, this polynomial will give zero error at $\widehat{x}$, since it interpolates f there. From this polynomial we can extract a formula for $f(\widehat{x}) - p_n(\widehat{x})$ by measuring how much the degree $n+1$ interpolant improves upon the degree-n interpolant p_n at $\widehat{x}$.

Since we wish to understand the relationship of the degree $n+1$ interpolant to p_n, we shall write that degree $n+1$ interpolant in a manner that explicitly incorporates p_n. Given this setting, use of the Newton form of the interpolant is natural; i.e., we write the degree $n+1$ polynomial as

$$p_n(x) + \lambda \prod_{j=0}^{n} (x - x_j)$$

for some constant λ chosen to make the interpolant exact at $\widehat{x}$. For convenience, we write

$$w(x) \equiv \prod_{j=0}^{n} (x - x_j)$$

and then denote the error of this degree $n+1$ interpolant by

$$\phi(x) \equiv f(x) - \big(p_n(x) + \lambda w(x)\big).$$

To make the polynomial $p_n(x) + \lambda w(x)$ interpolate f at $\widehat{x}$, we shall pick λ such that $\phi(\widehat{x}) = 0$. The fact that $\widehat{x} \notin \{x_j\}_{j=0}^{n}$ ensures that $w(\widehat{x}) \neq 0$, and so we can force $\phi(\widehat{x}) = 0$ by setting

$$\lambda = \frac{f(\widehat{x}) - p_n(\widehat{x})}{w(\widehat{x})}.$$

Furthermore, since $f(x_j) = p_n(x_j)$ and $w(x_j) = 0$ at all the $n+1$ interpolation points $x_0, \ldots, x_n$, we also have $\phi(x_j) = f(x_j) - p_n(x_j) - \lambda w(x_j) = 0$. Thus, ϕ is a function with at least $n+2$ zeros in the interval $[a, b]$. Rolle's Theorem[8] tells us that between every two consecutive zeros of ϕ, there is some zero of ϕ'. Since ϕ has at least $n+2$ zeros in $[a, b]$, ϕ' has at least $n+1$ zeros in this same interval. We can repeat this argument with ϕ' to see that ϕ'' must have at least n zeros in $[a, b]$. Continuing in this manner with higher derivatives, we eventually conclude that $\phi^{(n+1)}$ must have at least one zero in $[a, b]$; we denote this zero as ξ, so that $\phi^{(n+1)}(\xi) = 0$.

[8] Recall the Mean Value Theorem from calculus: Given $d > c$, suppose $f \in C[c, d]$ is differentiable on (c, d). Then there exists some $\eta \in (c, d)$ such that $(f(d) - f(c))/(d - c) = f'(\eta)$. Rolle's Theorem is a special case: If $f(d) = f(c)$, then there is some point $\eta \in (c, d)$ such that $f'(\eta) = 0$.

We now want a more concrete expression for $\phi^{(n+1)}$. Note that

$$\phi^{(n+1)}(x) = f^{(n+1)}(x) - p_n^{(n+1)}(x) - \lambda w^{(n+1)}(x).$$

Since p_n is a polynomial of degree n or less, $p_n^{(n+1)} \equiv 0$. Now observe that w is a polynomial of degree $n+1$. We could write out all the coefficients of this polynomial explicitly, but that is a bit tedious,

and we do not need all of them. Simply observe that we can write $w(x) = x^{n+1} + q(x)$, for some $q \in \mathcal{P}_n$, and this polynomial q will vanish when we take $n+1$ derivatives:

$$w^{(n+1)}(x) = \left(\frac{d^{n+1}}{dx^{n+1}} x^{n+1} \right) + q^{(n+1)}(x) = (n+1)! + 0.$$

Assembling the pieces, $\phi^{(n+1)}(x) = f^{(n+1)}(x) - \lambda\,(n+1)!$. Since $\phi^{(n+1)}(\xi) = 0$, we conclude that

$$\lambda = \frac{f^{(n+1)}(\xi)}{(n+1)!}.$$

Substituting this expression into $0 = \phi(\widehat{x}) = f(\widehat{x}) - p_n(\widehat{x}) - \lambda w(\widehat{x})$, we obtain

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

This error bound has strong parallels to the remainder term in Taylor's formula. Recall that for sufficiently smooth h, the Taylor expansion of f about the point x_0 is given by

$$f(x) = f(x_0) + (x - x_0) f'(x_0) + \cdots + \frac{(x - x_0)^k}{k!} f^{(k)}(x_0) + \textsc{remainder}.$$

Ignoring the remainder term at the end, note that the Taylor expansion gives a polynomial model of f, but one based on local information about f and its derivatives, as opposed to the polynomial interpolant, which is based on global information, but only about f, not its derivatives.

An interesting feature of the interpolation bound is the polynomial $w(x) = \prod_{j=0}^{n} (x - x_j)$. This quantity plays an essential role in approximation theory, and also a closely allied subdiscipline of complex analysis called *potential theory*. Naturally, one might wonder what choice of points $\{x_j\}$ minimizes $|w(x)|$: We will revisit this question when we study approximation theory in the near future. For now, we simply note that the points that minimize $|w(x)|$ over $[a, b]$ are called *Chebyshev points*, which are clustered more densely at the ends of the interval $[a, b]$.

Example 1.1 ($f(x) = \sin(x)$). We shall apply the interpolation bound to $f(x) = \sin(x)$ on $x \in [-5, 5]$. Since $f^{(n+1)}(x) = \pm \sin(x)$ or $\pm \cos(x)$, we have $\max_{x \in [-5,5]} |f^{(n+1)}(x)| = 1$ for all n. The interpolation result we just proved then implies that *for any choice of distinct interpolation points in* $[-5, 5]$,

$$\prod_{j=0}^{n} |x - x_j| < 10^{n+1},$$

the worst case coming if all the interpolation points are clustered at an end of the interval $[-5, 5]$. Now our theorem ensures that

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

For small values of n, this bound will be very large, but eventually $(n+1)!$ grows much faster than 10^{n+1}, so we conclude that our error must go to zero as $n \to \infty$ *regardless of where in* $[-5, 5]$ *we place our interpolation points*! The error bound is shown in the first plot below.

Consider the following specific example: Interpolate $\sin(x)$ at points uniformly selected in $[-1, 1]$. At first glance, you might think there is no reason that we should expect our interpolants p_n to converge to $\sin(x)$ for all $x \in [-5, 5]$, since we are only using data from the subinterval $[-1, 1]$, which is only 20% of the total interval and does not even include one entire period of the sine function. (In fact, $\sin(x)$ attains neither its maximum nor minimum on $[-1, 1]$.) Yet the error bound we proved above ensures that the polynomial interpolant must converge throughout $[-5, 5]$. This is illustrated in the first plot below. The next plots show the interpolants $p_4(x)$ and $p_{10}(x)$ generated from these interpolation points. Not surprisingly, these interpolants are most accurate near $[-1, 1]$, the location of the interpolation points (shown as circles), but we still see convergence well beyond $[-1, 1]$, in the same way that the Taylor expansion for $\sin(x)$ at $x = 0$ will converge everywhere.

Example 1.2 (Runge's Example)**.** The error bound (1.7) suggests those functions for which interpolants might fail to converge as $n \to \infty$: beware if higher derivatives of f are large in magnitude over the interpolation interval. The most famous example of such behavior is due to Carl Runge, who studied convergence of interpolants for $f(x) = 1/(1 + x^2)$ on the interval $[-5, 5]$. This function looks beautiful: it resembles a bell curve, with no singularities in sight on $\mathbb{R}$, as Figure 1.8 shows. However, the interpolants to f at uniformly spaced points over $[-5, 5]$ do not seem to converge even for $x \in [-5, 5]$.

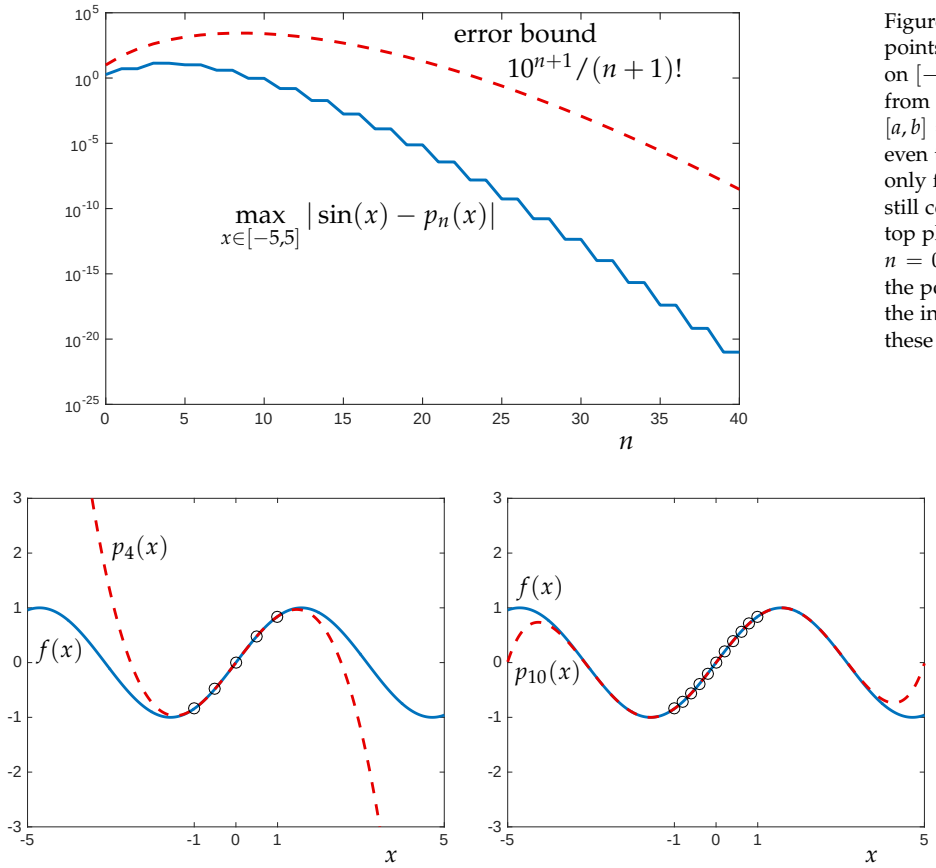


Figure 1.7: Interpolation of $\sin(x)$ at points $x_0, \ldots, x_n$ uniformly distributed on $[-1, 1]$. We develop an error bound from Theorem 1.3 for the interval $[a, b] = [-5, 5]$. The bound proves that even though the interpolation points only fall in $[-1, 1]$, the interpolant will still converge throughout $[-5, 5]$. The top plot shows this convergence for $n = 0, \ldots, 40$; the bottom plots show the polynomials p_4 and p_10, along with the interpolation points that determine these polynomials (black circles).

Look at successive derivatives of f; they expose its crucial flaw:

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

$$f''(x) = \frac{8x^2}{(1+x^2)^3} - \frac{2}{(1+x^2)^2}$$

$$f'''(x) = -\frac{48x^3}{(1+x^2)^4} + \frac{24x}{(1+x^2)^3}$$

$$f^{(iv)}(x) = \frac{348x^4}{(1+x^2)^5} - \frac{288x^2}{(1+x^2)^4} + \frac{24}{(1+x^2)^3}$$

$$f^{(vi)}(x) = \frac{46080x^6}{(1+x^2)^7} - \frac{57600x^4}{(1+x^2)^6} + \frac{17280x^2}{(1+x^2)^5} - \frac{720}{(1+x^2)^4}.$$

At certain points on $[-5, 5]$, $f^{(n+1)}$ blows up more rapidly than $(n+1)!$, and the interpolation bound (1.7) suggests that p_n *will not* converge to f on $[-5, 5]$ as n gets large. Not only does p_n fail to converge to f; the error between certain interpolation points gets enormous as n increases.

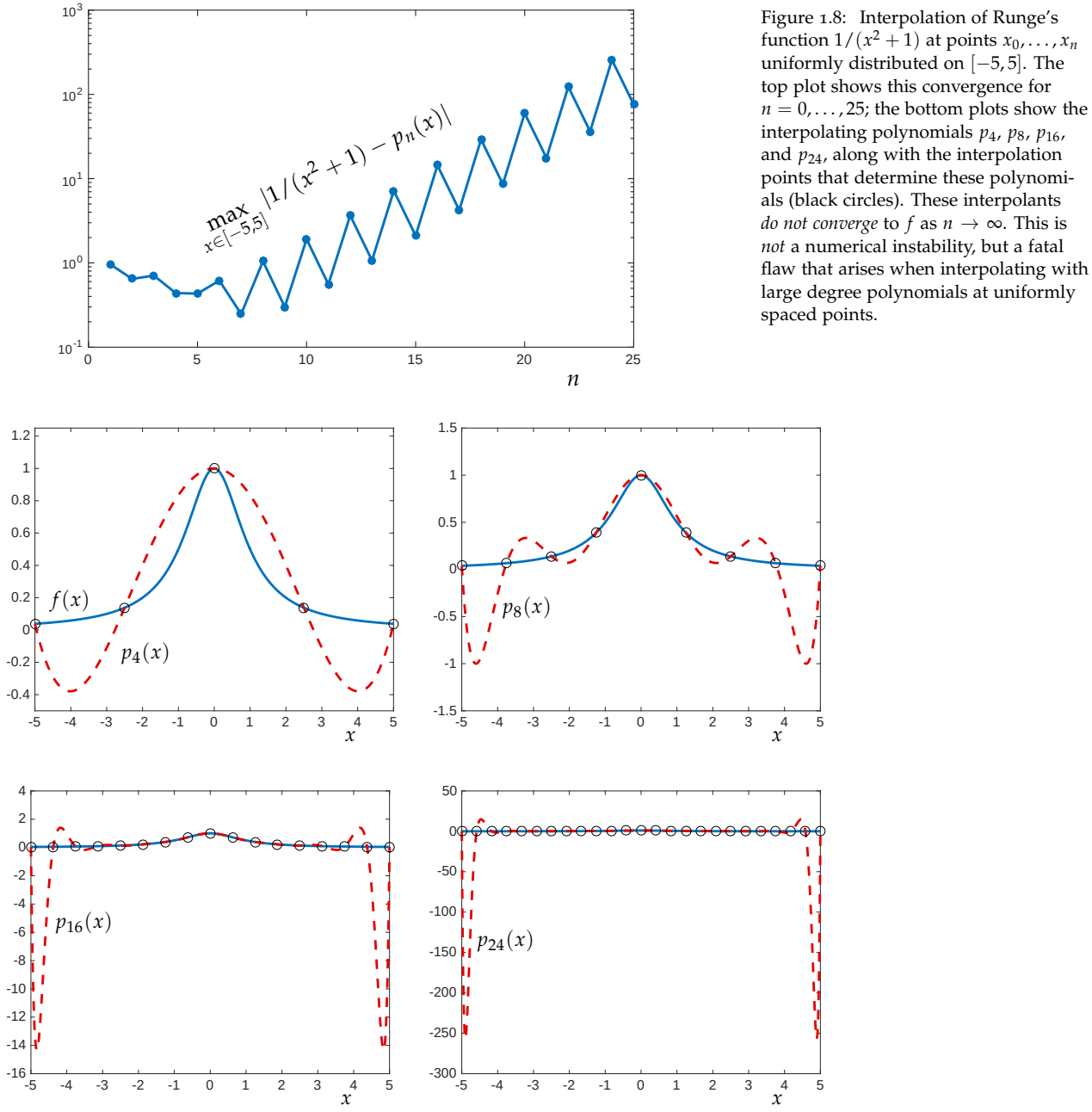


Figure 1.8: Interpolation of Runge's function $1/(x^2+1)$ at points $x_0, \ldots, x_n$ uniformly distributed on $[-5,5]$. The top plot shows this convergence for $n = 0, \ldots, 25$; the bottom plots show the interpolating polynomials p_4, p_8, p_{16}, and p_{24}, along with the interpolation points that determine these polynomials (black circles). These interpolants *do not converge* to f as $n \to \infty$. This is *not* a numerical instability, but a fatal flaw that arises when interpolating with large degree polynomials at uniformly spaced points.

The following code uses MATLAB's Symbolic Toolbox to compute higher derivatives of the Runge function. Several of the resulting plots follow.[9] *Note how the scale on the vertical axis changes from plot to plot!*

[9] Not all versions of MATLAB have the Symbolic Toolbox, but you should be able to run this code on any Student Edition or on copies on Virginia Tech network.

```
% rungederiv.m
% routine to plot derivatives of Runge's example,
%     f(x) = 1/(1+x^2) on [-5,5]
```

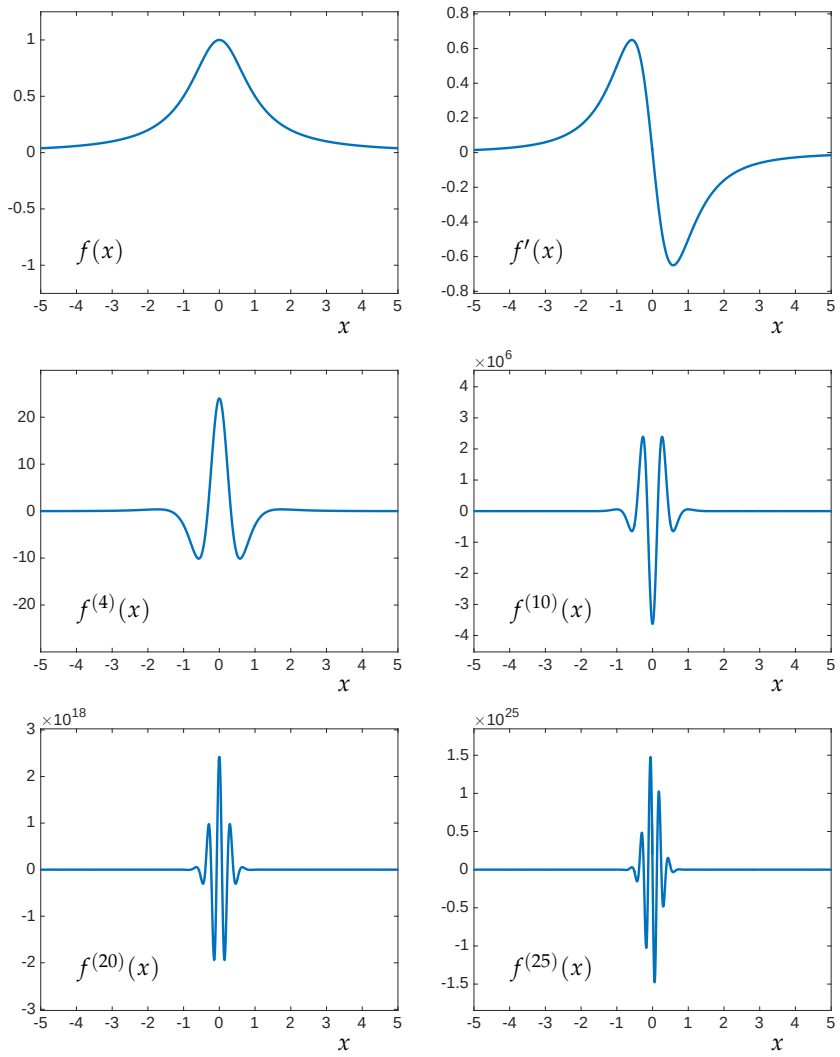


Figure 1.9: Runge's function

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

and a few of its derivatives on $x \in [-5,5]$. Notice how large the derivatives grow in magnitude: the vertical scale on the plot for $f^{(25)}$ (bottom-right) is 10^{25}.

```
figure(1), clf, set(gca,'fontsize',18)
for j=0:25
   syms x
   fj = vectorize(diff(1/(x^2+1),j));            % compute derivative (Symbolic Toolbox)
   x = linspace(-5,5,1000); fjx = eval(fj);      % evaluate on a grid of points
   plot(x,fjx,'b-','linewidth',2);               % plot derivative
   title(sprintf('Runge''s Example: f^{(%d)}(x)',j),'fontsize',14)
   input(' ')
end
```

Some improvement can be made by a careful selection of the interpolation points $\{x_0\}$. In fact, if one interpolates Runge's example, $f(x) = 1/(1+x^2)$, at the *Chebyshev points* for $[-5,5]$,

$$x_j = 5\cos\Big(\frac{j\pi}{n}\Big), \qquad j = 0,\ldots,n,$$

then the interpolant will converge!

As a general rule, interpolation at Chebyshev points is greatly preferred over interpolation at uniformly spaced points for reasons we shall understand in a few lectures. However, even this set is not perfect: there exist functions for which the interpolants at Chebyshev points do not converge. Examples to this effect were constructed by Marcinkiewicz and Grunwald in the 1930s. We close with two results of a more general nature.[10] We require some general notation to describe a family of interpolation points that can change as the polynomial degree increases. Toward this end, let $\{x_j^{[n]}\}_{j=0}^n$ denote the set of interpolation points used to construct the degree-n interpolant. As we are concerned here with the behavior of interpolants as $n \to \infty$, so we will speak of the *system of interpolation points* $\{\{x_j^{[n]}\}_{j=0}^n\}_{n=0}^\infty$.

[10] An excellent exposition of these points is given in volume 3 of I. P. Natanson, *Constructive Function Theory* (Ungar, 1965).

Our first result is bad news.

Theorem 1.4 (Faber's Theorem)**.**
Let $\{\{x_j^{[n]}\}_{j=0}^n\}_{n=0}^\infty$ be any system of interpolation points with $x_j^{[n]} \in [a,b]$ and $x_j^{[n]} \neq x_\ell^{[n]}$ for $j \neq \ell$ (i.e., distinct interpolation points for each polynomial degree). Then there exists some function $f \in C[a,b]$ such that the polynomials p_n that interpolate f at $\{x_j^{[n]}\}_{j=0}^n$ do not converge uniformly to f in $[a,b]$ as $n \to \infty$.

The good news is that there always exists a suitable set of interpolation points for any given $f \in C[a,b]$.

Theorem 1.5 (Marcinkiewicz's Theorem)**.**
Given any $f \in C[a,b]$, there exist a system of interpolation points with $x_j^{[n]} \in [a,b]$ such that the polynomials p_n that interpolate f at $\{x_j^{[n]}\}_{j=0}^n$ converge uniformly to f in $[a,b]$ as $n \to \infty$.

These results are both quite abstract; for example, the construction of the offensive example in Faber's Theorem is not nearly as concrete as Runge's nice example for uniformly spaced points discussed above. We will revisit the question of the convergence of interpolants in a few weeks when we discuss Chebyshev polynomials. Then we will be able to say something much more positive: there exists a nice set of points that works for all but the ugliest functions in $C[a,b]$.

LECTURE 4: *Constructing Finite Difference Formulas*

1.7 Application: Interpolants for Finite Difference Formulas

The most obvious use of interpolants is to construct polynomial models of more complicated functions. However, numerical analysts rely on interpolants for many other numerical chores. For example, in a few weeks we shall see that common techniques for approximating definite integrals amount to exactly integrating a polynomial interpolant. Here we turn to a different application: the use of interpolating polynomials to derive finite difference formulas that approximate derivatives, the to use those formulas to construct approximations of differential equation boundary value problems.

1.7.1 Derivatives of Interpolants

Theorem 1.3 from the last lecture showed how well the interpolant $p_n \in \mathcal{P}_n$ approximates f. Here we seek deeper connections between p_n and f.

> How well do derivatives of p_n approximate derivatives of f?

Let $p \in \mathcal{P}_n$ denote the degree-n polynomial that interpolates f at the distinct points $x_0, \ldots, x_n$. We want to derive a bound on the error $f'(x) - p'(x)$. Let us take the proof of Theorem 1.3 as a template, and adapt it to analyze the error in the derivative.

For simplicity, assume that $\widehat{x} \in \{x_0, \ldots, x_n\}$, i.e., assume that $\widehat{x}$ is one of the interpolation points. Suppose we extend $p(x)$ by one degree so that the derivative of the resulting polynomial at $\widehat{x}$ matches $f'(\widehat{x})$. To do so, use the Newton form of the interpolant, writing the new polynomial as

$$p(x) + \lambda w(x),$$

again with

$$w(x) := \prod_{j=0}^{n} (x - x_j).$$

The derivative interpolation condition at $\widehat{x}$ is

$$f'(\widehat{x}) = p'(\widehat{x}) + \lambda w'(\widehat{x}), \tag{1.8}$$

and since $w(x_j) = 0$ for $j = 0, \ldots, n$, the new polynomial maintains the standard interpolation at the $n + 1$ interpolation points:

$$f(x_j) = p(x_j) + \lambda w(x_j), \qquad j = 0, \ldots, n. \tag{1.9}$$

Here we must tweak the proof of Theorem 1.3 slightly. As in that proof, define the error function

$$\phi(x) := f(x) - \big(p(x) + \lambda w(x)\big).$$

Because of the standard interpolation conditions (1.9) at $x_0, \ldots, x_n$, ϕ must have $n+1$ zeros. Now Rolle's theorem implies that ϕ' has (at least) n zeros, each of which occurs strictly between every two consecutive interpolation points. But in addition to these points, ϕ' must have another root at $\widehat{x}$ (which we have required to be one of the interpolation points, and thus distinct from the other n roots). Thus, ϕ' has $n+1$ distinct zeros on $[a,b]$.

Now, repeatedly apply Rolle's theorem to see that ϕ'' has n distinct zeros, ϕ''' has $n-1$ distinct zeros, etc., to conclude that $\phi^{(n+1)}$ has a zero: call it ξ. That is,

$$0 = \phi^{(n+1)}(\xi) = f^{(n+1)}(\xi) - \big(p^{(n+1)}(\xi) + \lambda w^{(n+1)}(\xi)\big). \tag{1.10}$$

We must analyze

$$\phi^{(n+1)}(x) = f^{(n+1)}(x) - \big(p^{(n+1)}(x) + \lambda w^{(n+1)}(x)\big).$$

Just as in the proof of Theorem 1.3, note that $p^{(n+1)} = 0$ since $p \in \mathcal{P}_n$ and $w^{(n+1)}(x) = (n+1)!$. Thus from (1.10) conclude

$$\lambda = \frac{f^{(n+1)}(\xi)}{(n+1)!}.$$

From (1.8) we arrive at

$$f'(\widehat{x}) - p'(\widehat{x}) = \lambda w'(\widehat{x}) = \frac{f^{(n+1)}(\xi)}{(n+1)!} w'(\widehat{x}).$$

To arrive at a concrete estimate, perhaps we should say something more specific about $w'(\widehat{x})$. Expanding w and computing w' explicitly will take us far into the weeds; it suffices to invoke an interesting result from 1889.

Lemma 1.1 (Markov brothers' inequality for first derivatives)**.**
For any polynomial $q \in \mathcal{P}_n$,

$$\max_{x\in[a,b]} |q'(x)| \le \frac{2n^2}{b-a} \max_{x\in[a,b]} |q(x)|.$$

We can thus summarize our discussion as the following theorem, an analogue of Theorem 1.3.

Lemma 1.1 was proved by Andrey Markov in 1889, generalizing a result for $n = 2$ that was obtained by the famous chemist Mendeleev in his research on specific gravity. Markov's younger brother Vladimir extended it to higher derivatives (with a more complicated right-hand side) in 1892. The interesting history of this inequality (and extensions into the complex plane) is recounted in a paper by Ralph Boas, Jr. on 'Inequalities for the derivatives of polynomials,' *Math. Magazine* 42 (4) 1969, 165–174. The result is called the 'Markov brothers' inequality' to distinguish it from the more famous 'Markov's inequality' in probability theory (named, like 'Markov chains,' for Andrey; Vladimir died of tuberculosis at the age of 25 in 1897).

Theorem 1.6 (Bound on the derivative of an interpolant)**.** Suppose $f \in C^{(n+1)}[a,b]$ and let $p_n \in \mathcal{P}_n$ denote the polynomial that interpolates $\{(x_j, f(x_j))\}_{j=0}^n$ at distinct points $x_j \in [a,b]$, $j = 0, \ldots, n$. Then for every $x_k \in \{x_0, \ldots, x_n\}$, there exists some $\xi \in [a,b]$ such that

$$f'(x_k) - p_n'(x_k) = \frac{f^{(n+1)}(\xi)}{(n+1)!} w'(x_k),$$

where $w(x) = \prod_{j=0}^n (x - x_j)$. From this formula follows the bound

$$|f'(x_k) - p_n'(x_k)| \leq \frac{2n^2}{b-a} \left(\max_{\xi \in [a,b]} \frac{|f^{(n+1)}(\xi)|}{(n+1)!} \right) \left(\max_{x \in [a,b]} \prod_{j=0}^n |x - x_j| \right). \tag{1.11}$$

Why don't we simply 'take a derivative of Theorem 1.3'? The subtlety is the $f^{(n+1)}(\xi)$ term in Theorem 1.3. Since ξ depends on x, taking the derivative of $f^{(n+1)}(\xi(x))$ via the chain rule would require explicit knowledge of $\xi(x)$. We don't want to work out a formula for $\xi(x)$ for each f and interval $[a,b]$.

Contrast the bound (1.11) with (1.7) from Theorem 1.3: the bounds are the same, aside from the leading constant $2n^2/(b-a)$ inherited from Lemma 1.1.

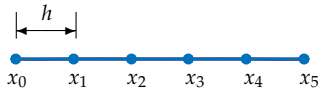
For our later discussion it will help to get a rough bound for he case where the interpolation points are uniformly distributed, i.e.,

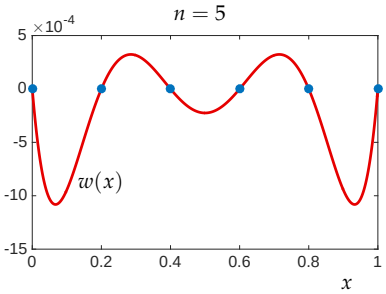
$$x_j = a + jh, \qquad j = 0, \ldots, n$$

with spacing equal to $h := (b-a)/n$. We seek to bound

$$\max_{x \in [a,b]} \prod_{j=0}^n |x - x_j|,$$

i.e., maximize the product of the distances of x from each of the interpolation points. Consider the sketch in the margin. Think about how you would place $x \in [x_0, x_n]$ so as to make $\prod_{j=0}^n |x - x_j|$ as large as possible. Putting x somewhere toward the ends, but not too near one of the interpolation points, will maximize product. Convince yourself that, regardless of where x is placed within $[x_0, x_n]$:





Notice that for $n = 5$ uniformly spaced points on $[0,1]$, $w(x)$ takes its maximum magnitude between the two interpolation points on each end of the domain.

- at least one interpolation point is no more than $h/2$ away from x;
- a different interpolation point is no more than h away from x;
- a different interpolation point is no more than $2h$ away from x;

  $\vdots$
- the last remaining (farthest) interpolation point is no more than $nh = b - a$ away from x.

This reasoning gives the bound

$$\max_{x \in [a,b]} \prod_{j=0}^n |x - x_j| \leq \frac{h}{2} \cdot h \cdot 2h \cdots nh = \frac{h^{n+1} n!}{2}. \tag{1.12}$$

Substituting this into (1.11) and using $b - a = nh$ gives the following result.

Corollary 1.1 (The derivative of an interpolant at equispaced points)**.** Suppose $f \in C^{(n+1)}[a,b]$ and let $p_n \in \mathcal{P}_n$ denote the polynomial that interpolates $\{(x_j, f(x_j))\}_{j=0}^n$ at equispaced points $x_j = a + jh$ for $h = (b-a)/n$. Then for every $x_k \in \{x_0, \ldots, x_n\}$,

$$|f'(x_k) - p_n'(x_k)| \leq \frac{nh^n}{n+1}\left(\max_{\xi\in[a,b]} |f^{(n+1)}(\xi)|\right). \tag{1.13}$$

1.7.2 *Finite difference formulas*

The preceding analysis was toward a very specific purpose: to use interpolating polynomials to develop formulas that approximate derivatives of f from the value of f at a few points.

Example 1.3 (First derivative)**.** We begin with the simplest case: formulas for the first derivative $f'(x)$. Pick some value for x_0 and some spacing parameter $h > 0$.

First construct the linear interpolant to f at x_0 and $x_1 = x_0 + h$. Using the Newton form, we have

$$\begin{aligned} p_1(x) &= f(x_0) + \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x - x_0) \\ &= f(x_0) + \frac{f(x_1) - f(x_0)}{h}(x - x_0). \end{aligned}$$

Take a derivative of the interpolant:

$$p_1'(x) = \frac{f(x_1) - f(x_0)}{h}, \tag{1.14}$$

which is precisely the conventional definition of the derivative, if we take the limit $h \to 0$. But how accurate an approximation is it? Appealing to Corollary 1.1 with $n = 1$ and $[a,b] = [x_0, x_1] = x_0 + [0,h]$, we have

$$|f'(x_k) - p_1'(x_k)| \leq \left(\frac{1}{2}\max_{\xi\in[x_0,x_1]} |f''(\xi)|\right) h \tag{1.15}$$

Does the bound (1.15) improve if we use a quadratic interpolant to f through x_0, $x_1 = x_0 + h$ and $x_2 = x_0 + 2h$? Again using the Newton form, write

$$\begin{aligned} p_2(x) &= f(x_0) + \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x - x_0) + \frac{f(x_2) - f(x_0) - \frac{f(x_1)-f(x_0)}{x_1-x_0}(x_2 - x_0)}{(x_2 - x_0)(x_2 - x_1)}(x - x_0)(x - x_1) \\ &= f(x_0) + \frac{f(x_1) - f(x_0)}{h}(x - x_0) + \frac{f(x_0) - 2f(x_1) + f(x_2)}{2h^2}(x - x_0)(x - x_1). \end{aligned} \tag{1.16}$$

Taking a derivative of this interpolant with respect to x gives

$$p_2'(x) = \frac{f(x_1) - f(x_0)}{h} + \frac{f(x_0) - 2f(x_1) + f(x_2)}{2h^2}(2x - x_0 - x_1).$$

Evaluate this at $x = x_0$, $x = x_1$, and $x = x_2$ and simplify as much as possible to get:

$$p_2'(x_0) = \frac{-3f(x_0) + 4f(x_1) - f(x_2)}{2h} \tag{1.17}$$

$$p_2'(x_1) = \frac{f(x_2) - f(x_0)}{2h} \tag{1.18}$$

$$p_2'(x_2) = \frac{f(x_0) - 4f(x_1) + 3f(x_2)}{2h}. \tag{1.19}$$

These beautiful formulas are *right-looking*, *central*, and *left-looking* approximations to f'. Though we used an interpolating polynomial to derive these formulas, those polynomials are now nowhere in sight: they are merely the scaffolding that lead to these formulas. How accurate are these formulas? Corollary 1.1 with $n = 2$ and $[a, b] = [x_0, x_2] = x_0 + [0, 2h]$ gives

These formulas can also be derived by strategically combining Taylor expansions for $f(x+h)$ and $f(x-h)$. That is an easier route to simple formulas like (1.18), but is less appealing when more sophisticated approximations like (1.17) and (1.19) (and beyond) are needed.

$$|f'(x_k) - p_2'(x_k)| \leq \left(\frac{2}{3} \max_{\xi \in [x_0, x_2]} |f'''(\xi)|\right) h^2. \tag{1.20}$$

Notice that these approximations indeed scale with h^2, rather than h, and so the quadratic interpolant leads to a much better approximation to f', at the cost of evaluating f at three points (for $f'(x_0)$ and $f'(x_2)$), rather than two.

Example 1.4 (Second derivative)**.** While we have only proved a bound for the error in the first derivative, $f'(x) - p'(x)$, you can see that similar bounds should hold when higher derivatives of p are used to approximate corresponding derivatives of f. Here we illustrate with the second derivative.

Since p_1 is linear, $p_1''(x) = 0$ for all x, and the linear interpolant will not lead to any meaningful bound on $f''(x)$. Thus, we focus on the quadratic interpolant to f at the three uniformly spaced points x_0, x_1, and x_2. Take two derivatives of the formula (1.16) for $p_2(x)$ to obtain

$$p_2''(x) = \frac{f(x_0) - 2f(x_1) + f(x_2)}{h^2}, \tag{1.21}$$

which is a famous approximation to the second derivative that is often used in the finite difference discretization of differential equations. One can show that, like the approximations $p_2'(x_k)$, this formula is accurate to order h^2.

Example 1.5 (*Mathematica* code for computing difference formulas)**.** *Code to follow....*

LECTURE 5: *Finite Difference Methods for Differential Equations*

1.7.3 Application: Boundary Value Problems

Example 1.6 (Dirichlet boundary conditions)**.** Suppose we want to solve the differential equation

$$-u''(x) = g(x), \qquad x \in [0,1]$$

for the unknown function u, subject to the *Dirichlet* boundary conditions

$$u(0) = u(1) = 0.$$

One common approach to such problems is to approximate the solution u on a uniform grid of points

$$0 = x_0 < x_1 < \cdots < x_n = 1$$

with $x_j = j/N$.

We seek to approximate the solution $u(x)$ at each of the grid points $x_0, \ldots, x_n$. The Dirichlet boundary conditions give the end values immediately:

$$u(x_0) = 0, \qquad u(x_n) = 0.$$

At each of the interior grid points, we require a local approximation of the equation

$$-u''(x_j) = g(x_j), \qquad j = 1, \ldots, n-1.$$

For each, we will (implicitly) construct the quadratic interpolant $p_{2,j}$ to $u(x)$ at the points x_{j-1}, x_j, and x_{j+1}, and then approximate

$$-p''_{2,j}(x_j) \approx -u''(x_j) = g(x_j).$$

Aside from some index shifting, we have already constructed $p''_{2,j}$ in equation (1.21):

$$p''_{2,j}(x) = \frac{u(x_{j-1}) - 2u(x_j) + u(x_{j+1})}{h^2}. \tag{1.22}$$

Just one small caveat remains: we cannot construct $p''_{2,j}(x)$, *because we do not know the values of* $u(x_{j-1})$, $u(x_j)$, *and* $u(x_{j+1})$: finding those values is the point of our entire endeavor. Thus we define approximate values

$$u_j \approx u(x_j), \qquad j = 1, \ldots, n-1.$$

and will instead use the polynomial $p_{2,j}$ that interpolates u_{j-1}, u_j, and u_{j+1}, giving

$$p''_{2,j}(x) = \frac{u_{j-1} - 2u_j + u_{j+1}}{h^2}. \tag{1.23}$$

Let us accumulate all our equations for $j = 0, \ldots, n$:

$$\begin{aligned}
u_0 &= 0 \\
u_0 - 2u_1 + u_2 &= -h^2 g(x_1) \\
u_1 - 2u_2 + u_3 &= -h^2 g(x_2) \\
&\vdots \\
u_{n-3} - 2u_{n-2} + u_{n-1} &= -h^2 g(x_{n-2}) \\
u_{n-2} - 2u_{n-1} + u_n &= -h^2 g(x_{n-1}) \\
u_n &= 0.
\end{aligned}$$

Notice that this is a system of $n+1$ linear equations in $n+1$ variables $u_0, \ldots, u_{n+1}$. Thus we can arrange this in matrix form as

$$\begin{bmatrix}
1 & & & & & & & \\
1 & -2 & 1 & & & & & \\
 & 1 & -2 & 1 & & & & \\
 & & \ddots & \ddots & \ddots & & & \\
 & & & 1 & -2 & 1 & & \\
 & & & & 1 & -2 & 1 & \\
 & & & & & & 1
\end{bmatrix}
\begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ \vdots \\ u_{n-2} \\ u_{n-1} \\ u_n \end{bmatrix}
=
\begin{bmatrix} 0 \\ -h^2 g(x_1) \\ -h^2 g(x_2) \\ \vdots \\ -h^2 g(x_{n-2}) \\ -h^2 g(x_{n-1}) \\ 0 \end{bmatrix}, \tag{1.24}$$

where the blank entries are zero. Notice that the first and last entries are trivial: $u_0 = u_n = 0$, and so we can trim them off to yield the slightly simpler matrix

$$\begin{bmatrix}
-2 & 1 & & & \\
1 & -2 & 1 & & \\
 & \ddots & \ddots & \ddots & \\
 & & 1 & -2 & 1 \\
 & & & 1 & -2
\end{bmatrix}
\begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_{n-2} \\ u_{n-1} \end{bmatrix}
=
\begin{bmatrix} -h^2 g(x_1) \\ -h^2 g(x_2) \\ \vdots \\ -h^2 g(x_{n-2}) \\ -h^2 g(x_{n-1}) \end{bmatrix}. \tag{1.25}$$

Solve this $(n-1) \times (n-1)$ linear system of equations using Gaussian elimination. One can show that the solution to the differential equation inherits the accuracy of the interpolant: the error $|u(x_j) - u_j|$ behaves like $O(h^2)$ as $h \to 0$.

Ideally, use an efficient version of Gaussian elimination that exploits the banded structure of this matrix to give a solution in $O(n)$ operations.

Example 1.7 (Mixed boundary conditions)**.** Modify the last example to keep the same differential equation

$$-u''(x) = g(x), \qquad x \in [0, 1]$$

but now use *mixed* boundary conditions,

$$u'(0) = u(1) = 0.$$

The derivative condition on the left is the only modification; we must change the first row of equation (1.24) to encode this condition. One might be tempted to use a simple linear interpolant to approximate the boundary condition on the left side, adapting formula (1.14) to give:

$$\frac{u_1 - u_0}{h} = 0. \tag{1.26}$$

This equation makes intuitive sense: it forces $u_1 = u_0$, so the approximate solution will have zero slope on its left end. This gives the equation

$$\begin{bmatrix} -1 & 1 & & & & & \\ 1 & -2 & 1 & & & & \\ & 1 & -2 & 1 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & 1 & -2 & 1 & \\ & & & & 1 & -2 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ \vdots \\ u_{n-2} \\ u_{n-1} \end{bmatrix} = \begin{bmatrix} 0 \\ -h^2 g(x_1) \\ -h^2 g(x_2) \\ \vdots \\ -h^2 g(x_{n-2}) \\ -h^2 g(x_{n-1}) \end{bmatrix}, \tag{1.27}$$

where we have trimmed off the elements associated with the $u_n = 0$.

The approximation of the second derivative (1.23) is accurate up to $\mathcal{O}(h^2)$, whereas the estimate (1.26) of $u'(0) = 0$ is only $\mathcal{O}(h)$ accurate. Will it matter if we compromise accuracy that little bit, if only in one of the n equations in (1.27)? What if instead we approximate $u'(0) = 0$ to second-order accuracy?

Equations (1.17)–(1.19) provide three formulas that approximate the first derivative to second order. Which one is appropriate in this setting? The *right-looking* formula (1.17) gives the approximation

$$u'(0) \approx \frac{-3u_0 + 4u_1 - u_2}{2h}, \tag{1.28}$$

which involves the variables u_0, u_1, and u_2 that we are already considering. In contrast, the centered formula (1.18) needs an estimate of $u(-h)$, and the left-looking formula (1.19) needs $u(-h)$ and $u(-2h)$. Since these values of u fall outside the domain $[0, 1]$ of u, the centered and left-looking formulas would not work.

Combining the right-looking formula (1.28) with the boundary condition $u'(0) = 0$ gives

$$\frac{-3u_0 + 4u_1 - u_2}{2h} = 0,$$

with which we replace the first row of (1.27) to obtain

$$\text{(1.29)}\qquad \begin{bmatrix} -3 & 4 & -1 & & & & \\ 1 & -2 & 1 & & & & \\ & 1 & -2 & 1 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & 1 & -2 & 1 & \\ & & & & 1 & -2 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ \vdots \\ u_{n-2} \\ u_{n-1} \end{bmatrix} = \begin{bmatrix} 0 \\ -h^2 g(x_1) \\ -h^2 g(x_2) \\ \vdots \\ -h^2 g(x_{n-2}) \\ -h^2 g(x_{n-1}) \end{bmatrix}.$$

Is this $\mathcal{O}(h^2)$ accurate approach at the boundary worth the (rather minimal) extra effort? Let us investigate with an example. Set the right-hand side of the differential equation to

$$g(x) = \cos(\pi x/2),$$

which corresponds to the exact solution

$$u(x) = \frac{4}{\pi^2}\cos(\pi x/2).$$

Verify that u satisfies the boundary conditions $u'(0) = 0$ and $u(1) = 0$.

Figure 1.10 compares the solutions obtained by solving (1.27) and (1.29) with $n = 4$. Clearly, the simple adjustment that gave the $\mathcal{O}(h^2)$ approximation to $u'(0) = 0$ makes quite a difference! This figures shows that the solutions from (1.27) and (1.29) differ, but plots like this are not the best way to understand how the approximations compare as $n \to \infty$. Instead, compute maximum error at the interpolation points,

$$\max_{0 \le j \le n} |u(x_j) - u_j|$$

Indeed, we used this small value of n because it is difficult to see the difference between the exact solution and the approximation from (1.29) for larger n.

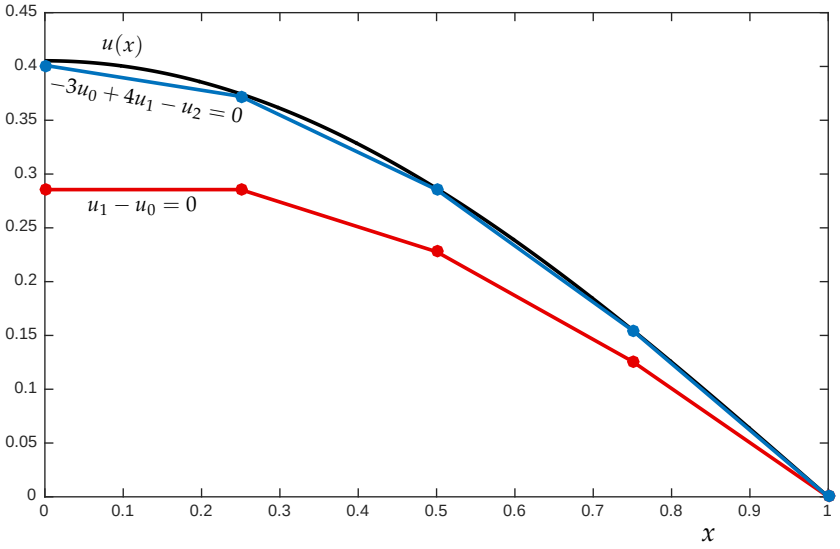


Figure 1.10: Approximate solutions to $-u''(x) = \cos(\pi x/2)$ with $u'(0) = u(1) = 0$. The black curve shows $u(x)$. The red approximation is obtained by solving (1.27), which uses the $\mathcal{O}(h)$ approximation $u'(0) = 0$; the blue approximation is from (1.29) with the $\mathcal{O}(h^2)$ approximation of $u'(0) = 0$. Both approximations use $n = 4$.

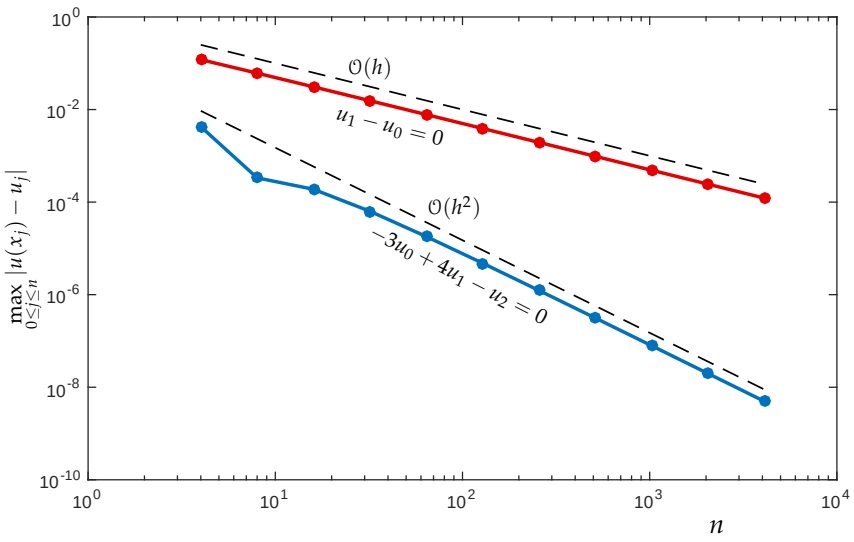


Figure 1.11: Convergence of approximate solutions to $-u''(x) = \cos(\pi x/2)$ with $u'(0) = u(1) = 0$. The red line shows the approximation from (1.27); it converges like $\mathcal{O}(h)$ as $h \to 0$. The blue line shows the approximation from (1.29), which converges like $\mathcal{O}(h^2)$.

for various values of n. Figure 1.11 shows the results of such experiments for $n = 2^2, 2^3, \ldots, 2^{12}$. Notice that this figure is a 'log-log' plot; on such a scale, the errors fall on straight lines, and from the slope of these lines one can determine the order of convergence. The slope of the red curve is -1, so the accuracy of the approximations from (1.27) is $\mathcal{O}(n^{-1}) = \mathcal{O}(h)$ accurate. The slope of the blue curve is -2, so (1.29) gives an $\mathcal{O}(n^{-2}) = \mathcal{O}(h^2)$ accurate approximation.

How large would n need to be, to get the same accuracy from the $\mathcal{O}(h)$ approximation that was produced by the $\mathcal{O}(h^2)$ approximation with $n = 2^{12} = 4096$? Extrapolation of the red curve suggests we would need roughly $n = 10^8$.

This example illustrates a general lesson: when constructing finite difference approximations to differential equations, one must ensure that the approximations to the boundary conditions have the same order of accuracy as the approximation of the differential equation itself. These formulas can be nicely constructed by from derivatives of polynomial interpolants of appropriate degree.

LECTURE 6: *Interpolating Derivatives*

1.8 Hermite Interpolation and Generalizations

Example 1.1 demonstrated that polynomial interpolants to $\sin(x)$ attain arbitrary accuracy for $x \in [-5, 5]$ as the polynomial degree increases, even if the interpolation points are taken exclusively from $[-1, 1]$. In fact, as $n \to \infty$ interpolants based on data from $[-1, 1]$ will converge to $\sin(x)$ *for all* $x \in \mathbb{R}$. More precisely, for any $x \in \mathbb{R}$ and any $\varepsilon > 0$, there exists some positive integer N such that $|\sin(x) - p_n(x)| < \varepsilon$ for all $n \geq N$, where p_n interpolates $\sin(x)$ at $n + 1$ uniformly-spaced interpolation points in $[-1, 1]$.

In fact, this is not as surprising as it might first appear. The Taylor series expansion uses derivative information at a single point to produce a polynomial approximation of f that is accurate at nearby points. In fact, the interpolation error bound derived in the previous lecture bears close resemblance to the remainder term in the Taylor series. If $f \in C^{(n+1)}[a, b]$, then expanding f at $x_0 \in (a, b)$, we have

$$f(x) = \sum_{k=0}^{n} \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k + \frac{f^{(n+1)}(\xi)}{(n+1)!}(x - x_0)^{n+1}$$

This is the *Lagrange form* of the error.

for some $\xi \in [x, x_0]$ that depends on x. The first sum is simply a degree n polynomial in x; from the final term – the Taylor remainder – we obtain the bound

$$\max_{x \in [a,b]} \left| f(x) - \left(\sum_{k=0}^{n} \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k \right) \right| \leq \left(\max_{\xi \in [a,b]} \frac{|f^{(n+1)}(\xi)|}{(n+1)!} \right) \left(\max_{x \in [a,b]} |x - x_0|^{n+1} \right),$$

which should certainly remind you of the interpolation error formula in Theorem 1.3.

One can view polynomial interpolation and Taylor series as two extreme approaches to approximating f: one uses global information, but only about f; the other uses only local information, but requires extensive knowledge of the derivatives of f. In this section we shall discuss an alternative based on the best features of each of these ideas: use global information about both f and its derivatives.

1.8.1 Hermite interpolation

In cases where the polynomial interpolants of the previous sections incurred large errors for some $x \in [a, b]$, one typically observes that the slope of the interpolant differs markedly from that of f at some of the interpolation points $\{x_j\}$. (Recall Runge's example in Figure 1.8.) Why not then *force the interpolant to match both f and f' at the interpolation points*?

Often the underlying application provides a motivation for such derivative matching. For example, if the interpolant approximates the position of a particle moving in space, we might wish the interpolant to match not only position, but also velocity. *Hermite interpolation* is the general procedure for constructing such interpolants.

Typically the position of a particle is given in terms of a second-order differential equation (in classical mechanics, arising from Newton's second law, $F = ma$). To solve this second-order ODE, one usually writes it as a system of first-order equations whose numerical solution we will study later in the semester. One component of the system is position, the other is velocity, and so one automatically obtains values for both f (position) and f' (velocity) simultaneously.

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

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

determine some $h_n \in \mathcal{P}_{2n+1}$ such that

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

Note that h must generally be a polynomial of degree $2n+1$ to have sufficiently many degrees of freedom to satisfy the $2n+2$ constraints. We begin by addressing the existence and uniqueness of this interpolant.

Existence is best addressed by explicitly constructing the desired polynomial. We adopt a variation of the *Lagrange approach* used in Section 1.5. We seek degree-$(2n+1)$ polynomials $\{A_k\}_{k=0}^n$ and $\{B_k\}_{k=0}^n$ such that

$$A_k(x_j) = \begin{cases} 0, & j \neq k, \\ 1, & j = k, \end{cases} \qquad A_k'(x_j) = 0 \text{ for } j = 0, \ldots, n;$$

$$B_k(x_j) = 0 \text{ for } j = 0, \ldots, n, \qquad B_k'(x_j) = \begin{cases} 0, & j \neq k \\ 1, & j = k \end{cases}.$$

These polynomials would yield a basis for $\mathcal{P}_{2n+1}$ in which h_n has a simple expansion:

$$h_n(x) = \sum_{k=0}^n f(x_k) A_k(x) + \sum_{k=0}^n f'(x_k) B_k(x). \tag{1.30}$$

To show how we can construct the polynomials A_k and B_k, we recall the Lagrange basis polynomials used for the standard interpolation problem,

$$\ell_k(x) = \prod_{j=0, j\neq k}^n \frac{(x - x_j)}{(x_k - x_j)}.$$

Consider the definitions

$$A_k(x) := \big(1 - 2(x - x_k)\ell_k'(x_k)\big)\ell_k^2(x),$$
$$B_k(x) := (x - x_k)\ell_k^2(x).$$

Note that since $\ell_k \in \mathcal{P}_n$, we have $A_k, B_k \in \mathcal{P}_{2n+1}$. Figure 1.12 shows these Hermite basis polynomials and their derivatives for $n = 5$ using

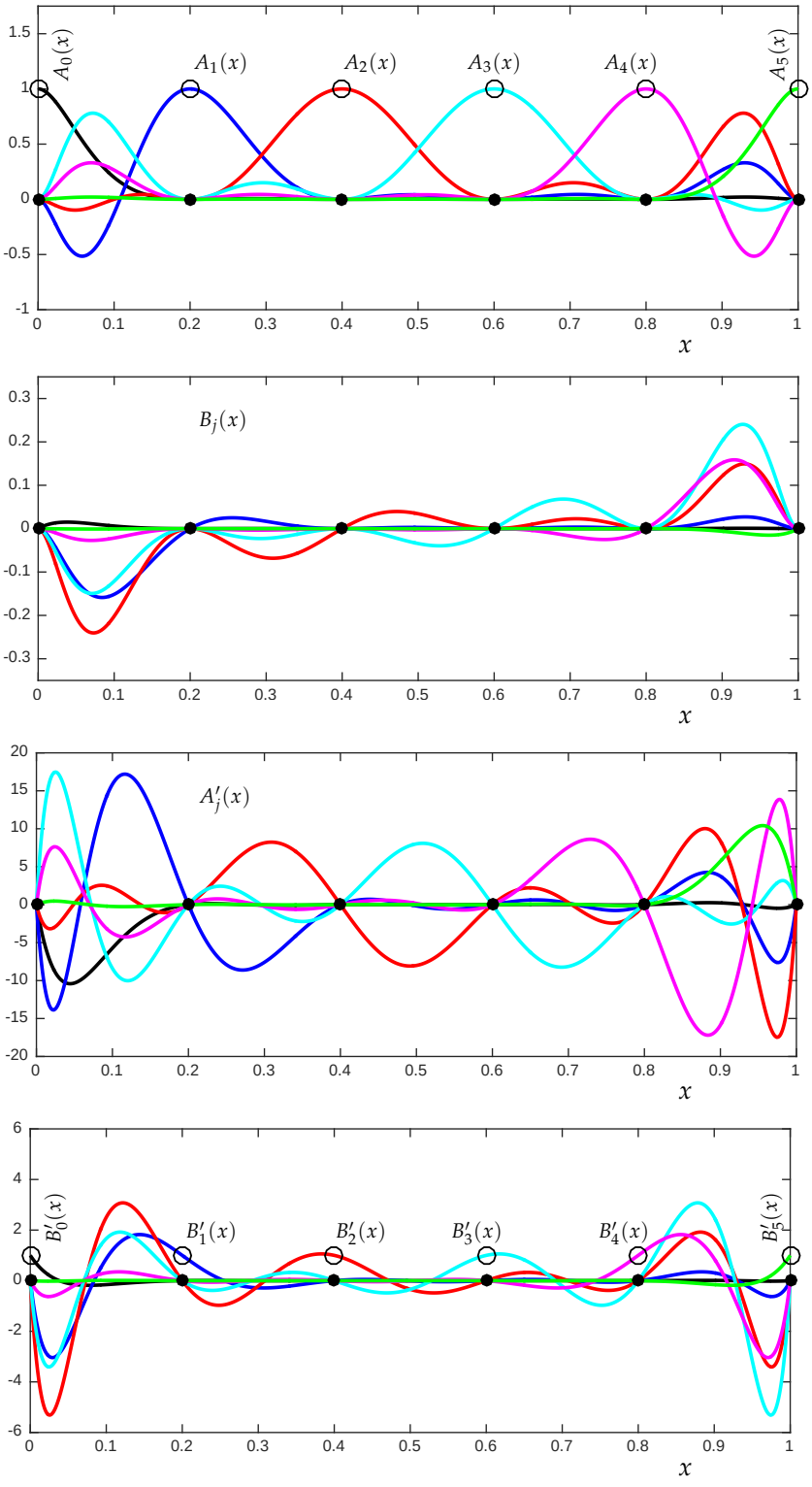


Figure 1.12: The Hermite basis polynomials for $n = 5$ on the interval $[a, b] = [0, 1]$ with $x_j = j/5$ (black dots).
- $A_0, \ldots, A_5$: $A_j(x_j) = 1$ (black circles).
- $B_0, \ldots, B_5$: $B_j(x_k) = 0$ for all j, k.
- $A'_0, \ldots, A'_5$: $A'_j(x_k) = 0$ for all j, k.
- $B'_0, \ldots, B'_5$: $B'_j(x_j) = 1$ (black circles).

uniformly spaced points on $[0, 1]$. It is a straightforward exercise to verify that these A_k and B_k, and their first derivatives, obtain the specified values at $\{x_j\}_{j=0}^n$.

These interpolation conditions at the points $\{x_j\}$ ensure that the $2n + 2$ polynomials $\{A_k, B_k\}_{k=0}^n$, each of degree $2n + 1$, form a basis for $\mathcal{P}_{2n+1}$, and thus we can always write h_n via the formula (1.30).

Figure 1.13 compares the standard polynomial interpolant $p_n \in \mathcal{P}_n$ to the Hermite interpolant $h_n \in \mathcal{P}_{2n+1}$ and the standard interpolant of the same degree, $p_{2n+1} \in \mathcal{P}_{2n+1}$ for the example $f(x) = \sin(20x) + e^{5x/2}$ using uniformly spaced points on $[0, 1]$ with $n = 5$. Note the distinction between h_n and p_{2n+1}, which are both polynomials of the same degree.

Here are a couple of basic results whose proofs follow the same techniques as the analogous proofs for the standard interpolation problem.

Theorem 1.7. The Hermite interpolant $h_n \in \mathcal{P}_{2n+1}$ is unique.

The uniqueness result hinges on the fact that we interpolate f and f' both at all interpolation points. If we vary the number of derivatives interpolated at each data point, we open the possibility of non-unique interpolants.

Theorem 1.8. Suppose $f \in C^{2n+2}[x_0, x_n]$ and let $h_n \in \mathcal{P}_{2n+1}$ such that $h_n(x_j) = f(x_j)$ and $h_n'(x_j) = f'(x_j)$ for $j = 0, \ldots, n$. Then for any $x \in [x_0, x_n]$, there exists some $\xi \in [x_0, x_n]$ such that

$$f(x) - h_n(x) = \frac{f^{(2n+2)}(\xi)}{(2n+2)!} \prod_{j=0}^{n} (x - x_j)^2.$$

The proof of this latter result is directly analogous to the standard polynomial interpolation error in Theorem 1.3. Think about how you would prove this result for yourself.

Hint: the proof has some resemblance to our proof of Theorem 1.6. Invoke Rolle's theorem to get n roots of a certain function, then use the derivative interpolation to get another $n + 1$ roots.

1.8.2 *Hermite–Birkhoff interpolation*

Of course, one need not stop at interpolating f and f'. Perhaps your application has more general requirements, where you want to interpolate higher derivatives, too, or have the number of derivatives interpolated differ at each interpolation point. Such general polynomials are called *Hermite–Birkhoff interpolants*, and you already have the tools at your disposal to compute them. Simply formulate the problem as a linear system and find the desired coefficients, but beware that in some situations, *there may be infinitely many polynomials that satisfy the interpolation conditions*. For these problems, it is generally simplest to work with the monomial basis, though one could design Newton- or Lagrange-inspired bases for particular situations.

For example, suppose you seek an interpolant that is particularly accurate in the vicinity of one of the interpolation points, and so you wish to interpolate higher derivatives at that point: a hybrid between an interpolating polynomial and a Taylor expansion.

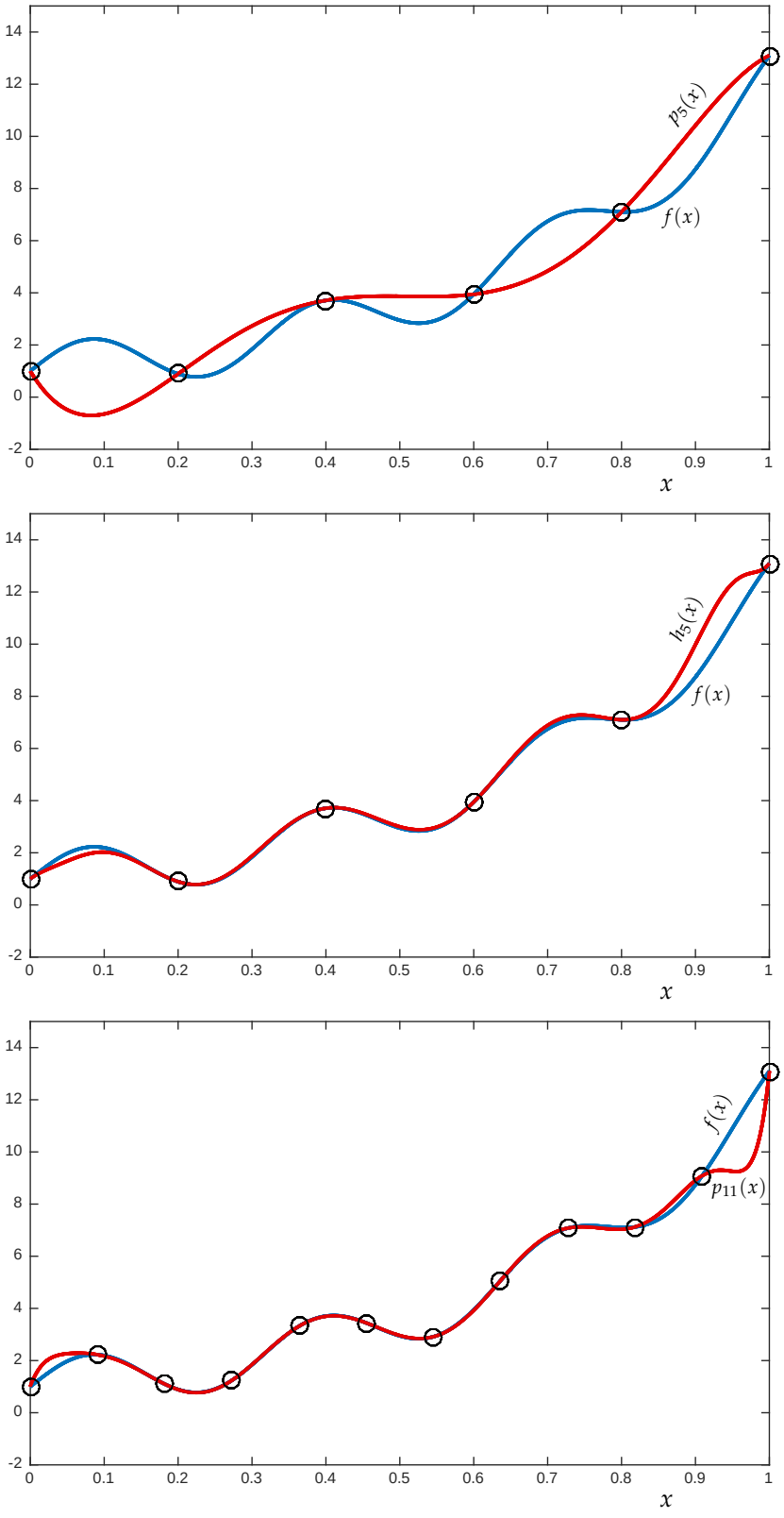


Figure 1.13: Interpolation of $f(x) = \sin(20x) + e^{5x/2}$ at uniformly spaced points for $x \in [0, 1]$. Top plot: the standard polynomial interpolant $p_5 \in \mathcal{P}_5$. Middle plot: the Hermite interpolant $h_5 \in \mathcal{P}_{11}$. Bottom plot: the standard interpolant $p_{11} \in \mathcal{P}_{11}$.

Though the last two plots show polynomials of the same degree, notice how the interpolants differ. (At first glance it appears the Hermite interpolation condition fails at the rightmost point in the middle plot; zoom in to see that the slope of the interpolant indeed matches $f'(1)$.)

1.8.3 *Hermite–Fejér interpolation*

Another Hermite-like scheme initially sounds like a bad idea: make the interpolant have *zero slope* at all the interpolation points.

> Given $f \in C^1[a,b]$ and $n+1$ points $\{x_j\}_{j=0}^n$ satisfying
>
> $$a \le x_0 < x_1 < \cdots < x_n \le b,$$
>
> determine some $h_n \in \mathcal{P}_{2n+1}$ such that
>
> $$h_n(x_j) = f(x_j), \qquad h_n'(x_j) = 0 \quad \text{for } j = 0, \dots, n.$$

That is, explicitly construct h_n such that its derivatives in general *do not match those of* f. This method, called *Hermite–Fejér interpolation*, turns out to be remarkably effective, even better than standard Hermite interpolation in certain circumstances. In fact, Fejér proved that if we choose the interpolation points $\{x_j\}$ in the right way, h_n is guaranteed to converge to f uniformly as $n \to \infty$.

Theorem 1.9. For each $n \ge 1$, let h_n be the Hermite–Fejér interpolant of $f \in C[a,b]$ at the Chebyshev interpolation points

$$x_j = \frac{a+b}{2} + \Big(\frac{b-a}{2}\Big)\cos\Big(\frac{(2j+1)\pi}{2n+2}\Big), \quad j = 0, \dots, n.$$

Then $h_n(x)$ converges uniformly to f on $[a,b]$.

For a proof of Theorem 1.9, see page 57 of I. P. Natanson, *Constructive Function Theory*, vol. 3 (Ungar, 1965).

LECTURE 7: *Trigonometric Interpolation*

1.9 Trigonometric interpolation for periodic functions

Thus far all our interpolation schemes have been based on polynomials. However, if the function f is *periodic*, one might naturally prefer to interpolate f with some set of periodic functions.

To be concrete, suppose we have a continuous 2π-periodic function f that we wish to interpolate at the uniformly spaced points $x_k = 2\pi k/n$ for $k = 0, \ldots, n$ with $n = 5$. We shall build an interpolant as a linear combination of the 2π-periodic functions

'2π-periodic' means that f is continuous throughout $\mathbb{R}$ and $f(x) = f(x + 2\pi)$ for all $x \in \mathbb{R}$.

The choice of period 2π makes the notation a bit simpler, but the idea can be easily adapted for any period.

$$b_0(x) = 1, \quad b_1(x) = \sin(x), \quad b_2(x) = \cos(x), \quad b_3(x) = \sin(2x), \quad b_4(x) = \cos(2x).$$

Note that we have *six* interpolation conditions at x_k for $k = 0, \ldots, 5$, but only *five* basis functions. This is not a problem: since f is periodic, $f(x_0) = f(x_5)$, and the same will be true of our 2π-periodic interpolant: the last interpolation condition is automatically satisfied.

We shall construct an interpolant of the form

$$t_5(x) = \sum_{k=0}^{4} c_k b_k(x)$$

such that

$$t_5(x_j) = f(x_j), \quad j = 0, \ldots, 4.$$

To compute the unknown coefficients $c_0, \ldots, c_4$, set up a linear system as usual,

$$\begin{bmatrix} b_0(x_0) & b_1(x_0) & b_2(x_0) & b_3(x_0) & b_4(x_0) \\ b_0(x_1) & b_1(x_1) & b_2(x_1) & b_3(x_1) & b_4(x_1) \\ b_0(x_2) & b_1(x_2) & b_2(x_2) & b_3(x_2) & b_4(x_2) \\ b_0(x_3) & b_1(x_3) & b_2(x_3) & b_3(x_3) & b_4(x_3) \\ b_0(x_4) & b_1(x_4) & b_2(x_4) & b_3(x_4) & b_4(x_4) \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} = \begin{bmatrix} f(x_0) \\ f(x_1) \\ f(x_2) \\ f(x_3) \\ f(x_4) \end{bmatrix},$$

which can be readily generalized to accommodate more interpolation points. We could solve this system for $c_0, \ldots, c_n$, but we prefer to express the problem in a more convenient basis for the trigonometric functions. Recall Euler's formula,

You would $b_6(x) = \sin(3x)$, $b_7(x) = \cos(3x)$, etc.: one function for each additional interpolation point. Generally you would use an odd value of n, to include pairs of sines and cosines.

$$e^{i\theta x} = \cos(\theta x) + i \sin(\theta x),$$

which also implies that

$$e^{-i\theta x} = \cos(\theta x) - i \sin(\theta x).$$

To prove this, write the Taylor expansion of $e^{i\theta x}$, then separate the real and imaginary components to give Taylor expansions for $\cos(\theta x)$ and $\sin(\theta x)$.

From these formulas it follows that

$$\operatorname{span}\{e^{i\theta x}, e^{-i\theta x}\} = \{\cos(\theta x), \sin(\theta x)\}.$$

Note that we can also write $b_0(x) \equiv 1 = e^{i0x}$. Putting these pieces together, we arrive at an alternative basis for the trigonometric interpolation space:

$$\text{span}\{1, \sin(x), \cos(x), \sin(2x), \cos(2x)\} = \text{span}\{e^{-2ix}, e^{-ix}, e^{0ix}, e^{ix}, e^{2ix}\}.$$

The interpolant t_n can thus be expressed in the form

$$t_4(x) = \sum_{k=-2}^{2} \gamma_k e^{ikx} = \sum_{k=-2}^{2} \gamma_k (e^{ix})^k.$$

This last sum is written in a manner that emphasizes that t_4 is *a polynomial in the variable* e^{ix}, and hence t_n is a *trigonometric polynomial*. In this basis, the interpolation conditions give the linear system

$$\begin{bmatrix} e^{-2ix_0} & e^{-ix_0} & e^{0ix_0} & e^{ix_0} & e^{i2x_0} \\ e^{-2ix_1} & e^{-ix_1} & e^{0ix_1} & e^{ix_1} & e^{i2x_1} \\ e^{-2ix_2} & e^{-ix_2} & e^{0ix_2} & e^{ix_2} & e^{i2x_2} \\ e^{-2ix_3} & e^{-ix_3} & e^{0ix_3} & e^{ix_3} & e^{i2x_3} \\ e^{-2ix_4} & e^{-ix_4} & e^{0ix_4} & e^{ix_4} & e^{i2x_4} \end{bmatrix} \begin{bmatrix} \gamma_{-2} \\ \gamma_{-1} \\ \gamma_0 \\ \gamma_1 \\ \gamma_2 \end{bmatrix} = \begin{bmatrix} f(x_0) \\ f(x_1) \\ f(x_2) \\ f(x_3) \\ f(x_4) \end{bmatrix},$$

again with the natural generalization to larger odd integers n. At first blush this matrix looks no simpler than the one we first encountered, but a fascinating structure lurks. Notice that a generic entry of this matrix has the form $e^{\ell i x_k}$ for $\ell = -(n-1)/2, \ldots, (n-1)/2$ and $k = 0, \ldots, n-1$. Since $x_k = 2\pi k/n$, rewrite this entry as

$$e^{\ell i x_k} = \left(e^{ix_k}\right)^\ell = \left(e^{2\pi i k/n}\right)^\ell = \left(e^{2\pi i/n}\right)^{k\ell} = \omega^{k\ell},$$

where $\omega = e^{2\pi i/n}$ is an *nth root of unity*. In the $n = 5$ case, the linear system can thus be written as

This name comes from the fact that $\omega^n = 1$.

$$\begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^{-2} & \omega^{-1} & \omega^0 & \omega^1 & \omega^2 \\ \omega^{-4} & \omega^{-2} & \omega^0 & \omega^2 & \omega^4 \\ \omega^{-6} & \omega^{-3} & \omega^0 & \omega^3 & \omega^6 \\ \omega^{-8} & \omega^{-4} & \omega^0 & \omega^4 & \omega^8 \end{bmatrix} \begin{bmatrix} \gamma_{-2} \\ \gamma_{-1} \\ \gamma_0 \\ \gamma_1 \\ \gamma_2 \end{bmatrix} = \begin{bmatrix} f(x_0) \\ f(x_1) \\ f(x_2) \\ f(x_3) \\ f(x_4) \end{bmatrix}. \tag{1.31}$$

Denote this system by $\mathbf{F}\boldsymbol{\gamma} = \mathbf{f}$. Notice that each column of $\mathbf{F}$ equals some (entrywise) power of the vector

$$\begin{bmatrix} \omega^0 \\ \omega^1 \\ \omega^2 \\ \omega^3 \\ \omega^4 \end{bmatrix}.$$

In other words, *the matrix* $\mathbf{F}$ *has Vandermonde structure*. From our past experience with polynomial fitting addressed in Section 1.2.1, we

might fear that this formulation is ill-suited to numerical computations, i.e., solutions γ to the system $\mathbf{F}\gamma = \mathbf{f}$ could be polluted by large numerical errors.

In the language of numerical linear algebra, we might fear that the matrix $\mathbf{F}$ is *ill-conditioned*, i.e., the *condition number* $\|\mathbf{F}\|\,\|\mathbf{F}^{-1}\|$ is large.

Before jumping to this conclusion, examine $\mathbf{F}^*\mathbf{F}$. To form $\mathbf{F}^*$ note that $\overline{\omega^{-\ell}} = \omega^{\ell}$, so

$\mathbf{F}^*$ is the conjugate-transpose of $\mathbf{F}$:

$$\mathbf{F}^* = \overline{\mathbf{F}}^{\mathrm{T}},$$

so $(\mathbf{F}^*)_{j,k} = \overline{\mathbf{F}_{k,j}}$.

$$\mathbf{F}^*\mathbf{F} = \begin{bmatrix} \omega^0 & \omega^2 & \omega^4 & \omega^6 & \omega^8 \\ \omega^0 & \omega^1 & \omega^2 & \omega^3 & \omega^4 \\ \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^{-1} & \omega^{-2} & \omega^{-3} & \omega^{-4} \\ \omega^0 & \omega^{-2} & \omega^{-4} & \omega^{-6} & \omega^{-8} \end{bmatrix} \begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^{-2} & \omega^{-1} & \omega^0 & \omega^1 & \omega^2 \\ \omega^{-4} & \omega^{-2} & \omega^0 & \omega^2 & \omega^4 \\ \omega^{-6} & \omega^{-3} & \omega^0 & \omega^3 & \omega^6 \\ \omega^{-8} & \omega^{-4} & \omega^0 & \omega^4 & \omega^8 \end{bmatrix}.$$

The (ℓ, k) entry for $\mathbf{F}^*\mathbf{F}$ thus takes the form

$$(\mathbf{F}^*\mathbf{F})_{\ell,k} = \omega^0 + \omega^{(k-\ell)} + \omega^{2(k-\ell)} + \omega^{3(k-\ell)} + \omega^{4(k-\ell)}.$$

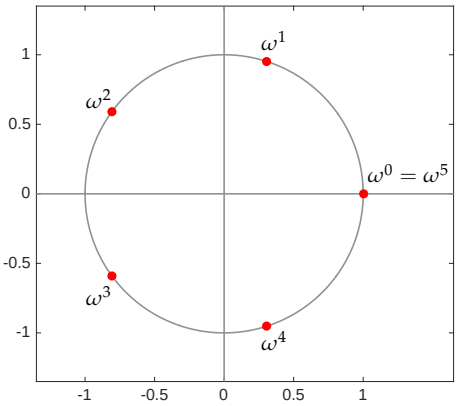
On the diagonal, when $\ell = k$, we simply have

$$(\mathbf{F}^*\mathbf{F})_{k,k} = \omega^0 + \omega^0 + \omega^0 + \omega^0 + \omega^0 = n.$$

On the off-diagonal, use $\omega^n = 1$ to see that all the off diagonal entries simplify to

$$(\mathbf{F}^*\mathbf{F})_{\ell,k} = \omega^0 + \omega^1 + \omega^2 + \omega^3 + \omega^4, \qquad \ell \neq k.$$

You can think of this last entry as n times the average of ω^0, ω^1, ω^2, ω^3, and ω^4, which are uniformly spaced points on the unit circle, shown in the plot to the right.



As these points are uniformly positioned about the unit circle, their mean must be zero, and hence

$$(\mathbf{F}^*\mathbf{F})_{\ell,k} = 0, \qquad \ell \neq k.$$

We thus must conclude that

$$\mathbf{F}^*\mathbf{F} = n\mathbf{I},$$

thus giving a formula for the inverse:

$$\mathbf{F}^{-1} = \frac{1}{n}\mathbf{F}^*.$$

The system $\mathbf{F}\gamma = \mathbf{f}$ can be immediately solved without the need for any factorization of $\mathbf{F}$:

$$\gamma = \frac{1}{n}\mathbf{F}^*\mathbf{f}.$$

The ready formula for $\mathbf{F}^{-1}$ is reminiscent of a *unitary* matrix. In fact, the matrices

$$\frac{1}{\sqrt{n}}\mathbf{F} \quad \text{and} \quad \frac{1}{\sqrt{n}}\mathbf{F}^*$$

$\mathbf{Q} \in \mathbb{C}^{n\times n}$ is unitary if and only if $\mathbf{Q}^{-1} = \mathbf{Q}^*$, or, equivalently, $\mathbf{Q}^*\mathbf{Q} = \mathbf{I}$.

are indeed unitary, and hence $\|n^{-1/2}\mathbf{F}\|_2 = \|n^{-1/2}\mathbf{F}^*\|_2 = 1$.

From this we can compute the condition number of $\mathbf{F}$:

$$\|\mathbf{F}\|_2\|\mathbf{F}^{-1}\|_2 = \frac{1}{n}\|\mathbf{F}\|_2\|\mathbf{F}^*\|_2 = \|n^{-1/2}\mathbf{F}\|_2\|n^{-1/2}\mathbf{F}^*\|_2 = 1.$$

This special Vandermonde matrix is perfectly conditioned! One can easily solve the system $\mathbf{F}\gamma = \mathbf{f}$ to high precision. The key distinction between this case and standard polynomial interpolation is that now we have a Vandermonde matrix based on *points* e^{ix_k} *that are equally spaced about the unit circle in the complex plane*, whereas before our points were distributed over an interval on the real line. This distinction makes all the difference between an unstable matrix equation and one that is not only perfectly stable, but also forms the cornerstone of modern signal processing.

In fact, we have just computed the 'Discrete Fourier Transform' (DFT) of the data vector

$$\begin{bmatrix} f(x_0) \\ f(x_1) \\ \vdots \\ f(x_{n-1}) \end{bmatrix}.$$

The coefficients $\gamma_{-(n-1)/2}, \ldots, \gamma_{(n-1)/2}$ that make up the vector

$$\gamma = \frac{1}{n}\mathbf{F}^*\mathbf{f}$$

are the *discrete Fourier coefficients* of the data in $\mathbf{f}$. From where does this name derive?

1.9.1 Connection to Fourier series

In a real analysis course, one learns that a 2π-periodic function f can be written as the Fourier series

$$f(x) = \sum_{k=-\infty}^{\infty} c_k \mathrm{e}^{\mathrm{i}kx},$$

where the *Fourier coefficients* c_ℓ are defined via

$$c_k := \frac{1}{2\pi}\int_0^{2\pi} f(x)\mathrm{e}^{-\mathrm{i}kx}.$$

Notice that $\gamma_k = ((1/n)\mathbf{F}^*\mathbf{f})_k$ is an approximation to this c_k:

$$\begin{aligned}\gamma_k &= \frac{1}{n}\sum_{\ell=0}^{n-1} f(x_\ell)\omega^{-\ell k} \\ &= \frac{1}{n}\sum_{k=0}^{n-1} f(x_k)\mathrm{e}^{-(2\pi k/n)\mathrm{i}\ell} = \frac{1}{n}\sum_{k=0}^{n-1} f(x_k)\mathrm{e}^{-\mathrm{i}\ell x_k}.\end{aligned}$$

The matrix 2-norm is defined as

$$\|\mathbf{F}\|_2 = \max_{\mathbf{x}\neq\mathbf{0}} \frac{\|\mathbf{F}\mathbf{x}\|_2}{\|\mathbf{x}\|_2},$$

where the vector norm on the right hand side is the Euclidean norm

$$\|\mathbf{y}\|_2 = \Big(\sum_k |y_k|^2\Big)^{1/2} = (\mathbf{y}^*\mathbf{y})^{1/2}.$$

The 2-norm of a unitary matrix is one: If $\mathbf{Q}^*\mathbf{Q} = \mathbf{I}$, then

$$\|\mathbf{Q}\mathbf{x}\|_2^2 = \mathbf{x}^*\mathbf{Q}^*\mathbf{Q}\mathbf{x} = \mathbf{x}^*\mathbf{x} = \|\mathbf{x}\|^2,$$

so $\|\mathbf{Q}\|_2 = 1$.

To ensure pointwise convergence of this series for all $x \in [0, 2\pi]$, f must be a continuous 2π-periodic function with a continuous first derivative. The functions $e_k(x) = \mathrm{e}^{\mathrm{i}kx}/\sqrt{2\pi}$ form an orthonormal basis for the space $L^2[0, 2\pi]$ with the inner product

$$(f, g) = \int_0^{2\pi} f(x)\overline{g(x)}\,\mathrm{d}x.$$

The Fourier series is simply an expansion of f in this basis: $f = \sum_k (f, e_k)\, e_k$.

Now use the fact that $f(x_0)e^{-i\ell x_0} = f(x_n)e^{-i\ell x_n}$ to view the last sum as a *composite trapezoid rule* approximation of an integral:

The composite trapezoid rule will be discussed in Chapter 3.

$$\begin{aligned} 2\pi\gamma_\ell &= \frac{2\pi}{n}\left(\frac{1}{2}f(x_0)e^{-i\ell x_0} + \sum_{k=1}^{n-1} f(x_k)e^{-i\ell x_k} + \frac{1}{2}f(x_n)e^{-i\ell x_n}\right) \\ &\approx \int_0^{2\pi} f(x)e^{-i\ell x}\,dx \\ &= 2\pi c_\ell. \end{aligned}$$

The coefficient γ_ℓ that premultiplies $e^{i\ell x}$ in the trigonometric interpolating polynomial is actually an approximation of the Fourier coefficient c_ℓ.

Let us go one step further. Notice that the trigonometric interpolant

$$t_n(x) = \sum_{k=-(n-1)/2}^{(n-1)/2} \gamma_k e^{ikx}$$

is an approximation to the Fourier series

$$f(x) = \sum_{k=-\infty}^{\infty} c_k\, e^{ikx}$$

obtained by (1) truncating the series, and (2) replacing c_k with γ_k. To assess the quality of the approximation, we need to understand the magnitude of the terms dropped from the sum, as well as the accuracy of the composite trapezoid rule approximation γ_k to c_k. We will thus postpone discussion of $f(x) - t_n(x)$ until we develop a few more analytical tools in the next two chapters.

1.9.2 *Computing the discrete Fourier coefficients*

Normally we would require $O(n^2)$ operations to compute these coefficients using matrix-vector multiplication with $\mathbf{F}^*$, but Cooley and Tukey discovered in 1965 that given the amazing structure in $\mathbf{F}^*$, one can arrange operations so as to compute $\boldsymbol{\gamma} = n^{-1}\mathbf{F}^*\mathbf{f}$ in only $O(n \log n)$ operations: a procedure that we now famously call the *Fast Fourier Transform* (FFT).

Apparently the FFT was discovered earlier by Gauss, but it was forgotten, given its limited utility before the advent of automatic computation. Jack Good (Bletchley Park codebreaker and, later, a Virginia Tech statistician) published a similar idea in 1958. Good recalls: 'John Tukey (December 1956) and Richard L. Garwin (September 1957) visited Cheltenham and I had them round to steaks and fries on separate occasions. I told Tukey briefly about my FFT (with little detail) and, in Cooley and Tukey's well known paper of 1965, my 1958 paper is the only citation.' See D. L. Banks, 'A conversation with I. J. Good,' *Stat. Sci.* 11 (1996) 1–19.

We can summarize this section as follows.

> The FFT of a vector of uniform samples of a 2π-periodic function f gives the coefficients for the trigonometric interpolant to f at those sample points. These coefficients approximate the function's Fourier coefficients.

Example 1.8 (Trig interpolation of a smooth periodic function)**.** Figure 1.14 shows the degree $n = 5, 7, 9$ and 11 trigonometric interpolants to the 2π-periodic function $f(x) = e^{\cos(x)+\sin(2x)}$. Notice that although all the interpolation points are all drawn from the interval $[0, 2\pi)$ (indicated by the gray region on the plot), the interpolants are just as accurate outside this region. In contrast, a standard polynomial fit through the same points will behave very differently: (non-constant) polynomials must satisfy $|p_n(x)| \to \infty$ as $|x| \to \infty$. Figure 1.15 shows this behavior for $n = 7$: for $x \in [0, 2\pi]$, the polynomial fit to f is about as accurate as the $n = 7$ trigonometric polynomial in Figure 1.14. Outside of $[0, 2\pi]$,the polynomial is much worse.

Example 1.9 (Trig interpolation of non-smooth function)**.** Figure 1.15 shows that a standard (non-perioidic) polynomial fit to a periodic function can yield a good approximation, at least over the interval from which the interpolation points are drawn. Now turn the tables: how well does a (periodic) trigonometric polynomial fit a smooth but non-periodic function? Simply take $f(x) = x$ on $[0, 2\pi]$, and construct the trigonometric interpolant as described above for $n = 11$. The top plot in Figure 1.16 shows that t_{11} gives a very poor approximation to f, constrained by design to be periodic even though f is not. The fact that $f(2\pi) \neq f(0)$ acts like a discontinuity, vastly impairing the quality of the approximation. The bottom plot in Figure 1.16 repeats this exercise for $f(x) = (x - \pi)^2$. Since in this case $f(0) = f(2\pi)$ we might expect better results; indeed, the approximation looks quite reasonable. Note, however, that $f'(0) \neq f'(\pi)$, and this lack of smoothness severely slows convergence of t_n to f as $n \to \infty$. Figure 1.17 contrasts this slow rate of convergence with the much faster convergence observed for $f(x) = e^{\cos(x)+\sin(2x)}$ used in Figure 1.14. Clearly the periodic interpolant is much better suited to smooth f.

1.9.3 Fast MATLAB implementation

MATLAB organizes its Fast Fourier Transform is a slightly different fashion than we have described above. To fit with MATLAB, reorder the unknowns in the system (1.31) to obtain

$$\begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \omega^{-2} & \omega^{-1} \\ \omega^0 & \omega^2 & \omega^4 & \omega^{-4} & \omega^{-2} \\ \omega^0 & \omega^3 & \omega^6 & \omega^{-6} & \omega^{-3} \\ \omega^0 & \omega^4 & \omega^8 & \omega^{-8} & \omega^{-4} \end{bmatrix} \begin{bmatrix} \gamma_0 \\ \gamma_1 \\ \gamma_2 \\ \gamma_{-2} \\ \gamma_{-1} \end{bmatrix} = \begin{bmatrix} f(x_0) \\ f(x_1) \\ f(x_2) \\ f(x_3) \\ f(x_4) \end{bmatrix}, \tag{1.32}$$

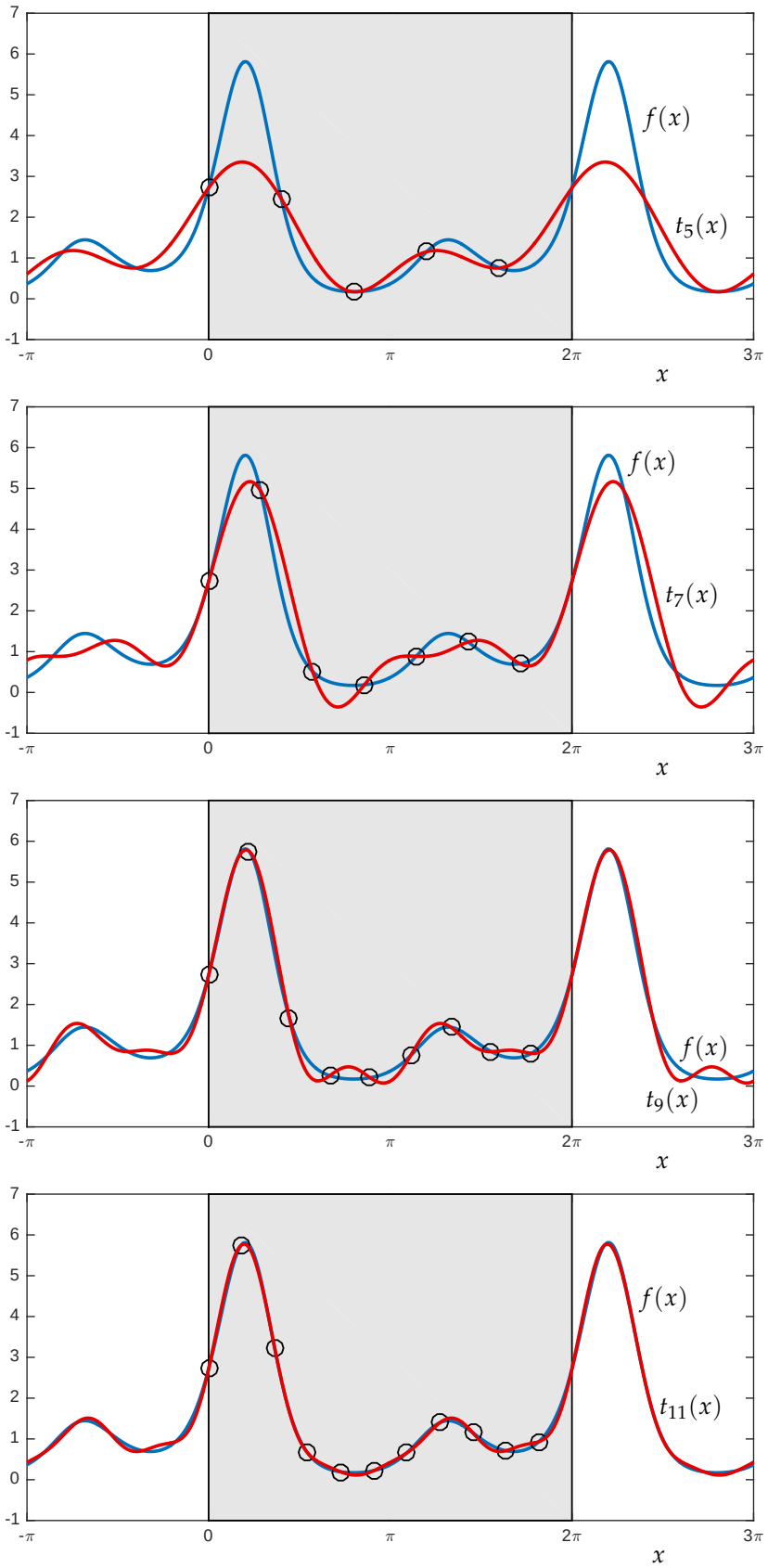


Figure 1.14: Trigonometric interpolant to 2π-periodic function $f(x) = e^{\cos(x)+\sin(2x)}$, using $n = 5, 7, 9$ and 11 points uniformly spaced over $[0, 2\pi)$ ($\{x_k\}_{k=0}^{n}$ for $x_k = 2\pi k/n$). Since both f and the interpolant are periodic, the function fits well throughout $\mathbb{R}$, not just on the interval for which the interpolant was designed.

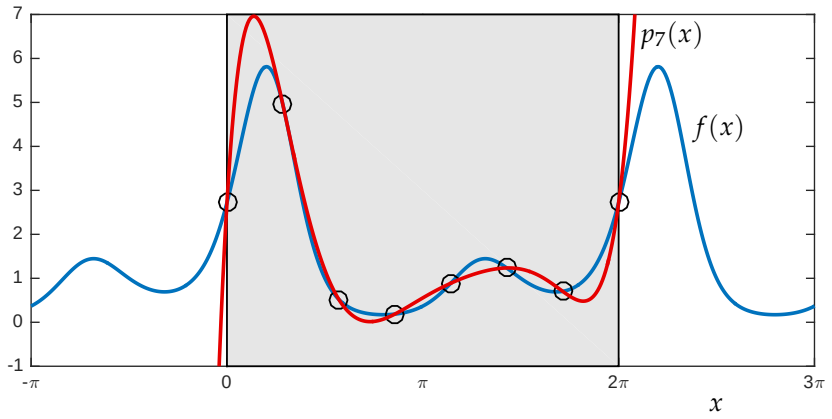


Figure 1.15: Polynomial fit of degree $n = 7$ through uniformly spaced grid points $x_0, \ldots, x_n$ for $x_j = 2\pi j/n$, for the same function $f(x) = e^{\cos(x)+\sin(2x)}$ used in Figure 1.14. In contrast to the trigonometric fits in the earlier figure, the polynomial grows very rapidly outside the interval $[0, 2\pi]$. Moral: if your function is periodic, fit it with a trigonometric polynomial.

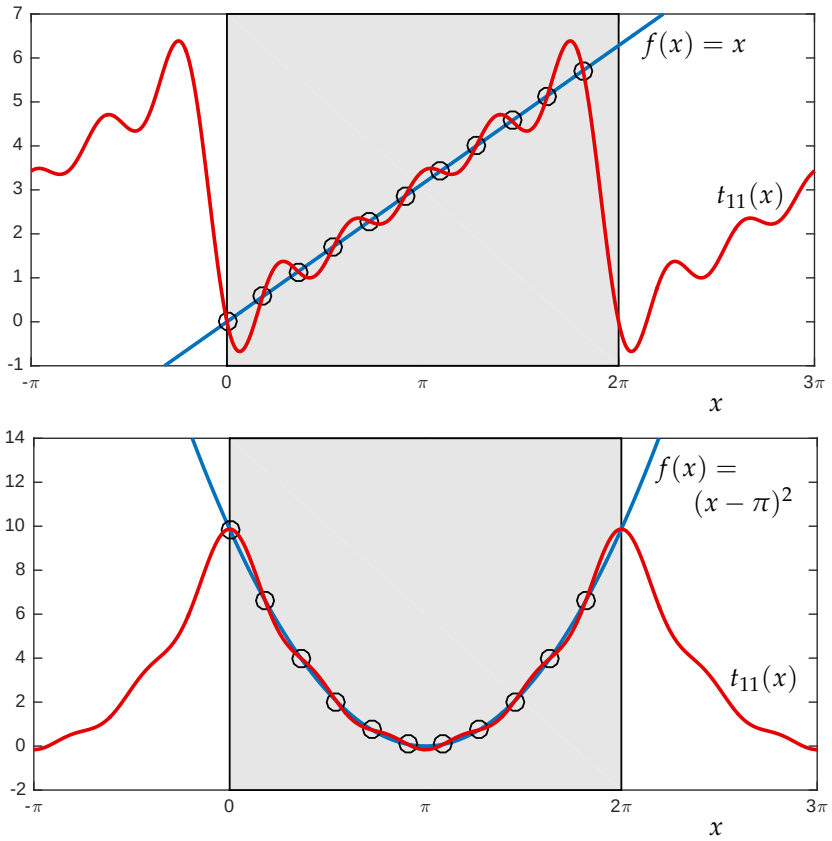


Figure 1.16: Trigonometric polynomial fit of degree $n = 11$ through uniformly spaced grid points $x_0, \ldots, x_n$ for $x_j = 2\pi j/n$, for the *non-periodic* function $f(x) = x$ (top) and for $f(x) = (x-\pi)^2$ (bottom). By restricting the latter function to the domain $[0, 2\pi]$, one can view it as a continuous periodic function with a jump discontinuity in the first derivative. The interpolant t_{11} seems to give a good approximation to f, but the discontinuity in the derivative slows the convergence of t_n to f as $n \to \infty$.

which amounts to reordering the *columns* of the matrix in (1.31). You can obtain this matrix by the command `ifft(eye(n))`. For $n = 5$,

$$5 * \texttt{ifft(eye(5))} = \begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \omega^{-2} & \omega^{-1} \\ \omega^0 & \omega^2 & \omega^4 & \omega^{-4} & \omega^{-2} \\ \omega^0 & \omega^3 & \omega^6 & \omega^{-6} & \omega^{-3} \\ \omega^0 & \omega^4 & \omega^8 & \omega^{-8} & \omega^{-4} \end{bmatrix}.$$

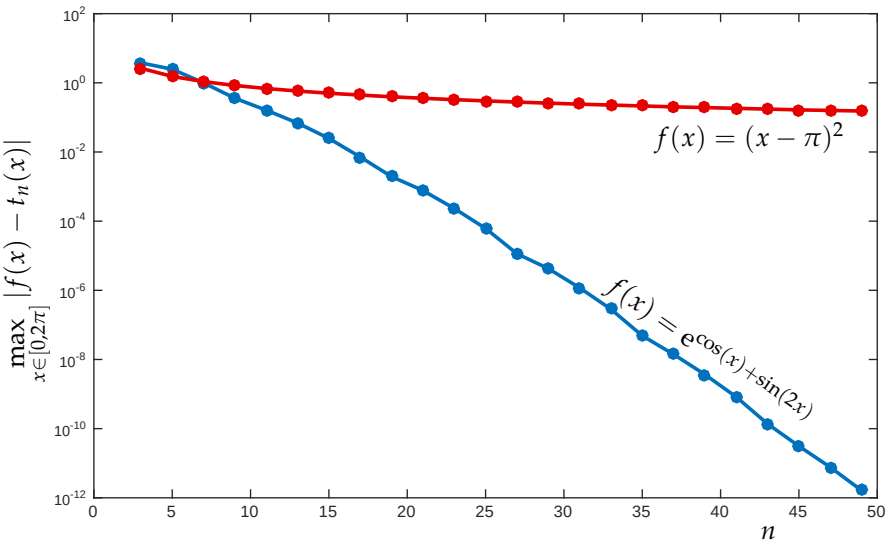


Figure 1.17: Convergence of the trigonometric polynomial interpolants to $f(x) = e^{\cos(x)+\sin(2x)}$ and $f(x) = (x-\pi)^2$. For the first function, convergence is extremely rapid as $n \to \infty$. The second function, restricted to $[0, 2\pi]$, can be viewed as a continuous but not continuously differentiable function. Though the approximation in Figure 1.16 looks good over $[0, 2\pi]$, the convergence of t_n to f is slow as $n \to \infty$.

Similarly, the inverse of this matrix can be computed from `fft(eye(n))` command:

$$\texttt{fft(eye(5))/5} = \frac{1}{5}\begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^{-1} & \omega^{-2} & \omega^{-3} & \omega^{-4} \\ \omega^0 & \omega^{-2} & \omega^{-4} & \omega^{-6} & \omega^{-8} \\ \omega^0 & \omega^{2} & \omega^{4} & \omega^{6} & \omega^{8} \\ \omega^0 & \omega^{1} & \omega^{2} & \omega^{3} & \omega^{4} \end{bmatrix}.$$

We could construct this matrix and multiply it against $\mathbf{f}$ to obtain γ, but that would require $\mathcal{O}(n^2)$ operations. Instead, we can compute γ directly using the `fft` command:

$$\gamma = \texttt{fft}(\mathbf{f})/\texttt{n}.$$

Recall that this reordered vector gives

$$\gamma = \begin{bmatrix} \gamma_0 \\ \gamma_1 \\ \gamma_2 \\ \gamma_{-2} \\ \gamma_{-1} \end{bmatrix},$$

which must be taken into account when constructing t_n.

Example 1.10 (MATLAB code for trigonometric interpolation)**.** We close with a sample of MATLAB code one could use to construct the interpolant t_n for the function $f(x) = e^{\cos(x)+\sin(2x)}$. First we present a generic code that will work for any (real- or complex-

valued) 2π-periodic f. Take special note of the simple one line command to find the coefficient vector γ.

```
f = @(x) exp(cos(x)+sin(2*x));                % define the function
n = 5;                                        % # terms in trig polynomial (must be odd)
xk = [0:n-1]'*2*pi/n;                         % interpolation points
xx = linspace(0,2*pi,500)';                   % fine grid on which to plot f, t_n
tn = zeros(size(xx));                         % initialize t_n

gamma = fft(f(xk))/n;                         % solve for coefficients, gamma

for k=1:(n+1)/2
   tn = tn + gamma(k)*exp(1i*(k-1)*xx);       % gamma_0, gamma_1, ... gamma_{(n-1)/2} terms
end
for k=(n+1)/2+1:n
   tn = tn + gamma(k)*exp(1i*(-n+k-1)*xx);    % gamma_{-(n-1)/2}, ..., gamma_{-1} terms
end
plot(xx,f(xx),'b-'), hold on                  % plot f
plot(xx, tn,'r-')                             % plot t_n
```

In the case that f is real-valued (as with all the examples shown in this section), one can further show that

$$\gamma_{-k} = \overline{\gamma_k},$$

indicating that the imaginary terms will not make any contribution to t_n. Since for $k = 1, \ldots, (n-1)/2$,

$$\gamma_{-k} e^{-1ikx} + \gamma_k e^{1ikx} = 2\Big(\mathrm{Re}(\gamma_k)\cos(kx) - \mathrm{Im}(\gamma_k)\sin(kx)\Big),$$

the code can be simplified slightly to construct t_n as follows.

```
gamma = fft(f(xk))/n;                         % solve for coefficients, gamma
tn = gamma(1)*exp(1i*0*xx);                   % initialize t_n(x) = gamma_0
for k=2:(n+1)/2
   tn = tn + 2*real(gamma(k))*cos((k-1)*xx) ...
           - 2*imag(gamma(k))*sin((k-1)*xx);  % exploit gamma_{-k} = conj(gamma_k)
end
```

LECTURE 8: *Piecewise interpolation*

1.10 Piecewise polynomial interpolation

WE HAVE SEEN, through Runge's example, that high degree polynomial interpolation can lead to large errors when the $(n+1)$st derivative of f is large in magnitude. In other cases, the interpolant converges to f, but the polynomial degree must be fairly high to deliver an approximation of acceptable accuracy throughout $[a, b]$. Beyond theoretical convergence questions, high-degree polynomials can be delicate to work with, even when using a stable implementation (the Lagrange basis, in its barycentric form). Many practical approximation problems are better solved by a simpler 'piecewise' alternative: instead of approximating f with one high-degree interpolating polynomial over a large interval $[a, b]$, patch together many low-degree polynomials that each interpolate f on some subinterval of $[a, b]$.

1.10.1 Piecewise linear interpolation

The simplest piecewise polynomial interpolation uses linear polynomials to interpolate between adjacent data points. Informally, the idea is to 'connect the dots.' Given $n+1$ data points $\{(x_j, f_j)\}_{j=0}^n$, we need to construct n linear polynomials $\{s_j\}_{j=1}^n$ such that

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

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

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

Note that all the s_j's are *linear* polynomials. Unlike our earlier notation, the subscript j *does not* reflect the polynomial degree.

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

To analyze the error, we can apply the interpolation bound developed in the last lecture. If we let Δ denote the largest space between interpolation points,

$$\Delta := \max_{j=1,\ldots,n} |x_j - x_{j-1}|,$$

then the standard interpolation error bound gives

$$\max_{x \in [x_0, x_n]} |f(x) - S(x)| \leq \max_{x \in [x_0, x_n]} \frac{|f''(x)|}{2} \Delta^2.$$

In particular, this proves convergence as $\Delta \to 0$ provided $f \in C^2[x_0, x_n]$.

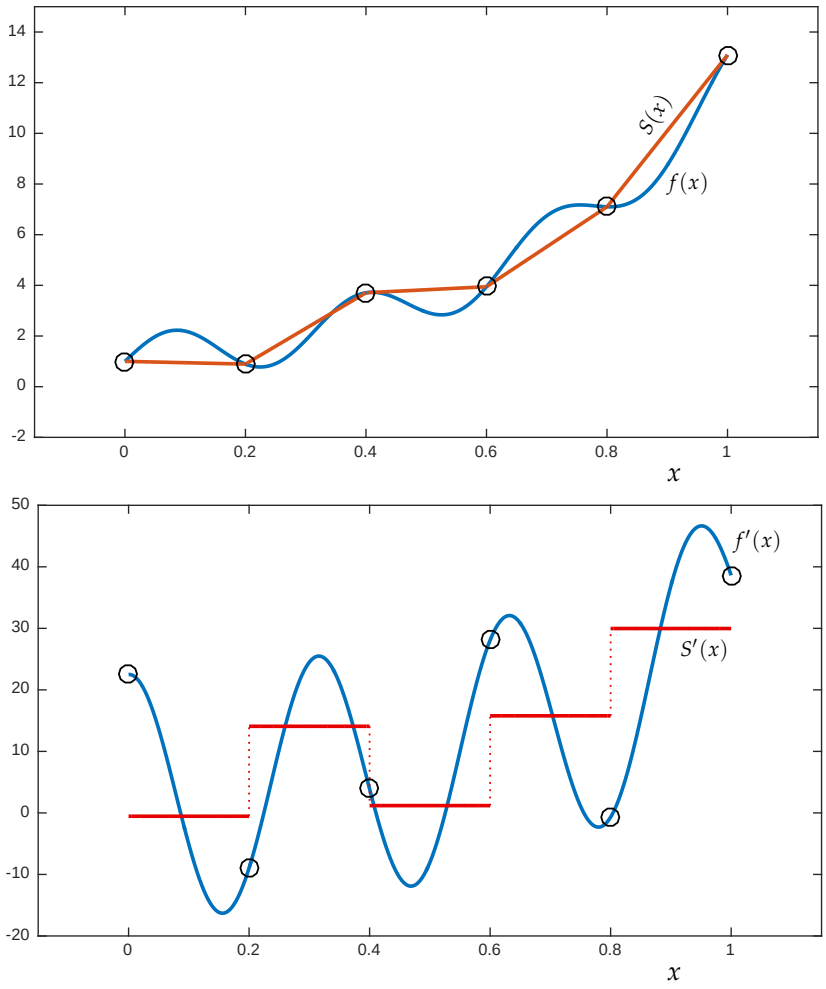


Figure 1.18: Piecewise linear interpolant to $f(x) = \sin(20x) + e^{5x/2}$ at $n = 5$ uniformly spaced points (top), and the derivative of this interpolant (bottom). Notice that the interpolant is continuous, but its derivative has jump discontinuities.

What could go wrong with this simple approach? The primary difficulty is that the interpolant is *continuous*, but generally not *continuously differentiable*. Still, these functions are easy to construct and cheap to evaluate, and can be very useful despite their simplicity.

1.10.2 *Piecewise cubic Hermite interpolation*

To remove the discontinuities in the first derivative of the piecewise linear interpolant, we begin by modeling our data with cubic polynomials over each interval $[x_j, x_{j+1}]$. Each such cubic has four free parameters (since $\mathcal{P}_3$ is a vector space of dimension 4); we require

these polynomials to interpolate both f and its first derivative:

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

To satisfy these conditions, take s_j to be the Hermite interpolant to the data $(x_{j-1}, f(x_{j-1}), f'(x_{j-1}))$ and $(x_j, f(x_j), f'(x_j))$. The resulting function, $S(x) := s_j(x)$ for $x \in [x_{j-1}, x_j]$, will always have a continuous derivative, $S \in C^1[x_0, x_n]$, but generally $S \notin C^2[x_0, x_n]$ due to discontinuities in the second derivative at the interpolation points.

In many applications, we lack specific values for $S'(x_j) = f'(x_j)$; we simply want the function $S(x)$ to be as *smooth* as possible. That motivates our next topic: splines.

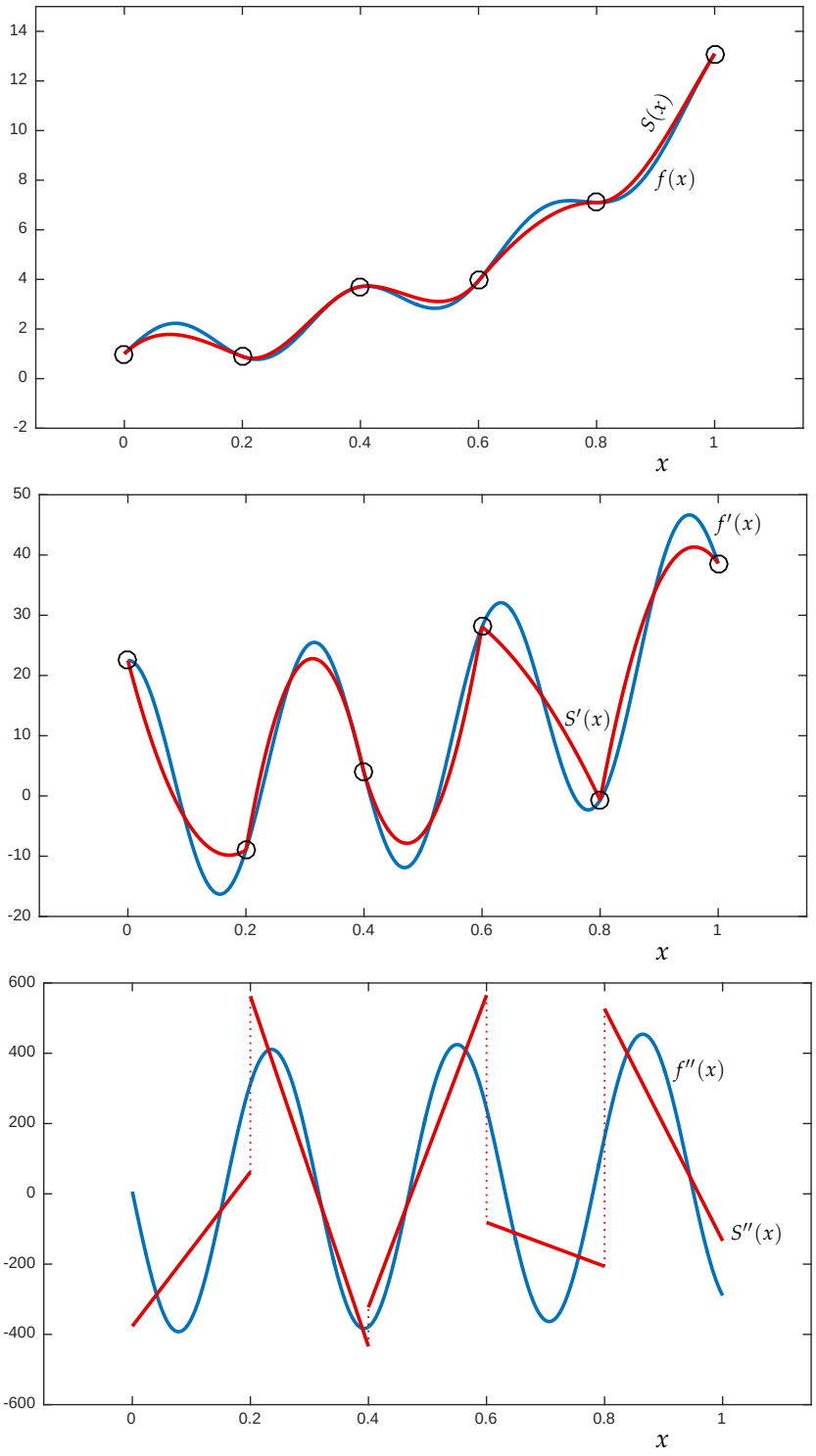


Figure 1.19: Piecewise cubic Hermite interpolant to $f(x) = \sin(20x) + e^{5x/2}$ at $n = 5$ uniformly spaced points (top), and the derivative of this interpolant (middle). Now both the interpolant and its derivative are continuous, and the derivative interpolates f'. However, the second derivative of the interpolant now has jump discontinuities (bottom).

LECTURE 9: *Introduction to Splines*

1.11 Splines

Spline fitting, our next topic in interpolation theory, is an essential tool for engineering design. As in the last lecture, we strive to interpolate data using low-degree polynomials between consecutive grid points. The piecewise linear functions of Section 1.10 were simple, but suffered from unsightly kinks at each interpolation point, reflecting a discontinuity in the first derivative. By increasing the degree of the polynomial used to model f on each subinterval, we can obtain smoother functions.

Long before numerical analysts got their hands on them, 'splines' were used in the woodworking, shipbuilding, and aircraft industries. In such work 'splines' refer to thin pieces of wood that are bent between physical constraints called *ducks* (apparently these were also called *dogs* and *rats* in some settings; modern versions are sometimes called *whales* because of their shape). The spline, a thin beam, bends gracefully between the ducks to give a graceful curve. For some discussion of this history, see the brief 'History of Splines' note by James Epperson in the 19 July 1998 NA Digest, linked from the class website. For a beautiful derivation of cubic splines from Euler's beam equation—that is, from the original physical situation, see Gilbert Strang's *Introduction to Applied Mathematics*, Wellesley Cambridge Press, 1986.

1.11.1 Cubic splines: first approach

Rather than setting $S'(x_j)$ to a particular value, suppose we simply require S' to be continuous throughout $[x_0, x_n]$. This added freedom allows us to impose a further condition: require S'' to be continuous on $[x_0, x_n]$, too. The polynomials we construct are called *cubic splines*. In spline parlance, the interpolation points $\{x_j\}_{j=0}^n$ are called *knots*.

These cubic spine requirements can be written as:

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

Compare these requirements to those imposed by piecewise cubic Hermite interpolation. Add up all these new requirements:

$$n + n + (n-1) + (n-1) = 4n - 2 \text{ constraints}$$

and compare to the total free variables available:

$$(n \text{ cubic polynomials}) \times (4 \text{ variables per cubic}) = 4n \text{ variables}.$$

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

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

- *natural* splines require $S''(x_0) = S''(x_n) = 0$;
- *complete* splines specify values for $S'(x_0)$ and $S'(x_n)$;
- *not-a-knot* splines require S''' to be continuous at x_1 and x_{n-1}.

Since the third derivative of a cubic is a constant, the *not-a-knot* requirement forces $s_1 = s_2$ and $s_{n-1} = s_n$. Hence, while $S(x)$ interpolates the data at x_2 and x_{n-1}, the derivative continuity requirements are automatic at those knots; hence the name "not-a-knot".

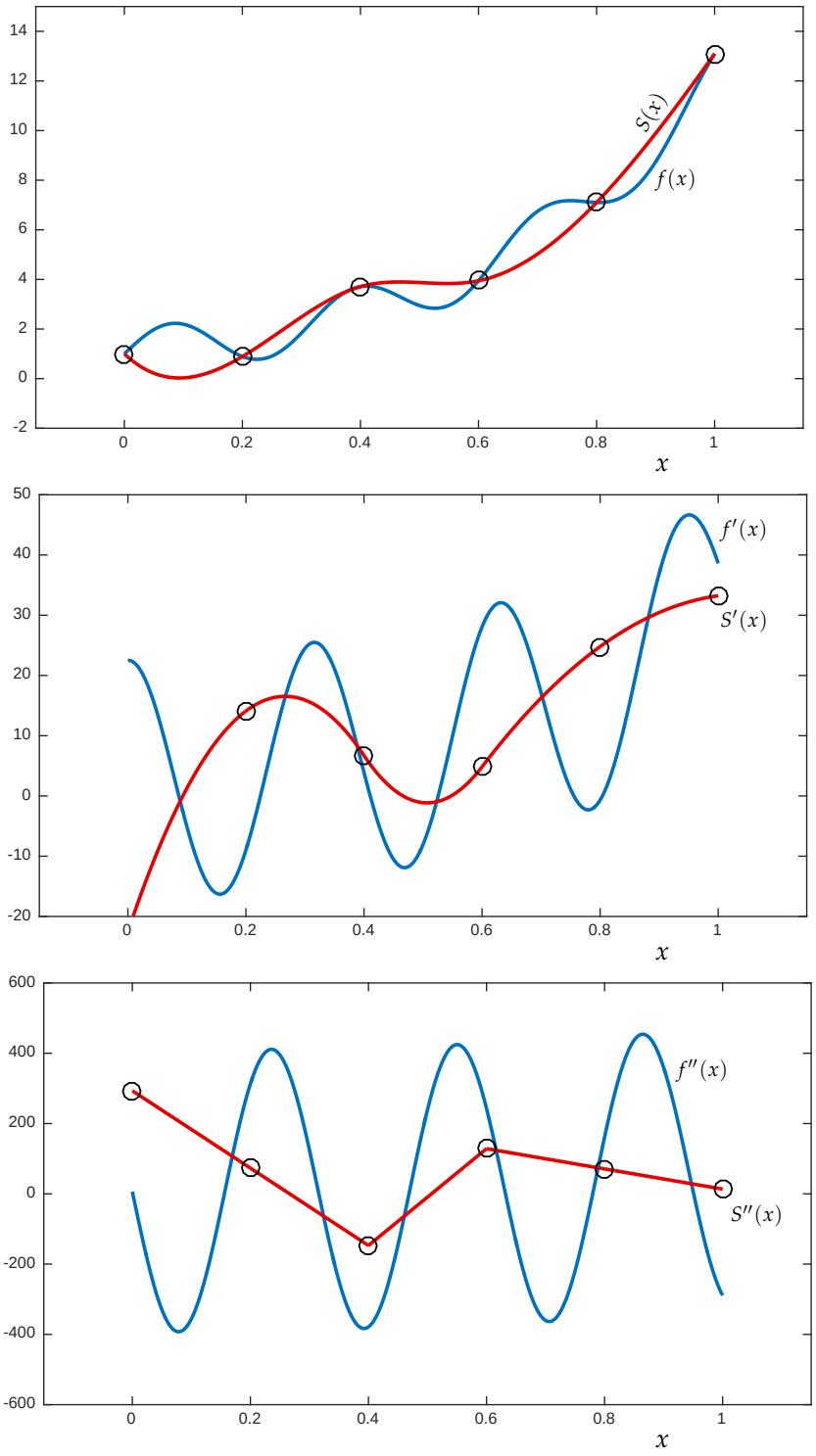


Figure 1.20: Not-a-knot cubic spline interpolant to $f(x) = \sin(20x) + e^{5x/2}$ at $n = 5$ uniformly spaced knots (top), along with its first (middle) and second (bottom) derivative. Note that S, S', and S'' are all continuous. Look closely at the plot of S'': clearly this function will have jump discontinuities at the interior nodes x_2 and x_3, but the *not-a-knot* condition forces S''' to be continuous at the knots x_1 and $x_4 = x_{n-1}$.

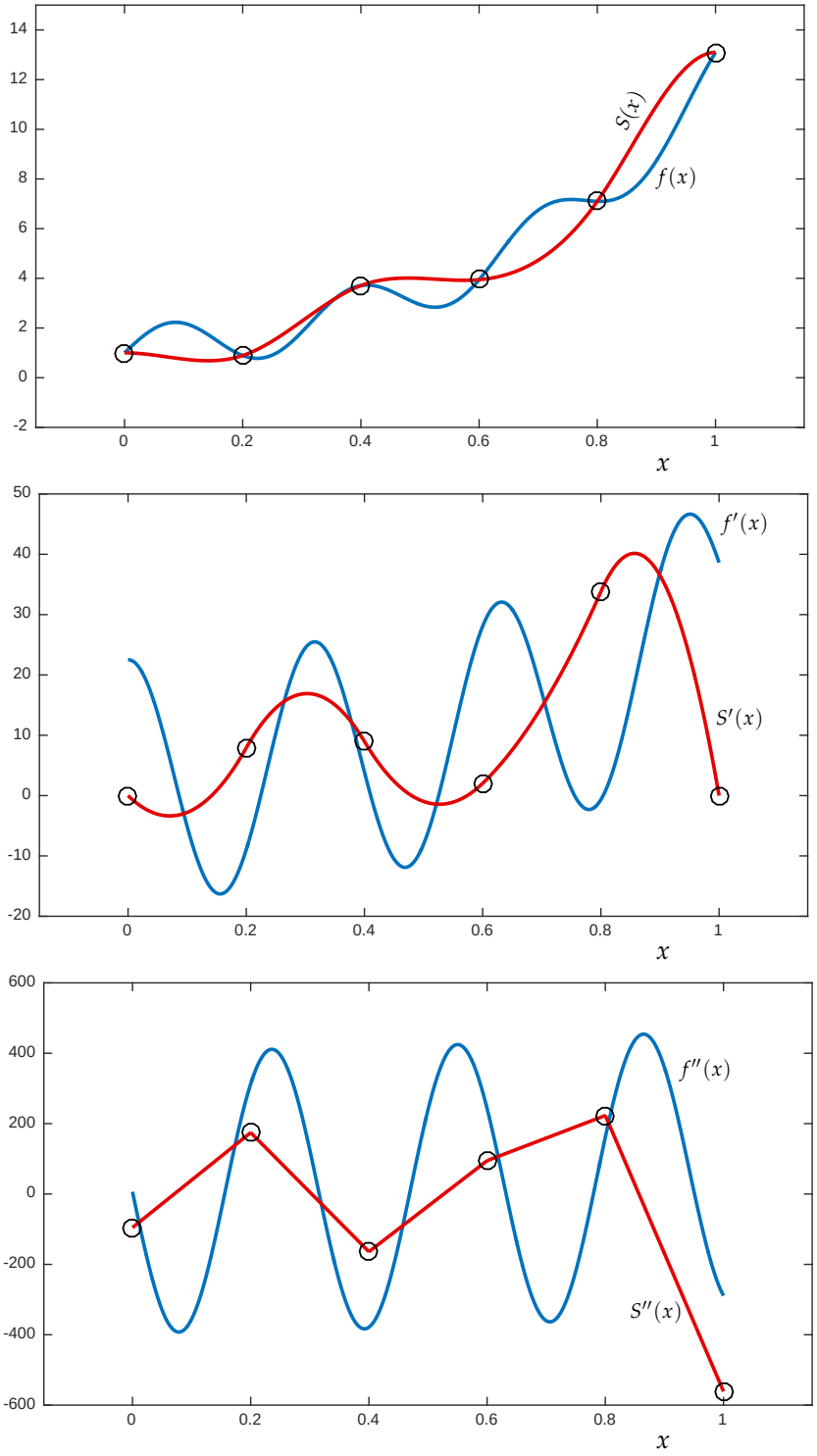


Figure 1.21: Complete cubic spline interpolant to $f(x) = \sin(20x) + e^{5x/2}$ at $n = 5$ uniformly spaced knots (top), along with its first (middle) and second (bottom) derivative. Note that S, S', and S'' are all continuous. For a *complete* cubic spline, one specifies the value of $S'(x_0)$ and $S'(x_n)$; in this case we have set $S'(x_0) = S'(x_n) = 0$, as you can confirm in the middle plot. In the bottom plot, see that $S'''(x)$ will have jump discontinuities at all the interior knots $x_1, \ldots, x_{n-1}$, in contrast to the not-a-knot spline shown in Figure 1.20.

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

Continuing with the example from the last section, Figure 1.20 shows a not-a-knot spline, where S''' is continuous at x_1 and x_{n-1}. The cubic polynomials s_1 for $[x_0, x_1]$ and s_2 for $[x_1, x_2]$ must satisfy

$$\begin{aligned} s_1(x_1) &= s_2(x_1) \\ s_1'(x_1) &= s_2'(x_1) \\ s_1''(x_1) &= s_2''(x_1) \\ s_1'''(x_1) &= s_2'''(x_1) \end{aligned}$$

Two cubics that match these four conditions must be the same: $s_1(x) = s_2(x)$, and hence x_1 is 'not a knot.' (The same goes for x_{n-1}.) Notice this behavior in Figure 1.20. In contrast, Figure 1.21 shows the complete cubic spline, where $S'(x_0) = S'(x_n) = 0$.

However we assign the two additional conditions, we get a system of $4n$ equations (the various constraints) in $4n$ unknowns (the cubic polynomial coefficients). These equations can be set up as a system involving a banded coefficient matrix (zero everywhere except for a limited number of diagonals on either side of the main diagonal). We could derive this linear system by directly enforcing the continuity conditions on the cubic polynomial that we have just described. Instead, we will develop a more general approach that expresses the spline function $S(x)$ as the linear combination of special basis functions, which themselves are splines.

One can arrange Gaussian elimination to solve an $n \times n$ tridiagonal system in $\mathcal{O}(n)$ operations.

Try constructing this matrix!

LECTURE 10: *B-Splines*

1.11.2 B-Splines: a basis for splines

Throughout our discussion of standard polynomial interpolation, we viewed $\mathcal{P}_n$ as a linear space of dimension $n+1$, and then expressed the unique interpolating polynomial in several different bases (monomial, Newton, Lagrange). The most elegant way to develop spline functions uses the same approach. A set of *basis splines*, depending only on the location of the knots and the degree of the approximating piecewise polynomials can be developed in a convenient, numerically stable manner. (Cubic splines are the most prominent special case.)

For example, each cubic basis spline, or *B-spline*, is a continuous piecewise-cubic function with continuous first and second derivatives. Thus any linear combination of such B-splines will inherit the same continuity properties. The coefficients in the linear combination are chosen to obey the specified interpolation conditions.

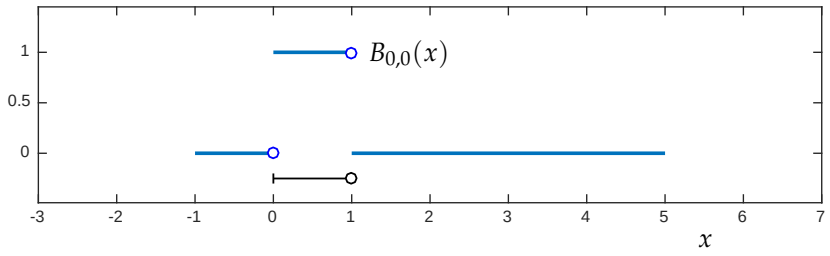
B-splines are built up recursively from constant B-splines. Though we are interpolating data at $n+1$ knots $x_0, \ldots, x_n$, to derive B-splines we need extra nodes outside $[x_0, x_n]$ as scaffolding upon which to construct the basis. Thus, add knots on either end of x_0 and x_n:

$$\cdots < x_{-2} < x_{-1} < x_0 < x_1 < \cdots < x_n < x_{n+1} < \cdots .$$

Given these knots, define the constant (zeroth-degree) B-splines:

$$B_{j,0}(x) = \begin{cases} 1 & x \in [x_j, x_{j+1}); \\ 0 & \text{otherwise.} \end{cases}$$

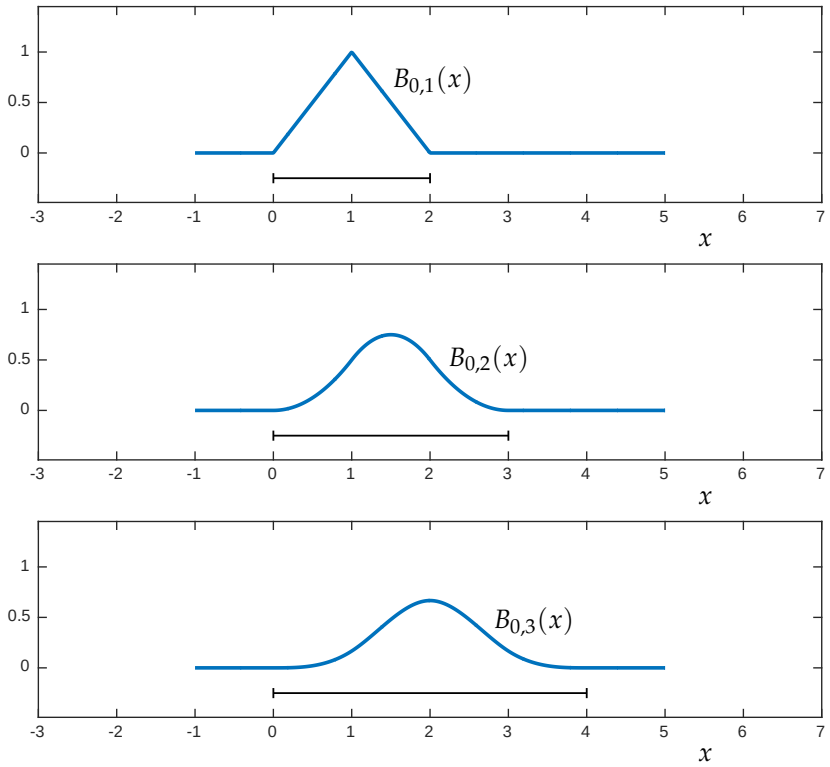
The following plot shows the basis function $B_{0,0}$ for the knots $x_j = j$. Note, in particular, that $B_{j,0}(x_{j+1}) = 0$. The line drawn beneath the spline marks the *support* of the spline, that is, the values of x for which $B_{0,0}(x) \neq 0$.



From these degree-0 B-splines, manufacture B-splines of higher degree via the recurrence

$$B_{j,k}(x) = \left(\frac{x - x_j}{x_{j+k} - x_j}\right) B_{j,k-1}(x) + \left(\frac{x_{j+k+1} - x}{x_{j+k+1} - x_{j+1}}\right) B_{j+1,k-1}(x). \tag{1.33}$$

While not immediately obvious from the formula, this construction ensures that $B_{j,k}$ has one more continuous derivative than does $B_{j,k-1}$. Thus, while $B_{j,0}$ is discontinuous (see previous plot), $B_{j,1}$ is continuous, $B_{j,2} \in C^1(\mathbb{R})$, and $B_{j,3} \in C^2(\mathbb{R})$. One can see this in the three plots below, where again $x_j = j$. As the degree increases, the B-spline $B_{j,k}$ becomes increasingly smooth. Smooth is good, but it has a consequence: the *support* of $B_{j,k}$ gets larger as we increase k. This, as we will see, has implications on the number of nonzero entries in the linear system we must ultimately solve to find the expansion of the desired spline in the B-spline basis.



From these plots and the recurrence defining $B_{j,k}$, one can deduce several important properties:

- $B_{j,k} \in C^{k-1}(\mathbb{R})$ (continuity);
- $B_{j,k}(x) = 0$ if $x \notin (x_j, x_{j+k+1})$ (compact support);
- $B_{j,k}(x) > 0$ for $x \in (x_j, x_{j+k+1})$ (positivity).

Finally, we are prepared to write down a formula for the spline that interpolates $\{(x_j, f_j)\}_{j=0}^n$ as a linear combination of the basis splines we have just constructed. Let $S_k(x)$ denote the spline consisting of piecewise polynomials in $\mathcal{P}_k$. In particular, S_k must obey the following properties:

- $S_k(x_j) = f_j$ for $j = 0, \ldots, n$;
- $S_k \in C^{k-1}[x_0, x_n]$ for $k \geq 1$.

The beauty of B-splines is that the second of these properties is automatically inherited from the B-splines themselves. (Any linear combination of $C^{k-1}(\mathbb{R})$ functions must itself be a $C^{k-1}(\mathbb{R})$ function.) The interpolation conditions give $n+1$ equations that constrain the unknown coefficients $c_{j,k}$ in the expansion of S_k:

$$S_k(x) = \sum_j c_{j,k} B_{j,k}(x). \tag{1.34}$$

What limits should j have in this sum? For the greatest flexibility, let j range over all values for which

$$B_{j,k}(x) \neq 0 \quad \text{for some } x \in [x_0, x_n].$$

If $B_{j,k}(x) = 0$ for all $x \in [x_0, x_n]$, it cannot contribute to the interpolation requirement $S_k(x_j) = f_j$, $j = 0, \ldots, n$.

Figure 1.22 shows the B-splines of degree $k = 1, 2, 3$ that overlap the interval $[x_0, x_4]$ for $x_j = j$. For $k \geq 1$, $B_{j,k}(x)$ is supported on (x_j, x_{j+k+1}), and hence the limits on the sum in (1.34) take the form

$$S_k(x) = \sum_{j=-k}^{n-1} c_{j,k} B_{j,k}(x), \qquad k \geq 1. \tag{1.35}$$

The sum involves $n + k$ coefficients $c_{j,k}$, which must be determined to

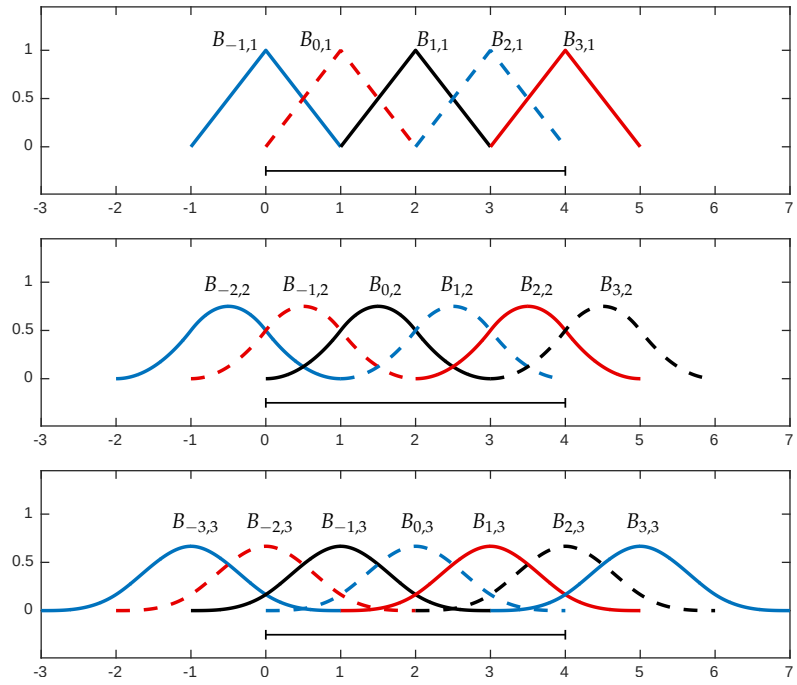


Figure 1.22: B-splines of degree $k = 1$ (top), $k = 2$ (middle), and $k = 3$ (bottom) that are supported on the interval $[x_0, x_n]$ for $x_j = j$ with $n = 4$. Note that $n + k$ B-splines are supported on $[x_0, x_n]$.

satisfy the $n+1$ interpolation conditions

$$f_\ell = S_k(x_\ell) = \sum_{j=-k}^{n-1} c_{j,k} B_{j,k}(x_\ell), \qquad \ell = 0, \ldots, n.$$

Before addressing general $k \geq 1$, we pause to handle the special case of $k = 0$, i.e., constant splines.

1.11.3 Constant splines, $k = 0$

The constant B-splines give $B_{n,0}(x_n) = 1$ and so, unlike the general $k \geq 1$ case, the $j = n$ B-spline must be included in the spline sum:

$$S_0(x) = \sum_{j=0}^{n} c_{j,0} B_{j,0}(x).$$

The interpolation conditions give, for $\ell = 0, \ldots, n$,

$$\begin{aligned} f_\ell = S_0(x_\ell) &= \sum_{j=0}^{n} c_{j,0} B_{j,0}(x_\ell) \\ &= c_{\ell,0} B_{\ell,0}(x_\ell) = c_{\ell,0}, \end{aligned}$$

since $B_{j,0}(x_\ell) = 0$ if $j \neq \ell$, and $B_{\ell,0}(x_\ell) = 1$ (recall the plot of $B_{0,0}(x)$ shown earlier). Thus $c_{\ell,0} = f_\ell$, and the degree $k = 0$ spline interpolant is simply

$$S_0(x) = \sum_{j=0}^{n} f_j B_{j,0}(x).$$

LECTURE 11: *Matrix Determination of Splines; Energy Minimization*

1.11.4 *General case, $k \geq 1$*

Now consider the general spline interpolant of degree $k \geq 1$,

$$S_k(x) = \sum_{j=-k}^{n-1} c_{j,k} B_{j,k}(x),$$

with constants $c_{-k,k}, \ldots, c_{n-1,k}$ determined to satisfy the interpolation conditions $S_k(\ell) = f_\ell$, i.e.,

$$\sum_{j=-k}^{n-1} c_{j,k} B_{j,k}(x_\ell) = f_\ell, \quad \ell = 0, \ldots, n.$$

By now we are accustomed to transforming constraints like this into matrix equations. Each value $\ell = 0, \ldots, n$ gives a row of the equation

$$\begin{bmatrix} B_{-k,k}(x_0) & B_{-k+1,k}(x_0) & \cdots & B_{n-1,k}(x_0) \\ B_{-k,k}(x_1) & B_{-k+1,k}(x_1) & \cdots & B_{n-1,k}(x_1) \\ \vdots & \vdots & \ddots & \vdots \\ B_{-k,k}(x_n) & B_{-k+1,k}(x_n) & \cdots & B_{n-1,k}(x_n) \end{bmatrix} \begin{bmatrix} c_{-k,k} \\ c_{-k+1,k} \\ \vdots \\ c_{n-1,k} \end{bmatrix} = \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_n \end{bmatrix}. \tag{1.36}$$

Let us consider the matrix in this equation. The matrix will have $n+1$ rows and $n+k$ columns, so when $k > 1$ the system of equations will be *underdetermined*. Since B-splines have 'small support' (i.e., $B_{j,k}(x) = 0$ for most $x \in [x_0, x_n]$), this matrix will be *sparse*: most entries will be zero.

One could obtain an $(n+1) \times (n+1)$ matrix by arbitrarily setting $k-1$ certain values of $c_{j,k}$ to zero, but this would miss a great opportunity: we can constructively include all $n+k$ B-splines and impose k extra properties on S_k to pick out a unique spline interpolant from the infinitely many options that satisfy the interpolation conditions.

The following subsections will describe the particular form the system (1.36) takes for $k = 1, 2, 3$. In each case we will illustrate the resulting spline interpolant through the following data set.

j	0	1	2	3	4
x_j	0	1	2	3	4
f_j	1	3	2	-1	1

(1.37)

1.11.5 *Linear splines, $k = 1$*

Linear splines are simple to construct: in this case $n + k = n + 1$, so the matrix in (1.36) is square. Let us evaluate it: since

$$B_{j,1}(x_\ell) = \begin{cases} 1, & \ell = j+1; \\ 0, & \ell \neq j+1, \end{cases}$$

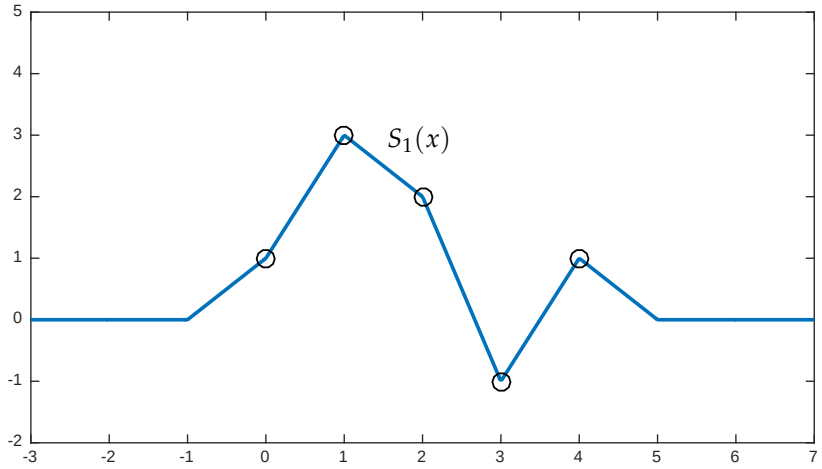


Figure 1.23: Linear spline S_1 interpolating 5 data points $\{(x_j, f_j)\}_{j=0}^4$.

the matrix is simply

$$\begin{bmatrix} B_{-1,1}(x_0) & B_{0,1}(x_0) & \cdots & B_{n-1,1}(x_0) \\ B_{-1,1}(x_1) & B_{0,1}(x_1) & \cdots & B_{n-1,1}(x_1) \\ \vdots & \vdots & \ddots & \vdots \\ B_{-1,1}(x_n) & B_{0,1}(x_n) & \cdots & B_{n-1,1}(x_n) \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & 1 \end{bmatrix} = \mathbf{I}.$$

The system (1.36) is thus trivial to solve, reducing to

$$\begin{bmatrix} c_{-1,1} \\ c_{-0,k} \\ \vdots \\ c_{n-1,k} \end{bmatrix} = \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_n \end{bmatrix},$$

which gives the unique linear spline

$$S_1(x) = \sum_{j=-1}^{n-1} f_{j+1} B_{j,1}(x).$$

This above discussion is a pedantic way to arrive at an obvious solution: Since the jth 'hat function' B-spline equals one at x_{j+1} and zero at all other knots, just write the unique formula for the interpolant immediately.

Figure 1.23 shows the unique piecewise linear spline interpolant to the data in (1.37), which is a linear combination of the five linear splines shown in Figure 1.22. Explicitly,

$$\begin{aligned} S_1(x) &= f_0 B_{-1,1}(x) + f_1 B_{0,1}(x) + f_2 B_{1,1}(x) + f_3 B_{2,1}(x) + f_4 B_{3,1}(x) \\ &= B_{-1,1}(x) + 3\,B_{0,1}(x) + 2\,B_{1,1}(x) - B_{2,1}(x) + B_{3,1}(x). \end{aligned}$$

Note that linear splines are simply C^0 functions that interpolate a given data set—between the knots, they are identical to the piecewise linear functions constructed in Section 1.10.1. Note that $S_1(x)$ is supported on (x_{-1}, x_{n+1}) with $S_1(x) = 0$ for all $x \notin (x_{-1}, x_{n+1})$. This is a general feature of splines: Outside the range of interpolation, $S_k(x)$ goes to zero as quickly as possible for a given set of knots while still maintaining the specified continuity.

1.11.6 Quadratic splines, $k = 2$

The construction of quadratic B-splines from the linear splines via the recurrence (1.33) forces the functions $B_{j,2}$ to have a continuous derivative, and also to be supported over three intervals per spline, as seen in the middle plot in Figure 1.22. The interpolant takes the form

$$S_2(x) = \sum_{j=-2}^{n-1} c_{j,2} B_{j,2}(x),$$

with coefficients specified by $n+1$ equations in $n+2$ unknowns:

$$\begin{bmatrix} B_{-2,2}(x_0) & B_{-1,2}(x_0) & \cdots & B_{n-1,2}(x_0) \\ B_{-2,2}(x_1) & B_{-1,2}(x_1) & \cdots & B_{n-1,2}(x_1) \\ \vdots & \vdots & \ddots & \vdots \\ B_{-2,2}(x_n) & B_{-1,2}(x_n) & \cdots & B_{n-1,2}(x_n) \end{bmatrix} \begin{bmatrix} c_{-2,2} \\ c_{-1,2} \\ \vdots \\ c_{n-1,2} \end{bmatrix} = \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_n \end{bmatrix}. \tag{1.38}$$

Since there are more variables than constraints, we expect infinitely many quadratic splines that interpolate the data.

Evaluate the entries of the matrix in (1.38). First note that

$$B_{j,2}(x_\ell) = 0, \quad \ell \notin \{j+1, j+2\},$$

so the matrix is zero in all entries except the main diagonal ($B_{j,2}(x_{j+2})$) and the first superdiagonal ($B_{j,2}(x_{j+1})$). To evaluate these nonzero entries, recall that the recursion (1.33) for B-splines gives

$$B_{j,2}(x) = \Big(\frac{x - x_j}{x_{j+2} - x_j}\Big) B_{j,1}(x) + \Big(\frac{x_{j+3} - x}{x_{j+3} - x_{j+1}}\Big) B_{j+1,1}(x).$$

Evaluate this function at x_{j+1} and x_{j+2}, using our knowledge of the

$B_{j,1}$ linear B-splines ('hat functions'):

$$B_{j,2}(x_{j+1}) = \Big(\frac{x_{j+1}-x_j}{x_{j+2}-x_j}\Big)B_{j,1}(x_{j+1}) + \Big(\frac{x_{j+3}-x_{j+1}}{x_{j+3}-x_{j+1}}\Big)B_{j+1,1}(x_{j+1})$$

$$= \Big(\frac{x_{j+1}-x_j}{x_{j+2}-x_j}\Big)\cdot 1 + \Big(\frac{x_{j+3}-x_{j+1}}{x_{j+3}-x_{j+1}}\Big)\cdot 0 = \frac{x_{j+1}-x_j}{x_{j+2}-x_j};$$

$$B_{j,2}(x_{j+2}) = \Big(\frac{x_{j+2}-x_j}{x_{j+2}-x_j}\Big)B_{j,1}(x_{j+2}) + \Big(\frac{x_{j+3}-x_{j+2}}{x_{j+3}-x_{j+1}}\Big)B_{j+1,1}(x_{j+2})$$

$$= \Big(\frac{x_{j+2}-x_j}{x_{j+2}-x_j}\Big)\cdot 0 + \Big(\frac{x_{j+3}-x_{j+2}}{x_{j+3}-x_{j+1}}\Big)\cdot 1 = \frac{x_{j+3}-x_{j+2}}{x_{j+3}-x_{j+1}}.$$

Use these formulas to populate the superdiagonal and subdiagonal of the matrix in (1.38). In the (important) special case of uniformly spaced knots

$$x_j = x_0 + jh, \quad \text{for fixed } h > 0,$$

gives the particularly simple formulas

$$B_{j,2}(x_{j+1}) = B_{j,2}(x_{j+2}) = \frac{1}{2},$$

hence the system (1.38) becomes

$$\begin{bmatrix} 1/2 & 1/2 & & & \\ & 1/2 & 1/2 & & \\ & & \ddots & \ddots & \\ & & & 1/2 & 1/2 \end{bmatrix} \begin{bmatrix} c_{-2,2} \\ c_{-1,2} \\ c_{0,2} \\ \vdots \\ c_{n-1,2} \end{bmatrix} = \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_n \end{bmatrix},$$

where the blank entries are zero. This $(n+1)\times(n+2)$ system will have infinitely many solutions, i.e., infinitely many splines that satisfy the interpolation conditions. How to choose among them? Impose *one* extra condition, such as $S_2'(x_0) = 0$ or $S_2'(x_n) = 0$.

As an example, let us work through the condition $S_2'(x_0) = 0$; it raises an interesting issue. Refer to the middle plot in Figure 1.22. Due to the small support of the quadratic B-splines, $B_{j,2}'(x_0) = 0$ for $j > 0$, so

$$S_2'(x_0) = c_{-2,2}B_{-2,2}'(x_0) + c_{-1,2}B_{-1,2}'(x_0) + c_{0,2}B_{0,2}'(x_0). \tag{1.39}$$

The derivatives of the B-splines at knots are tricky to compute. Differentiating the recurrence (1.33) with $k = 2$, we can formally write

$$B_{j,2}'(x) = \Big(\frac{1}{x_{j+2}-x_j}\Big)B_{j,1}(x) + \Big(\frac{x-x_j}{x_{j+2}-x_j}\Big)B_{j,1}'(x) - \Big(\frac{1}{x_{j+3}-x_{j+1}}\Big)B_{j+1,1}(x) + \Big(\frac{x_{j+3}-x}{x_{j+3}-x_{j+1}}\Big)B_{j+1,1}'(x).$$

Try to evaluate this expression at x_j, x_{j+1}, or x_{j+2}: you must face that fact that the linear B-splines $B_{j,1}$ and $B_{j+1,1}$ *are not differentiable at the knots*! You must instead check that the one-sided derivatives match, e.g.,

$$\lim_{\substack{h\to 0\\ h<0}} \frac{B_{j,2}(x_{j+1}+h) - B_{j,2}(x_{j+1})}{h} = \lim_{\substack{h\to 0\\ h>0}} \frac{B_{j,2}(x_{j+1}+h) - B_{j,2}(x_{j+1})}{h}.$$

A mildly tedious calculation verifies that indeed these one-sided first derivatives do match, and that is the point of splines: each time you increase the degree k, you increase the smoothness, so $B_{j,2} \in C^1(\mathbb{R})$.

Now regarding formula (1.39), one can compute

$$B'_{-2,2}(x_0) = -\frac{2}{x_1 - x_{-1}}, \quad B'_{-1,2}(x_0) = \frac{2}{x_1 - x_{-1}}, \quad B'_{0,2}(x_0) = 0,$$

and so, in the special case of a uniformly spaced grid ($x_j = x_0 + jh$), the condition $S'(x_0) = 0$ becomes

$$-\frac{1}{h}c_{-2,2} + \frac{1}{h}c_{-1,2} = 0.$$

Insert this equation as the first row in the linear system for the coefficients,

$$\begin{bmatrix} -1/h & 1/h & & & & \\ 1/2 & 1/2 & & & & \\ & 1/2 & 1/2 & & & \\ & & \ddots & \ddots & & \\ & & & 1/2 & 1/2 \end{bmatrix} \begin{bmatrix} c_{-2,2} \\ c_{-1,2} \\ c_{0,2} \\ \vdots \\ c_{n-1,2} \end{bmatrix} = \begin{bmatrix} 0 \\ f_0 \\ f_1 \\ \vdots \\ f_n \end{bmatrix},$$

and solve this for $c_{-2,2}, \ldots, c_{n-1,2}$ to determine the unique interpolating quadratic spline with $S_2'(x_0) = 0$.

Figure 1.24 shows quadratic spline interpolants to the data in (1.37). One spline is determined with the extra condition $S_2'(x_0) = 0$ described above; the other satisfies $S_2'(x_n) = 0$. In any case, the quadratic spline S_2 is supported on (x_{-2}, x_{n+2}).

1.11.7 Cubic splines, $k = 3$

Cubic splines are the most famous of all splines. We began this section by discussing cubic splines as an alternative to piecewise cubic Hermite interpolation. Now we will show how to derive the same cubic splines from the cubic B-splines.

Begin by reviewing the bottom plot in Figure 1.22. The cubic B-splines $B_{-3,3}, \ldots, B_{n-1,3}$ take nonzero values on the interval $[x_0, x_n]$, and hence we write the cubic spline as

$$S_3(x) = \sum_{j=-3}^{n-1} c_{j,3} B_{j,3}(x). \tag{1.40}$$

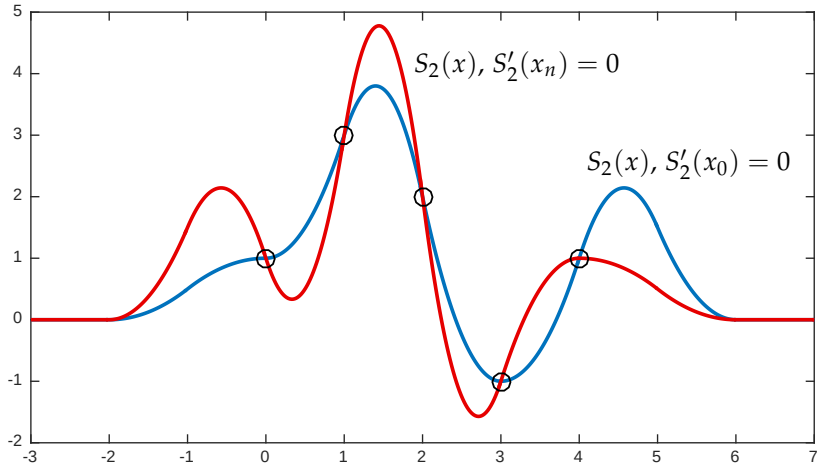


Figure 1.24: Two choices for the quadratic spline S_2 that interpolates the 5 data points $\{(x_j, f_j)\}_{j=0}^4$ in (1.37). The blue spline satisfies the extra condition that $S_2'(x_0) = 0$, while the red spline satisfies $S_2'(x_n) = 0$. Check to see that these conditions are consistent with the splines in the plot.

The linear system (1.36) now involves $n+1$ equations in $n+3$ unknowns:

$$\begin{bmatrix} B_{-3,3}(x_0) & B_{-2,3}(x_0) & \cdots & B_{n-1,3}(x_0) \\ B_{-3,3}(x_1) & B_{-2,3}(x_1) & \cdots & B_{n-1,3}(x_1) \\ \vdots & \vdots & \ddots & \vdots \\ B_{-3,3}(x_n) & B_{-2,3}(x_n) & \cdots & B_{n-1,3}(x_n) \end{bmatrix} \begin{bmatrix} c_{-3,3} \\ c_{-2,3} \\ \vdots \\ c_{n-1,3} \end{bmatrix} = \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_n \end{bmatrix}. \tag{1.41}$$

Given the support of cubic splines, note that

$$B_{j,3}(x_\ell) = 0, \quad \ell \notin \{j+1, j+2, j+3\},$$

which implies that only three diagonals of the matrix in (1.41) will be nonzero. We shall only work out the nonzero entries in the case of uniformly spaced knots, $x_j = x_0 + jh$ for fixed $h > 0$. In this case,

$$\begin{aligned} B_{j,3}(x_{j+1}) &= \left(\frac{x_{j+1} - x_j}{x_{j+3} - x_j}\right) B_{j,2}(x_{j+1}) + \left(\frac{x_{j+4} - x_{j+1}}{x_{j+4} - x_{j+1}}\right) B_{j+1,2}(x_{j+1}) = \left(\frac{h}{3h}\right) \cdot \frac{1}{2} + \left(\frac{3h}{3h}\right) \cdot 0 = \frac{1}{6} \\ B_{j,3}(x_{j+2}) &= \left(\frac{x_{j+2} - x_j}{x_{j+3} - x_j}\right) B_{j,2}(x_{j+2}) + \left(\frac{x_{j+4} - x_{j+2}}{x_{j+4} - x_{j+1}}\right) B_{j+1,2}(x_{j+2}) = \left(\frac{2h}{3h}\right) \cdot \frac{1}{2} + \left(\frac{2h}{3h}\right) \cdot \frac{1}{2} = \frac{2}{3} \\ B_{j,3}(x_{j+3}) &= \left(\frac{x_{j+3} - x_j}{x_{j+3} - x_j}\right) B_{j,2}(x_{j+3}) + \left(\frac{x_{j+4} - x_{j+3}}{x_{j+4} - x_{j+1}}\right) B_{j+1,2}(x_{j+3}) = \left(\frac{3h}{3h}\right) \cdot 0 + \left(\frac{h}{3h}\right) \cdot \frac{1}{2} = \frac{1}{6}, \end{aligned}$$

where we have used the fact that $B_{j,2}(x_{j+1}) = B_{j,2}(x_{j+2}) = 1/2$ and

$B_{j,2}(x_j) = B_{j,2}(x_{j+3}) = 0$. Substituting these values into (1.41) gives

$$
(1.42)\qquad
\begin{bmatrix}
1/6 & 2/3 & 1/6 & & & \\
 & 1/6 & 2/3 & 1/6 & & \\
 & & \ddots & \ddots & \ddots & \\
 & & & 1/6 & 2/3 & 1/6
\end{bmatrix}
\begin{bmatrix} c_{-3,3} \\ c_{-2,3} \\ \vdots \\ c_{n-1,3} \end{bmatrix}
=
\begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_n \end{bmatrix}
$$

involving a matrix with $n+1$ rows and $n+3$ columns. Again, infinitely many cubic splines satisfy these interpolation conditions; two independent requirements must be imposed to determine a unique spline. Recall the three alternatives discussed in Section 1.11.1: complete splines (specify a value for S_3' at x_0 and x_n), natural splines (force $S_3''(x_0) = S_3''(x_n) = 0$), or not-a-knot splines. One can show that imposing natural spline conditions on S_3 requires

$$(x_2 - x_{-1})c_{-3,3} - (x_2 + x_1 - x_{-1} - x_{-2})c_{-2,3} + (x_1 - x_{-2})c_{-1,3} = 0$$

$$(x_{n+2} - x_{n-1})c_{n-3,3} - (x_{n+2} + x_{n+1} - x_{n-1} - x_{n-2})c_{n-2,3} + (x_{n+1} - x_{n-2})c_{n-1,3} = 0,$$

which for equally spaced knots ($x_j = x_0 + jh$) simplify to

$$3hc_{-3,3} - 6hc_{-2,3} + 3hc_{-1,3} = 0$$

$$3hc_{n-3,3} - 6hc_{n-2,3} + 3hc_{n-1,3} = 0.$$

It is convenient to add these conditions (dividing out the h) as the first and last row of (1.41) to give

$$
(1.43)\qquad
\begin{bmatrix}
3 & -6 & 3 & & & \\
1/6 & 2/3 & 1/6 & & & \\
 & 1/6 & 2/3 & 1/6 & & \\
 & & \ddots & \ddots & \ddots & \\
 & & & 1/6 & 2/3 & 1/6 \\
 & & & 3 & -6 & 3
\end{bmatrix}
\begin{bmatrix} c_{-3,3} \\ c_{-2,3} \\ c_{-1,3} \\ \vdots \\ c_{n-2,3} \\ c_{n-1,3} \end{bmatrix}
=
\begin{bmatrix} 0 \\ f_0 \\ f_1 \\ \vdots \\ f_n \\ 0 \end{bmatrix}.
$$

This system of $n+3$ equations in $n+3$ variables has a unique solution, the natural cubic spline interpolant.

It is a useful exercise to work out the extra rows you would add to (1.41) to impose *complete* or *not-a-knot* boundary conditions.

Figure 1.25 shows the natural cubic spline interpolant to the data (1.37). Clearly this spline satisfies the interpolation conditions, but now there seems to be an artificial peak near $x = 5$ that you might not have anticipated from the data values. This is a feature

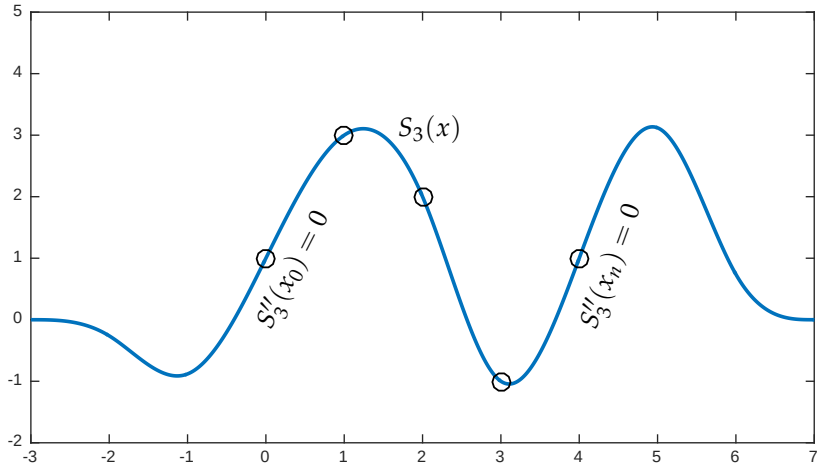


Figure 1.25: Cubic spline S_3 interpolant to 5 data points $\{(x_j, f_j)\}_{j=0}^4$, imposing the two extra *natural spline* conditions $S_3''(x_0) = S_3''(x_n) = 0$ to give a unique spline.

of the natural boundary conditions: by forcing S_3'' to be zero at x_0 and x_n, we ensure that the spline S_3 has constant slope at x_0 and x_n. Eventually this slope must be reversed, as our B-splines force $S_3(x)$ to be zero outside (x_{-3}, x_{n+3}), the support of the B-splines that contribute to the sum (1.40).

Of course, one can implement splines of higher degree, $k > 3$, if if greater continuity is required at the knots, or if there are more than two boundary conditions to impose (e.g., if one wants both first and second derivatives to be zero at the boundary). The procedure in that case follows the pattern detailed above: work out the entries in the matrix (1.36) and add in rows to encode the additional $k - 1$ constraints needed to specify a unique degree-k spline interpolant.

1.11.8 Optimality properties of splines

Splines often enjoy a beautiful property: among all sufficiently smooth interpolants , certain splines minimize 'energy', quantified for a function $g \in C^2[x_0, x_n]$ as

$$\int_{x_0}^{x_n} g''(x)^2 \, \mathrm{d}x.$$

To give a flavor for such results, we present one example.

For a similar result involving complete cubic splines, see Theorem 2.3.1 of Gautschi's *Numerical Analysis* (2nd ed., Birkhäuser, 2012). The proof here is an easy adaptation of Gautchsi's.

Theorem 1.10 (Natural cubic splines minimize energy)**.** Suppose S_3 is the natural cubic spline interpolant to $\{(x_j, f_j)\}_{j=0}^n$, and g is any $C^2[x_0, x_n]$ function that also interpolates the same data. Then

$$\int_{x_0}^{x_n} S_3''(x)^2 \, \mathrm{d}x \leq \int_{x_0}^{x_n} g''(x)^2 \, \mathrm{d}x.$$

Proof. The proof will actually quantify how much larger g'' is than S_3'' by showing that

$$\int_{x_0}^{x_n} g''(x)^2\,\mathrm{d}x = \int_{x_0}^{x_n} S_3''(x)^2\,\mathrm{d}x + \int_{x_0}^{x_n} \big(g''(x) - S_3''(x)\big)^2\,\mathrm{d}x. \tag{1.44}$$

Expanding the right-hand side, see that this claim is equivalent to

$$\int_{x_0}^{x_n} \big(g''(x) - S_3''(x)\big)S_3''(x)\,\mathrm{d}x = 0. \tag{1.45}$$

To prove this claim, break the integral on the left into segments $[x_j, x_{j+1}]$ between the knots. Write

$$\int_{x_0}^{x_n} \big(g''(x) - S_3''(x)\big)S_3''(x)\,\mathrm{d}x = \sum_{j=1}^{n} \int_{x_{j+1}}^{x_j} \big(g''(x) - S_3''(x)\big)S_3''(x)\,\mathrm{d}x.$$

This decomposition of $[x_0, x_n]$ will allow us to exploit the fact that S_3 is a standard cubic polynomial, and hence infinitely differentiable, on these subintervals.

On each subinterval, integrate by parts to obtain

$$\int_{x_{j+1}}^{x_j} \big(g''(x) - S_3''(x)\big)S_3''(x)\,\mathrm{d}x = \Big[\big(g'(x) - S_3'(x)\big)S_3''(x)\Big]_{x=x_{j-1}}^{x_j} - \int_{x_{j+1}}^{x_j} \big(g'(x) - S_3'(x)\big)S_3'''(x)\,\mathrm{d}x. \tag{1.46}$$

Focus now on the integral on the right-hand side; we can show it is *zero* by integrating it by parts to get

$$\int_{x_{j+1}}^{x_j} \big(g'(x) - S_3'(x)\big)S_3'''(x)\,\mathrm{d}x = \Big[\big(g(x) - S_3(x)\big)S_3'''(x)\Big]_{x=x_{j-1}}^{x_j} - \int_{x_{j+1}}^{x_j} \big(g(x) - S_3(x)\big)S_3''''(x)\,\mathrm{d}x. \tag{1.47}$$

The boundary term on the right is zero, since $g(x_\ell) - S_3(x_\ell) = 0$ for $\ell = 0, \ldots, n$ (both g and S_3 must interpolate the data). The integral on the right is also zero: since S_3 is a cubic polynomial on $[x_{j-1}, x_j]$, $S_3''''(x) = 0$. Thus (1.46) reduces to

$$\int_{x_{j+1}}^{x_j} \big(g''(x) - S_3''(x)\big)S_3''(x)\,\mathrm{d}x = \Big[\big(g'(x) - S_3'(x)\big)S_3''(x)\Big]_{x=x_{j-1}}^{x_j}$$

Adding up these contributions over all the subintervals,

$$\int_{x_0}^{x_n} \big(g''(x) - S_3''(x)\big)S_3''(x)\,\mathrm{d}x = \sum_{j=1}^{n} \Big[\big(g'(x) - S_3'(x)\big)S_3''(x)\Big]_{x=x_{j-1}}^{x_j}.$$

Most of the boundary terms on the right cancel one another out, leaving only

$$\int_{x_0}^{x_n} \big(g''(x) - S_3''(x)\big)S_3''(x)\,\mathrm{d}x = \Big(\big(g'(x_n) - S_3'(x_n)\big)S_3''(x_n)\Big) - \Big(\big(g'(x_0) - S_3'(x_0)\big)S_3''(x_0)\Big).$$

Each of the terms on the right is zero by virtue of the *natural* cubic spline condition $S_3''(x_0) = S_3''(x_n) = 0$. This confirms the formula (1.45), and hence the equivalent (1.44) that quantifies how much larger g'' can be than S_3''. ∎

1.11.9 *Some omissions*

The great utility of B-splines in engineering has led to the development of the subject far beyond these basic notes. Among the omissions are: interpolation imposed at points distinct from the knots, convergence of splines to the function they are approximating as the number of knots increases, integration and differentiation of splines, tension splines, etc. Splines in higher dimensions ('thin-plate splines') are used, for example, to design the panels of a car body.

1.12 *Handling Polynomials in MATLAB*

To close this discussion of interpolating polynomials, we mention a few notes about polynomials in MATLAB.

1.12.1 *MATLAB's Polynomial Format*

By convention, MATLAB represents polynomials by their coefficients, *listed by decreasing powers of* x. Thus $c_0 + c_1x + c_2x^2 + c_3x^3$ is represented by the vector

$$[c_3 \;\; c_2 \;\; c_1 \;\; c_0],$$

while $7 + 3x + 5x^3 - 2x^4$ would be represented by

$$[-2 \;\; 5 \;\; 0 \;\; 3 \;\; 7]$$

In this last example note the `0` corresponding to the x^2 term: all lower powers of x must be accounted for in coefficient vector.

Given a polynomial in a vector, say `p = [ -2 5 0 3 7]`, one can evaluate $p(x)$ using the command `polyval`, e.g.

```
>> polyval(p,x)
```

This command works if `x` is a scalar or a vector. Thus, for example, to plot $p(x)$ for $x \in [0, 1]$, one could use

```
>> x = linspace(0,1,500);    % 500 uniform points between 0 and 1
>> plot(x,polyval(p,x))      % plot p(x) with x from 0 to 1
```

One can also compute the roots of polynomials very easily with the command

```
>> roots(p)                  % compute roots of p(x)=0
```

though one should be cautious of numerical errors when the degree of the polynomial is large. One can construct a polynomial directly from its roots, using the `poly` command. For example,

```
>> poly([1:4])
ans =
     1   -10    35   -50    24
```

Type `type roots` to see MATLAB's code for the `roots` command. Scan to the bottom to see the crucial lines. From the coefficients MATLABconstructs a companion matrix, then computes its eigenvalues using the `eig` command. For some (larger degree) polynomials, these eigenvalues are very sensitive to perturbations, and the roots can be very inaccurate. For a famous example due to Wilkinson, try `roots(poly[1:24]))`, should return the roots $1, \ldots, 24$.

`poly` returns the coefficients of the monic polynomial with roots $1, 2, 3, 4$:

$$24 - 50x + 35x^2 - 10x^3 + x^4 = (x-1)(x-2)(x-3)(x-4).$$

This gives a slick way to construct the Lagrange basis function

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

given the vector $\texttt{xx} = [x_0 \ \cdots \ x_n]$ of interpolation points:

```
>> ell_j = poly(xx([1:j j+2:end]));         % specify roots of ell_j
>> ell_j = ell_j/polyval(ell_j,xx(j+1));    % scale so ell_j(xx(j+1)) = 1
```

Note that the indices of `xx` account for the fact that $x_j = \texttt{xx(j+1)}$.

1.12.2 *Constructing Polynomial Interpolants*

MATLABhas a built-in code for constructing polynomial interpolants. In fact, it is a special case of the polynomial approximation code `polyfit`. When you request that `polyfit` produce a degree-n polynomial through $n+1$ pairs of data, you obtain an interpolant. For example, the following code will interpolate $f(x) = sqrt(x)$ at $x_j = j/4$ for $j = 0, \ldots, 4$:

Beware! The numerical implementation of `polyfit` is not ideal for polynomial interpolation: the code uses the Vandermonde basis. Thus, restrict your use of `polyfit` to low degree polynomials. The command `type polyfit` will show you MATLAB's code.

```
>> f  = @(x) sqrt(x);          % define f
>> xx = [0:4]/4;               % define interpolation points
>> p = polyfit(xx,f(xx),4);    % quartic polynomial interpolant
>> polyval(p,xx)               % evaluate p at interpolation points
ans =
    0.0000    0.5000    0.7071    0.8660    1.0000
>> f(xx)                       % compare to f at interpolation points
ans =
         0    0.5000    0.7071    0.8660    1.0000
```

1.12.3 *Piecewise Polynomial Interpolants and Splines*

MATLAB also includes a general-purpose `interp1` command that constructs various piecewise polynomial interpolants. For example, the `'linear'` option constructs piecewise linear interpolants.

```
>> f  = @(x) sin(3*pi*x);               % define f
>> xx = [0:10]/10;                      % define "knots"
>> x  = linspace(0,1,500);              % evaluation points
>> plot(x,interp1(xx,f(xx),x,'linear'))  % plot piecewise linear interpolant
```

Alternatively, the `'spline'` option constructs the not-a-knot cubic spline approximation.

```
>> plot(x,interp1(xx,f(xx),x,'spline'))  % plot cubic spline interpolant
```

The `spline` command (which `interp1` uses to construct the spline) will return MATLAB's data structure that stores the cubic spline interpolant. `>> S = spline(xx,f(xx))`

```
S =
      form: 'pp'
    breaks: [1x11 double]
     coefs: [10x4 double]
    pieces: 10
     order: 4
       dim: 1
```

For example, `S.breaks` contains the list of knots. One can also pass arguments to `spline` to specify *complete* boundary conditions. However, there is no easy way to impose natural boundary conditions. For more sophisticated data fitting operations, MATLAB offers a Curve Fitting Toolbox (which fits both curves and surfaces).

Another option to `interp1` has a misleading name: `'pchip'` constructs a particular spline-like interpolant designed to be quite smooth: it cannot match any derivative information about f, as no derivative information is even passed to the function.

1.12.4 Chebfun

Chebfun is a free package of MATLAB routines developed by Nick Trefethen and colleagues at Oxford University. Using sophisticated techniques from polynomial approximation theory, Chebfun automatically represents an arbitrary (piecewise smooth) function $f(x)$ to machine precision, and allows all manner of operations on this function, overloading every conceivable MATLAB matrix/vector operation. There is no way to do this beautiful and powerful system justice in a few lines of text here. Go to `chebfun.org`, download the software, and start exploring. Suffice to say, Chebfun significantly enrich one's study and practice of numerical analysis.

In fact, it was used to generate a number of the plots in these notes.

2
Approximation Theory

LECTURE 12: *Introduction to Approximation Theory*

INTERPOLATION IS AN INVALUABLE TOOL in numerical analysis: it provides an easy way to replace a complicated function by a polynomial (or piecewise polynomial), and, *at least as importantly*, it provides a mechanism for developing numerical algorithms for more sophisticated problems. Interpolation is not the only way to approximate a function, though: and indeed, we have seen that the quality of the approximation can depend perilously on the choice of interpolation points.

We saw one example in Section 1.7: finite difference formulas for approximating derivatives and solving differential equation boundary value problems. Several other applications will follow later in the semester.

If approximation is our goal, interpolation is only one means to that end. In this chapter we investigate alternative approaches that directly optimize the quality of the approximation. How do we measure this quality? That depends on the application. Perhaps the most natural means is to *minimize the maximum error* of the approximation.

> Given $f \in C[a,b]$, find $p_* \in \mathcal{P}_n$ such that
>
> $$\max_{x\in[a,b]} |f(x) - p_*(x)| = \min_{p\in\mathcal{P}_n} \max_{x\in[a,b]} |f(x) - p(x)|.$$

This is called the *minimax approximation problem*.

Norms clarify the notation. For any $g \in C[a,b]$, define

$$\|g\|_\infty := \max_{x\in[a,b]} |g(x)|,$$

the 'infinity norm of g'. One can show that $\|\cdot\|_\infty$ satisfies the basic norm axioms on the vector space $C[a,b]$ of continuous functions. Thus the minimax approximation problem seeks $p_* \in \mathcal{P}_n$ such that

$$\|f - p_*\|_\infty = \min_{p\in\mathcal{P}_n} \|f - p\|_\infty.$$

$\|g\|_\infty \geq 0$ for all $g \in C[a,b]$
$\|g\|_\infty = 0 \iff g(x) = 0$ for all $x \in [a,b]$.
$\|\alpha g\|_\infty = |\alpha|\|g\|_\infty$ for all $\alpha \in \mathbb{C}$, $g \in C[a,b]$.
$\|g+h\|_\infty \leq \|g\|_\infty + \|h\|_\infty$, for all $g, h \in C[a,b]$.

Notice that, for better or worse, this approximation will be heavily influenced by extreme values of $f(x)$, even if they occur over only a small range of $x \in [a,b]$.

Some applications call instead for an approximation that balances the size of the errors against the range of x values over which they are attained. In such cases it is most common to minimize the integral of the square of the error, the *least squares approximation problem*.

> Given $f \in C[a,b]$, find $p_* \in \mathcal{P}_n$ such that
>
> $$\Big(\int_a^b (f(x) - p_*(x))^2 \, \mathrm{d}x \Big)^{1/2} = \min_{p \in \mathcal{P}_n} \Big(\int_a^b (f(x) - p(x))^2 \, \mathrm{d}x \Big)^{1/2}.$$

This problem is often associated with *energy minimization* in mechanics, giving one motivation for its widespread appeal. As before, we express this more compactly by introducing the *two-norm* of $g \in [a,b]$:

$$\|g\|_2 = \Big(\int_a^b |g(x)|^2 \, \mathrm{d}x \Big)^{1/2},$$

so the least squares problem becomes

$$\|f - p_*\|_2 = \min_{p \in \mathcal{P}_n} \|f - p\|_2.$$

This chapter focuses on these two problems. Before attacking them we mention one other possibility, minimizing the absolute value of the integral of the error: the *least absolute deviations* problem.

> Given $f \in C[a,b]$, find $p_* \in \mathcal{P}_n$ such that
>
> $$\int_a^b |f(x) - p_*(x)| \, \mathrm{d}x = \min_{p \in \mathcal{P}_n} \int_a^b |f(x) - p(x)| \, \mathrm{d}x.$$

This problem has become quite important in recent years. In particular, the analogous problem resulting when f is replaced by its vector discretization $\mathbf{f} \in \mathbb{C}^n$ plays a pivotal role in *compressive sensing*.

With this problem we associate the *one-norm* of $g \in C[a,b]$,

$$\|g\|_1 = \int_a^b |g(x)| \, \mathrm{d}x,$$

giving the least absolute deviations problem as

$$\|f - p_*\|_1 = \min_{p \in \mathcal{P}_n} \|f - p\|_1.$$

2.1 *Minimax Approximation: General Theory*

The goal of minimizing the maximum error of a polynomial p from the function $f \in C[a,b]$ is called *minimax* (or *uniform*, or L^∞) approximation: Find $p_* \in \mathcal{P}_n$ such that

$$\|f - p_*\|_\infty = \min_{p \in \mathcal{P}_n} \|f - p\|_\infty.$$

Let us begin by connecting this problem to polynomial interpolation. On Problem Set 2 you were asked to prove that

$$\|f - \mathbf{\Pi}_n f\|_\infty \leq \big(1 + \|\mathbf{\Pi}_n\|_\infty\big)\|f - p_*\|_\infty, \tag{2.1}$$

where $\mathbf{\Pi}_n$ is the linear interpolation operator for

That is, $p = \mathbf{\Pi}_n f \in \mathcal{P}_n$ is the polynomial that interpolates f at $x_0, \ldots, x_n$.

$$x_0 < x_1 < \cdots < x_n$$

with $x_0, \ldots, x_n \in [a,b]$. Here $\|\mathbf{\Pi}_n\|_\infty$ is the *operator norm* of $\mathbf{\Pi}_n$:

$$\|\mathbf{\Pi}_n\|_\infty = \max_{f \in C[a,b]} \frac{\|\mathbf{\Pi}_n f\|_\infty}{\|f\|_\infty}$$

You further show that

$$\|\mathbf{\Pi}_n\| = \max_{x \in [a,b]} \sum_{j=0}^{n} |\ell_j(x)|,$$

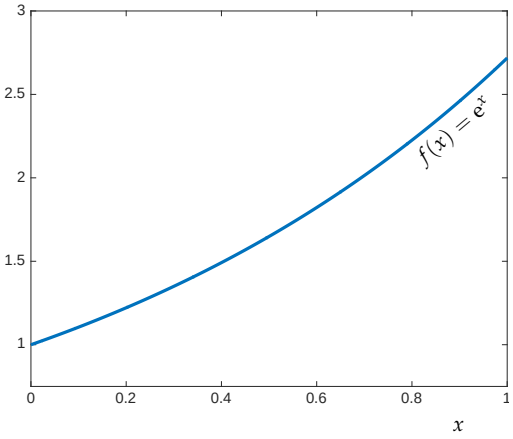
where ℓ_j denotes the jth Lagrange interpolation basis function

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

Now appreciate the utility of bound (2.1): the linear interpolant $\mathbf{\Pi}_n f$ (*which is easy to compute*) is within a factor of $1 + \|\mathbf{\Pi}_n\|_\infty$ of the optimal approximation p_*. Note that $\|\mathbf{\Pi}_n\|_\infty \geq 1$: how much larger than one depends on the distribution of the interpolation points.

In the following sections we shall characterize and compute p_* (indeed more difficult than computing the interpolant), then use the theory of minimax approximation to find an excellent set of *almost fail-safe* interpolation points.

We begin by working out a simple example by hand.



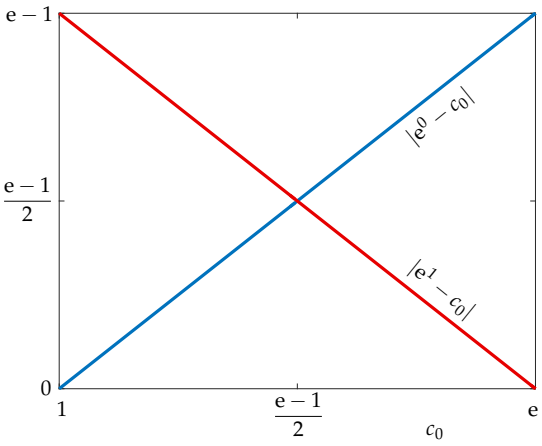
Example 2.1. Suppose we seek the constant that best approximates $f(x) = e^x$ over the interval $[0,1]$, shown in the margin. Before going on, sketch out a constant function (degree-0 polynomial) that approximates f in a manner that *minimizes the maximum error*.

Since $f(x)$ increases monotonically for $x \in [0,1]$, the optimal constant approximation $p_* = c_0$ must fall between $f(0) = 1$ and

$f(1) = \mathrm{e}$, i.e., $1 \le c_0 \le \mathrm{e}$. Moreover, since f is monotonic and p_* is a constant, the function $f - p_*$ is also monotonic, so the maximum error $\max_{x\in[a,b]} |f(x) - p_*(x)|$ must be attained at one of the end points, $x = 0$ or $x = 1$. Thus,

$$\|f - p_*\|_\infty = \max\{|\mathrm{e}^0 - c_0|, |\mathrm{e}^1 - c_0|\}.$$

The picture to the right shows $|\mathrm{e}^0 - c_0|$ (blue) and $|\mathrm{e}^1 - c_0|$ (red) for $c_0 \in [1, \mathrm{e}]$. The optimal value for c_0 will be the point at which *the larger of these two lines is minimal*. The figure clearly reveals that this happens when the errors are equal, at $c_0 = (1 + \mathrm{e})/2$. We conclude that the optimal minimax constant polynomial approximation to e^x on $x \in [0, 1]$ is $p_*(x) = c_0 = (1 + \mathrm{e})/2$.



The plots in Figure 2.1 compare f to the optimal polynomial p_* (top), and show the error $f - p_*$ (bottom). We picked c_0 so that the error $f - p_*$ was equal in magnitude at the end points $x = 0$ and $x = 1$; in fact, it is equal in magnitude, but opposite in sign,

$$\mathrm{e}^0 - c_0 = -(\mathrm{e}^1 - c_0).$$

This property—maximal error being attained with alternating sign—is a key feature of minimax approximation.

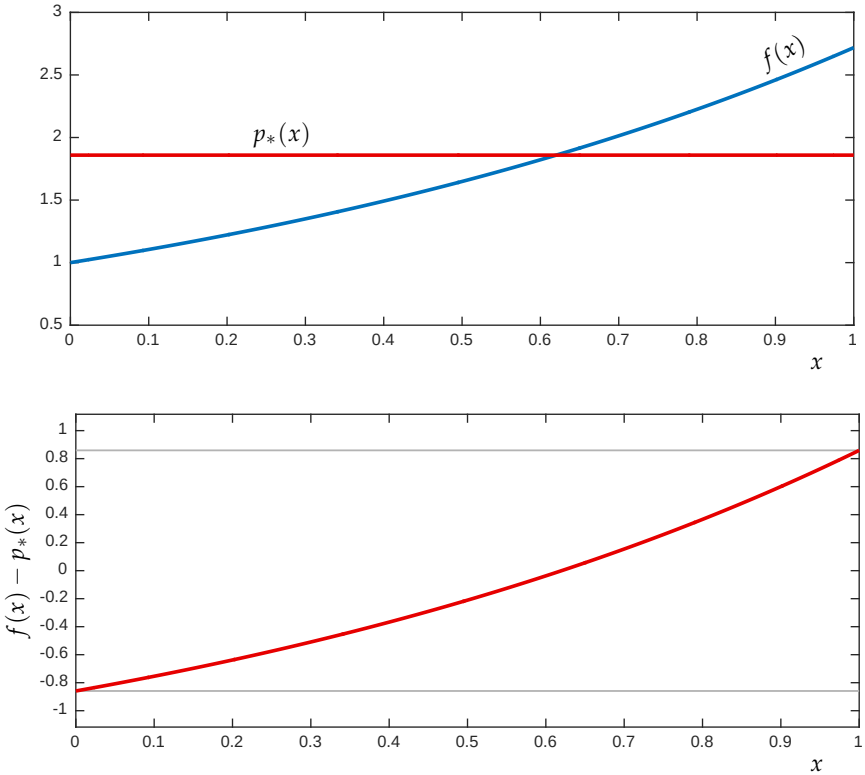


Figure 2.1: Minimax approximation of degree $k = 0$ to $f(x) = \mathrm{e}^x$ on $x \in [0, 1]$. The top plot compares f to p_*; the bottom plot shows the error $f - p_*$, whose extreme magnitude is *attained, with opposite sign, at two values of* $x \in [0, 1]$.

LECTURE 13: *Equioscillation, Part 1*

2.2 *Oscillation Theorem*

The previous example hints that the points at which the error $f - p_*$ attains its maximum magnitude play a central role in the theory of minimax approximation. The Theorem of de la Vallée Poussin is a first step toward such a result. We include its proof to give a flavor of how such results are established.

The proof is adapted from Section 8.3 of Süli and Mayers, *An Introduction to Numerical Analysis* (Cambridge, 2003).

Theorem 2.1 (de la Vallée Poussin's Theorem)**.**
Let $f \in C[a,b]$ and suppose $r \in \mathcal{P}_n$ is some polynomial for which there exist $n+2$ points $\{x_j\}_{j=0}^{n+1}$ with $a \le x_0 < x_1 < \cdots < x_{n+1} \le b$ at which the error $f(x) - r(x)$ oscillates signs, i.e.,

$$\operatorname{sgn}(f(x_j) - r(x_j)) = -\operatorname{sgn}(f(x_{j+1}) - r(x_{j+1}))$$

$$\operatorname{sgn}(x) = \begin{cases} 1, & x > 0; \\ 0, & x = 0; \\ -1, & x < 0. \end{cases}$$

for $j = 0, \ldots, n$. Then

$$\min_{p \in \mathcal{P}_n} \|f - p\|_\infty \ge \min_{0 \le j \le n+1} |f(x_j) - r(x_j)|. \tag{2.2}$$

Before proving this result, look at Figure 2.2 for an illustration of the theorem. Suppose we wish to approximate $f(x) = \mathrm{e}^x$ with some quintic polynomial, $r \in \mathcal{P}_5$ (i.e., $n = 5$). *This polynomial is not necessarily the minimax approximation to f over the interval* $[0,1]$. However, in the figure it is clear that for this r, we can find $n+2 = 7$ points at which the sign of the error $f(x) - r(x)$ oscillates. The red curve shows the error for the optimal minimax polynomial p_* (whose computation is discussed below). This is the point of de la Vallée Poussin's theorem: *Since the error $f(x) - r(x)$ oscillates sign $n+2$ times, the minimax error $\pm\|f - p_*\|_\infty$ exceeds $|f(x_j) - r(x_j)|$ at one of the points x_j that give the oscillating sign.* In other words, de la Vallée Poussin's theorem gives a nice mechanism for developing *lower bounds* on $\|f - p_*\|_\infty$.

These $n+2$ points are by no means unique: we have a continuum of choices available. However, taking the extrema of $f - r$ will give the the best bounds in the theorem.

Proof. Suppose we have $n+2$ ordered points, $\{x_j\}_{j=0}^{n+1} \subset [a,b]$, such that $f(x_j) - r(x_j)$ alternates sign at consecutive points, and let p_* denote the minimax polynomial,

$$\|f - p_*\|_\infty = \min_{p \in \mathcal{P}_n} \|f - p\|_\infty.$$

We will prove the result by contradiction. Thus suppose

$$\|f - p_*\|_\infty < |f(x_j) - r(x_j)|, \qquad \text{for all } j = 0, \ldots, n+1. \tag{2.3}$$

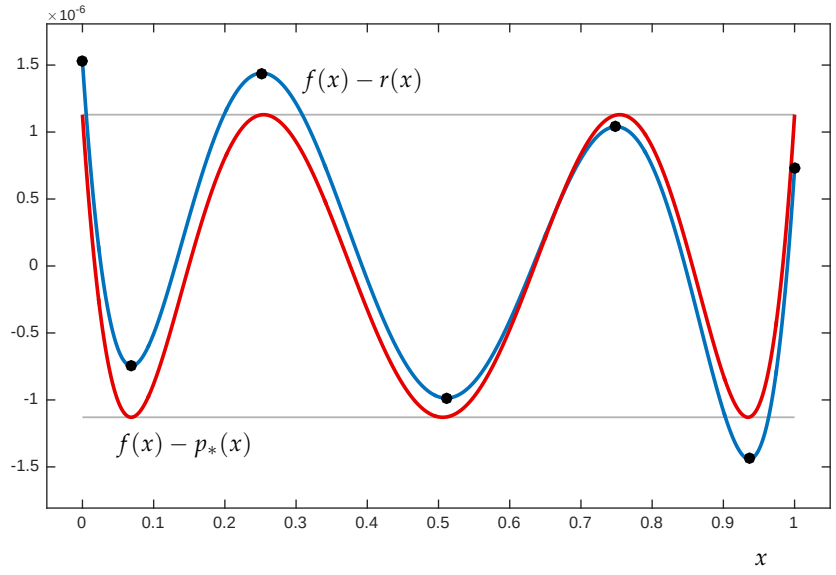


Figure 2.2: Illustration of de la Vallée Poussin's theorem for $f(x) = e^x$ and $n = 5$. Some polynomial $r \in \mathcal{P}_5$ gives an error $f - r$ for which we can identify $n + 2 = 7$ points x_j, $j = 0, \ldots, n + 1$ (black dots) at which $f(x_j) - r(x_j)$ oscillates sign. The minimum value of $|f(x_j) - r(x_j)|$ gives a *lower bound* the maximum error $\|f - p_*\|_\infty$ of the optimal approximation $p_* \in \mathcal{P}_5$.

As the left hand side is the maximum difference of $f - p_*$ over all $x \in [a, b]$, that difference can be no larger at $x_j \in [a, b]$, and so:

$$|f(x_j) - p_*(x_j)| < |f(x_j) - r(x_j)|, \qquad \text{for all } j = 0, \ldots, n + 1. \tag{2.4}$$

Now consider

$$p_*(x) - r(x) = (f(x) - r(x)) - (f(x) - p_*(x)),$$

which is a degree n polynomial, since $p_*, r \in \mathcal{P}_n$. Equation (2.4) states that $f(x_j) - r(x_j)$ always has larger magnitude than $f(x_j) - p_*(x_j)$. Thus, regardless of the sign of $f(x_j) - p_*(x_j)$, the magnitude $|f(x_j) - p_*(x_j)|$ will never be large enough to overcome $|f(x_j) - r(x_j)|$, and hence

$$\operatorname{sgn}(p_*(x_j) - r(x_j)) = \operatorname{sgn}(f(x_j) - r(x_j)).$$

We know from the hypothesis that $f(x) - r(x)$ must change sign at least $n + 1$ times (at least once in each interval (x_j, x_{j+1}) for $j = 0, \ldots, n$), and thus the degree-n polynomial $p_* - r$ must do the same. But $n + 1$ sign changes implies $n + 1$ roots; the only degree-n polynomial with $n + 1$ roots is the zero polynomial, i.e., $p_* = r$. However, this contradicts the strict inequality in equation (2.3). Hence, there must be at least one j for which

$$\|f - p_*\|_\infty \geq |f(x_j) - r(x_j)|,$$

thus yielding the theorem. ■

Now suppose we can find some degree-n polynomial, call it $\widetilde{r} \in \mathcal{P}_n$, and $n + 2$ points $x_0 < \cdots < x_{n+1}$ in $[a, b]$ such that not only does

the sign of $f - \widetilde{r}$ oscillate, but the error takes its extremal values at these points. That is,

$$|f(x_j) - \widetilde{r}(x_j)| = \|f - \widetilde{r}\|_\infty, \quad j = 0, \ldots, n+1,$$

and

$$f(x_j) - \widetilde{r}(x_j) = -\big(f(x_{j+1}) - \widetilde{r}(x_{j+1})\big), \quad j = 0, \ldots, n.$$

The red curve in Figure 2.2 shows an error function that satisfies these requirements.

Now apply de la Valée Poussin's theorem to this special polynomial $\widetilde{r}$. Equation (2.2) gives

$$\min_{p \in \mathcal{P}_n} \|f - p\| \geq \min_{0 \leq j \leq n+1} |f(x_j) - \widetilde{r}(x_j)|.$$

On the other hand, we have presumed that

$$|f(x_j) - \widetilde{r}(x_j)| = \|f - \widetilde{r}\|_\infty$$

for all $j = 0, \ldots, n+1$. Thus, by de la Vallée Poussin's theorem,

$$\min_{p \in \mathcal{P}_n} \|f - p\| \geq \min_{0 \leq j \leq n+1} |f(x_j) - \widetilde{r}(x_j)| = \|f - \widetilde{r}\|_\infty.$$

Since $\widetilde{r} \in \mathcal{P}_n$, it follows that

$$\min_{p \in \mathcal{P}_n} \|f - p\| = \|f - \widetilde{r}\|_\infty,$$

and this *equioscillating* $\widetilde{r}$ must be an optimal approximation to f.

The question remains: Does such a polynomial with equioscillating error always exist? The following theorem ensures it does.

For a direct proof, see Section 8.3 of Süli and Mayers. Another excellent resource is G. W. Stewart, *Afternotes Goes to Graduate School*, SIAM, 1998; see Stewart's Lecture 3.

Theorem 2.2 (Oscillation Theorem)**.** Suppose $f \in C[a, b]$. Then $p_* \in \mathcal{P}_n$ is a minimax approximation to f from $\mathcal{P}_n$ on $[a, b]$ *if and only if* there exist $n+2$ points $x_0 < x_1 < \cdots < x_{n+1}$ such that

$$|f(x_j) - p_*(x_j)| = \|f - p_*\|_\infty, \quad j = 0, \ldots, n+1$$

and the sign of the error oscillates at these points:

$$f(x_j) - p_*(x_j) = -\big(f(x_{j+1}) - p_*(x_{j+1})\big), \quad j = 0, \ldots, n.$$

Note that this result is *if and only if*: the oscillation property exactly *characterizes* the minimax approximation. We have proved one direction already by appeal to de la Vallée Poussin's theorem. The proof of the other direction is rather more involved.

LECTURE 14: *Equioscillation, Part 2*

A direct proof that an optimal minimax approximation $p_* \in \mathcal{P}_n$ must give an equioscillating error is rather tedious, requiring one the chase down the oscillation points one at a time. The following approach is a bit more appealing. We begin with a technical result from which the main theorem will readily follow.

Lemma 2.1. Let $p_* \in \mathcal{P}_n$ be a minimax approximation of $f \in C[a,b]$,

$$\|f - p_*\|_\infty = \min_{p \in \mathcal{P}_n} \|f - p\|_\infty,$$

and let $\mathcal{X}$ denote the set of all points $x \in [a,b]$ for which

$$|f(x) - p_*(x)| = \|f - p_*\|_\infty.$$

Then for all $q \in \mathcal{P}_n$,

$$\max_{x \in \mathcal{X}} \big(f(x) - p_*(x)\big) q(x) \geq 0. \tag{2.5}$$

This 'lemma' is a diluted version of *Kolmolgorov's Theorem*, which is (a) an 'if and only if' version of this lemma that (b) appeals to approximation with much more general classes of functions, not just polynomials, and (c) handles complex-valued functions. The proof here is adapted from that more general setting given in Theorem 2.1 of DeVore and Lorentz, *Constructive Approximation* (Springer, 1993).

Proof. We will prove the lemma by contradiction. Suppose $p_* \in \mathcal{P}_n$ is a minimax approximation, but that (2.5) fails to hold, i.e., there exists some $\widetilde{q} \in \mathcal{P}_n$ and $\varepsilon > 0$ such that

$$\max_{x \in \mathcal{X}} \big(f(x) - p_*(x)\big) \widetilde{q}(x) < -2\varepsilon.$$

We first note that $\|\widetilde{q}\|_\infty > 0$. Since $\big(f(x) - p_*(x)\big)q(x)$ is a continuous function on $[a,b]$, it must remain negative on some sufficiently small neighborhood of $\mathcal{X}$. More concretely, we can find $\delta > 0$ such that

$$\max_{x \in \widetilde{\mathcal{X}}} \big(f(x) - p_*(x)\big) \widetilde{q}(x) < -\varepsilon, \tag{2.6}$$

where

$$\widetilde{\mathcal{X}} := \{\xi \in [a,b] : \min_{x \in \mathcal{X}} |\xi - x| < \delta\}.$$

To arrive at a contradiction, we will design a function $\widetilde{p}$ that better approximates f than p_*, i.e., $\|f - \widetilde{p}\|_\infty < \|f - p_*\|_\infty$. This function will take the form

$$\widetilde{p}(x) = p_*(x) - \lambda \widetilde{q}(x)$$

for (small) constant λ we shall soon determine. Let $E := \|f - p_*\|_\infty$ and pick M such that $|\widetilde{q}(x)| \leq M$ for all $x \in \widetilde{\mathcal{X}}$. Then for all $x \in \widetilde{\mathcal{X}}$,

$$\begin{aligned} |f(x) - \widetilde{p}(x)|^2 &= \big(f(x) - p_*(x)\big)^2 + 2\lambda \big(f(x) - p_*(x)\big)\widetilde{q}(x) + \lambda^2 \widetilde{q}(x)^2 \\ &= E^2 + 2\lambda \big(f(x) - p_*(x)\big)\widetilde{q}(x) + \lambda^2 \widetilde{q}(x)^2 \\ &< E^2 - 2\lambda\varepsilon + \lambda^2 M^2, \end{aligned} \tag{2.7}$$

where this inequality follows from (2.6). To show that $\widetilde{p}$ is a better approximation to f than p_*, it will suffice to show that the right-hand side of (2.7) is smaller than E^2: note that for any $\lambda \in (0, 2\varepsilon/M^2)$, then

$$|f(x) - \widetilde{p}(x)|^2 < E^2 - 2\lambda\varepsilon + \lambda^2 M^2 < E^2 - \frac{4\varepsilon^2}{M^2} + \frac{4\varepsilon^2}{M^2} = E^2 = \|f - p_*\|^2 \tag{2.8}$$

for all $x \in \widetilde{\mathcal{X}}$. Thus $\widetilde{p}$ beats p_* on $\widetilde{\mathcal{X}}$. Now since $\mathcal{X}$ comprises the points where $|f(x) - p_*(x)|$ attains its maximum, away from $\widetilde{\mathcal{X}}$ this error must be bounded away from its maximum, i.e., there exists some $\eta > 0$ such that

$$\max_{\substack{x \in [a,b] \\ x \notin \widetilde{\mathcal{X}}}} |f(x) - p_*(x)| \leq E - \eta.$$

Now we want to show that $|f(x) - \widetilde{p}(x)| < E$ for these $x \notin \widetilde{\mathcal{X}}$ as well. In particular, for such x

$$\begin{aligned} |f(x) - \widetilde{p}(x)| &= |f(x) - p_*(x) + \lambda\widetilde{q}(x)| \\ &\leq |f(x) - p_*(x)| + \lambda|\widetilde{q}(x)| \\ &\leq E - \eta + \lambda\|\widetilde{q}\|_\infty, \end{aligned}$$

and so if $\lambda \in (0, \eta/\|\widetilde{q}\|_\infty)$,

$$|f(x) - \widetilde{p}(x)| < E - \eta + \frac{\eta}{\|\widetilde{q}\|_\infty}\|\widetilde{q}\|_\infty = E.$$

In conclusion, if

$$\lambda \in \Big(0, \min(2\varepsilon/M^2, \eta/\|\widetilde{q}\|_\infty)\Big),$$

then we constructed $\widetilde{p}(x) := p_*(x) - \lambda\widetilde{q}(x)$ such that

$$|f(x) - \widetilde{p}(x)| < E \quad \text{for all } x \in [a,b],$$

i.e., $\|f - \widetilde{p}\|_\infty < \|f - p_*\|$, contradicting the optimality of p_*. ∎

With this lemma, we can readily complete the proof of the Oscillation Theorem.

Completion of the Proof of the Oscillation Theorem. We must show that if p_* is a minimax approximation to f, then there exist $n+2$ points in $[a,b]$ on which the error $f - p_*$ changes sign. If $p_* = f$, the result holds trivially. Suppose then that $\|f - p_*\|_\infty > 0$. In the language of Lemma 2.1, we need to show that (a) the set $\mathcal{X}$ contains (at least) $n+2$ points and (b) the error oscillates sign at these points. Suppose this is not the case, i.e., we cannot identify $n+2$ consecutive points in $\mathcal{X}$ at which the error oscillates in sign. Suppose we can only identify

Recall that $\mathcal{X}$ contains all the points $x \in [a,b]$ for which the maximum error is attained: $|f(x) - p_*(x)| = \|f - p_*\|_\infty$.

m such points, $1 \le m < n+2$, which we label $x_0 < \cdots < x_{m-1}$. We will show how to construct a q that violates Lemma 2.1.

If $m = 1$, $f(x) - p_*(x)$ has the same sign for all $x \in \mathcal{X}$. Set $q(x) = -\text{sgn}(f(x_0) - p_*(x_0))$ (a constant, hence in $\mathcal{P}_n$), so that $(f(x) - p_*(x))q(x) < 0$ for all $x \in \mathcal{X}$, contradicting Lemma 2.1.

If $m > 1$, the between each consecutive pair of these m points one can then identify $\widetilde{x}_1, \ldots, \widetilde{x}_{m-1}$ where the error changes sign. (See the sketch in the margin.) Then define

$$q(x) = \pm(x - \widetilde{x}_1)(x - \widetilde{x}_2)\cdots(x - \widetilde{x}_{m-1}).$$

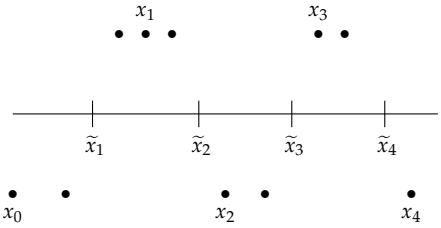
Since $m < n+2$ by assumption, $m - 1 \le n$, i.e., $q \in \mathcal{P}_n$, so Lemma 2.1 should hold with this choice of q. Since the sign of $q(x)$ does not change between its roots, it does not change within the intervals

$$(a, \widetilde{x}_1), \quad (\widetilde{x}_1, \widetilde{x}_2), \quad \cdots, \quad (\widetilde{x}_{m-2}, \widetilde{x}_{m-1}), \quad (\widetilde{x}_{m-1}, b),$$

and the sign of q flips between each of these intervals. Thus the sign of $\big(f(x) - p_*(x)\big)q(x)$ is the same for all $x \in \mathcal{X}$. Pick the $\pm$ sign in the definition of q such that

$$\big(f(x) - p_*(x)\big)q(x) < 0 \quad \text{for all } x \in \mathcal{X},$$

thus contradicting Lemma 2.1. Hence, there must be (at least) $n+2$ consecutive points in $\mathcal{X}$ at which the error flips sign. ∎



sketch for $m = 5$
$\bullet = f(x) - p_*(x)$ for $x \in \mathcal{X}$

Thus far we have been careful to only speak of *a* minimax approximation, rather than *the* minimax approximation. In fact, the later terminology is more precise, for the minimax approximant is unique.

Theorem 2.3 (Uniqueness of minimax approximant)**.**
The minimax approximant $p_* \in \mathcal{P}_n$ of $f \in C[a,b]$ over the interval $[a,b]$ is unique.

The proof is a straightforward application of the Oscillation Theorem. Suppose p_1 and p_2 are both minimax approximations from $\mathcal{P}_n$ to f on $[a,b]$. Then one can show that $(p_1 + p_2)/2$ is also a minimax approximation. Apply the Oscillation Theorem to obtain $n+2$ points at which the error for $(p_1 + p_2)/2$ oscillates sign. One can show that these points must also be oscillation points for p_1 and p_2, and that p_1 and p_2 agree at these $n+2$ points. Polynomials of degree n that agree at $n+2$ points must be the same.

For the full details of this proof, see Theorem 8.5 in Süli and Mayers.

This oscillation property forms the basis of algorithms that find the minimax approximation: iteratively adjust an approximating polynomial until it satisfies the oscillation property. The most famous algorithm for computing the minimax approximation is called the

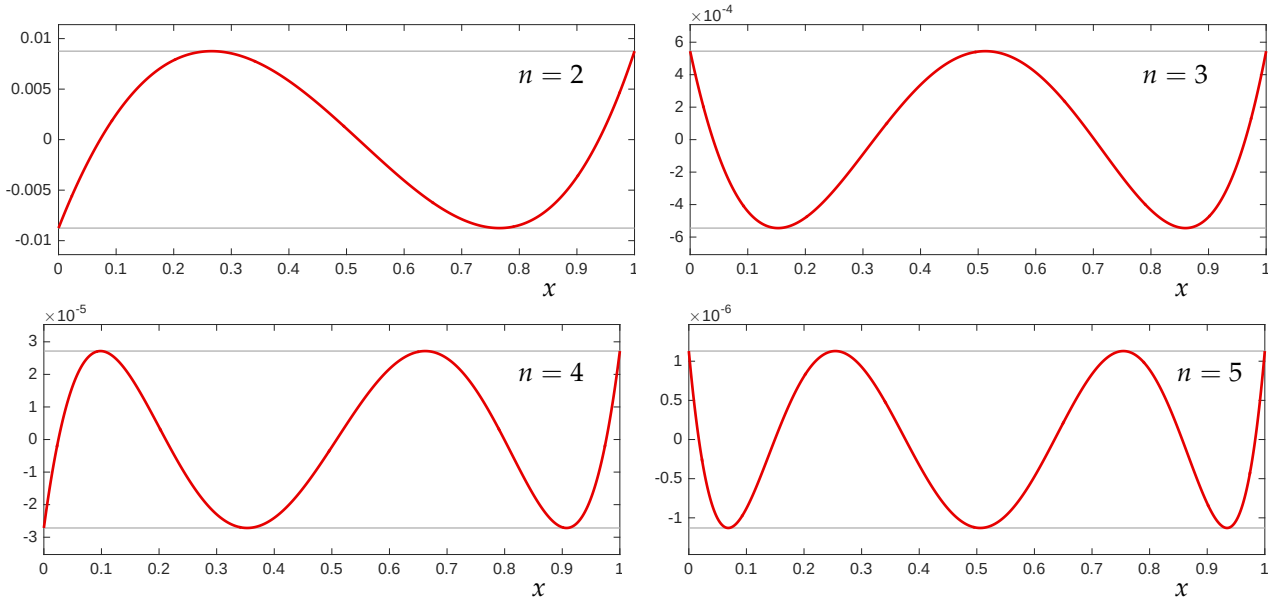


Figure 2.3: Illustration of the equioscillating minimax error $f - p_*$ for approximations of degree $n = 2, 3, 4$, and 5 with $f(x) = e^x$ for $x \in [a, b]$. In each case, the error attains its maximum with alternating sign at $n + 2$ points.

Remez exchange algorithm, essentially a specialized linear programming procedure. In exact arithmetic, this algorithm is guaranteed to terminate with the correct answer in finitely many operations.

The oscillation property is demonstrated in the Example 2.1, where we approximated $f(x) = e^x$ with a constant. Indeed, the maximum error is attained at two points (that is, $n + 2$, since $n = 0$), and the error differs in sign at those points. Figure 2.3 shows the errors $f(x) - p_*(x)$ for minimax approximations p_* of increasing degree. The oscillation property becomes increasingly apparent as the polynomial degree increases. In each case, there are $n + 2$ extreme points of the error, where n is the degree of the approximating polynomial.

These examples were computed in MATLAB using the Chebfun package's `remez` algorithm. For details, see `www.chebfun.org`.

Example 2.2 (e^x revisited). Now we shall use the Oscillation Theorem to compute the optimal linear minimax approximation to $f(x) = e^x$ on $[0, 1]$. Assume that the minimax polynomial $p_* \in \mathcal{P}_1$ has the form $p_*(x) = \alpha + \beta x$. Since f is convex, a quick sketch of the situation suggests the maximal error will be attained at the end points of the interval, $x_0 = 0$ and $x_2 = 1$. We assume this to be true, and seek some third point $x_1 \in (0, 1)$ that attains the same maximal error, δ, but with opposite sign. If we can find such a point, then the Oscillation Theorem guarantees that the resulting polynomial is optimal, confirming our assumption that the maximal error was attained at the ends of the interval.

This scenario suggests the following three equations:

$$\begin{aligned} f(x_0) - p_*(x_0) &= \delta \\ f(x_1) - p_*(x_1) &= -\delta \\ f(x_2) - p_*(x_2) &= \delta. \end{aligned}$$

Substituting the values $x_0 = a$, $x_2 = b$, and $p_*(x) = \alpha + \beta x$, these equations become

$$\begin{aligned} 1 - \alpha &= \delta \\ \mathrm{e}^{x_1} - \alpha - \beta x_1 &= -\delta \\ \mathrm{e} - \alpha - \beta &= \delta. \end{aligned}$$

The first and third equation together imply $\beta = \mathrm{e} - 1$. We also deduce that $2\alpha = \mathrm{e}^{x_1} - x_1(\mathrm{e} - 1) + 1$. A variety of choices for x_1 will satisfy these conditions, but in those cases δ will not be the *maximal error*. We must ensure that

$$|\delta| = \max_{x \in [a,b]} |f(x) - p_*(x)|.$$

To make this happen, require that *the derivative of error* be zero at x_1, reflecting that the error $f - p_*$ attains a local minimum/maximum at x_1. (The plots in Figure 2.3 confirm that this is reasonable.) Imposing the condition that $f'(x_1) - p_*'(x_1) = 0$ yields

This requirement need not hold at the points x_0 and x_2, since these points are on the ends of the interval $[a, b]$; it is only required at the interior points where the extreme error is attained, $x_j \in (a, b)$.

$$\mathrm{e}^{x_1} - \beta = 0.$$

Now we can explicitly solve the equations to obtain

$$\begin{aligned} \alpha &= \tfrac{1}{2}\big(\mathrm{e} - (\mathrm{e} - 1)\log(\mathrm{e} - 1)\big) = 0.89406\ldots \\ \beta &= \mathrm{e} - 1 = 1.71828\ldots \\ x_1 &= \log(\mathrm{e} - 1) = 0.54132\ldots \\ \delta &= \tfrac{1}{2}\big(2 - \mathrm{e} + (\mathrm{e} - 1)\log(\mathrm{e} - 1)\big) = 0.10593\ldots. \end{aligned}$$

Notice that we have a system of four *nonlinear* equations in four unknowns, due to the e^{x_1} term. Generally such nonlinear systems might not have a solution; in this case we can compute one.

Figure 2.4 shows the optimal approximation, along with the error $f(x) - p_*(x) = \mathrm{e}^x - (\alpha + \beta x)$. In particular, notice the size of the maximum error ($\delta = 0.10593\ldots$) and the point $x_1 = 0.54132\ldots$ at which this error is attained.

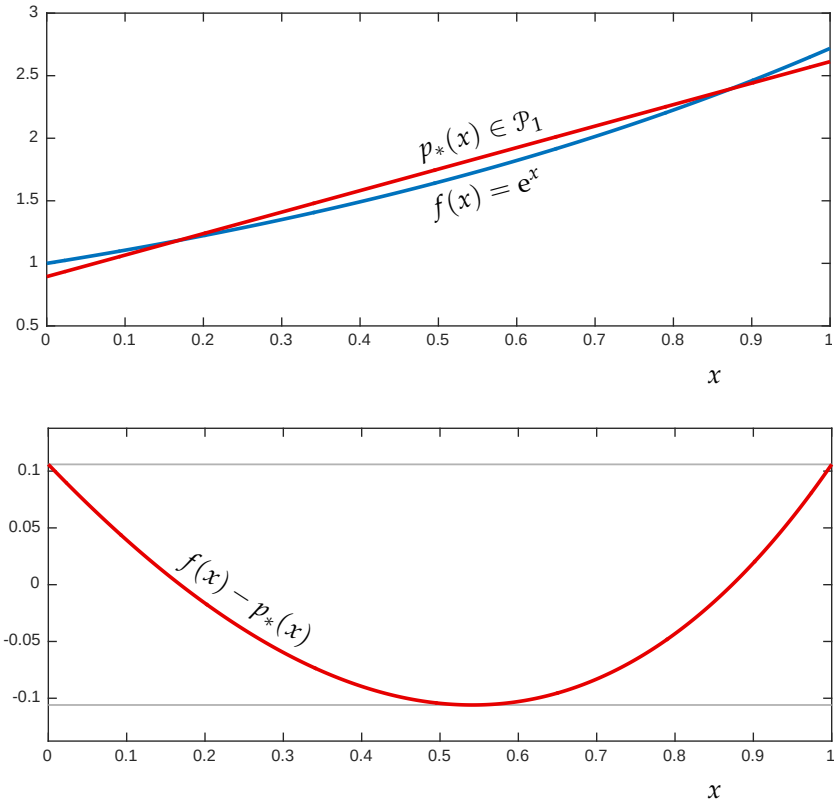


Figure 2.4: The top plot shows the minimax approximation p_* of degree $n = 1$ (red) to $f(x) = e^x$ (blue); the bottom plot shows the error $f(x) - p_*(x)$, equioscillating at $n + 1 = 3$ points.

LECTURE 15: *Chebyshev Polynomials for Optimal Interpolation*

2.3 *Optimal Interpolation Points via Chebyshev Polynomials*

As an application of the minimax approximation procedure, we consider how best to choose interpolation points $\{x_j\}_{j=0}^n$ to minimize

$$\|f - p_n\|_\infty,$$

where $p_n \in \mathcal{P}_n$ is the interpolant to f at the specified points.

Recall the interpolation error bound developed in Section 1.6: If $f \in C^{n+1}[a,b]$, then for any $x \in [a,b]$ there exists some $\xi \in [a,b]$ such that

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

Taking absolute values and maximizing over $[a,b]$ yields the bound

$$\|f - p_n\|_\infty = \max_{\xi\in[a,b]} \frac{|f^{(n+1)}(\xi)|}{(n+1)!} \max_{x\in[a,b]} \Big|\prod_{j=0}^{n}(x - x_j)\Big|.$$

For Runge's example, $f(x) = 1/(1+x^2)$ for $x \in [-5,5]$, we observed that $\|f - p_n\|_\infty \to \infty$ as $n \to \infty$ if the interpolation points $\{x_j\}$ are uniformly spaced over $[-5,5]$. However, Marcinkiewicz's theorem (Section 1.6) guarantees there is always some scheme for assigning the interpolation points such that $\|f - p_n\|_\infty \to 0$ as $n \to \infty$. While there is no fail-safe *a priori* system for picking interpolations points that will yield uniform convergence for *all* $f \in C[a,b]$, there is a distinguished choice that works exceptionally well for just about every function you will encounter in practice. We determine this set of interpolation points by choosing those $\{x_j\}_{j=0}^n$ that *minimize the error bound* (which is distinct from – but hopefully akin to – minimizing the error itself, $\|f - p_n\|_\infty$). That is, we want to solve

$$\min_{x_0,\dots,x_n} \max_{x\in[a,b]} \Big|\prod_{j=0}^{n}(x - x_j)\Big|. \tag{2.9}$$

Notice that

$$\begin{aligned}\prod_{j=0}^{n}(x - x_j) &= x^{n+1} - x^n \sum_{j=0}^{n} x_j + x^{n-1} \sum_{j=0}^{n}\sum_{k=0}^{n} x_j x_k - \cdots + (-1)^{n+1} \prod_{j=0}^{n} x_j \\ &= x^{n+1} - r(x),\end{aligned}$$

where $r \in \mathcal{P}_n$ is a degree-n polynomial depending on the interpolation nodes $\{x_j\}_{j=0}^n$.

For example, when $n = 1$,

$$(x - x_0)(x - x_1) = x^2 - \Big((x_0 + x_1)x - x_0 x_1\Big) = x^2 - r_1(x),$$

where $r_1(x) = (x_0 + x_1)x - x_0x_1$. By varying x_0 and x_1, we can obtain make r_1 any function in $\mathcal{P}_1$.

To find the optimal interpolation points according to (2.9), we should solve

$$\min_{r\in\mathcal{P}_n} \max_{x\in[a,b]} |x^{n+1} - r(x)| \;=\; \min_{r\in\mathcal{P}_n} \|x^{n+1} - r(x)\|_\infty.$$

Here the goal is to approximate an $(n+1)$-degree polynomial, x^{n+1}, with an n-degree polynomial. The method of solution is somewhat indirect: we will produce a class of polynomials of the form $x^{n+1} - r(x)$ that satisfy the requirements of the Oscillation Theorem, and thus $r(x)$ must be the minimax polynomial approximation to x^{n+1}. As we shall see, the roots of the resulting polynomial $x^{n+1} - r(x)$ will fall in the interval $[a, b]$, and can thus be regarded as 'optimal' interpolation points. For simplicity, we shall focus on the interval $[a, b] = [-1, 1]$.

Definition 2.1. The degree-n *Chebyshev polynomial* is defined for $x \in [-1, 1]$ by the formula

$$T_n(x) = \cos(n \cos^{-1} x).$$

At first glance, this formula may not appear to define a polynomial at all, since it involves trigonometric functions. But computing the first few examples, we find

$$n = 0: \quad T_0(x) = \cos(0 \cos^{-1} x) = \cos(0) = 1$$
$$n = 1: \quad T_1(x) = \cos(\cos^{-1} x) = x$$
$$n = 2: \quad T_2(x) = \cos(2 \cos^{-1} x) = 2\cos^2(\cos^{-1} x) - 1 = 2x^2 - 1.$$

For $n = 2$, we employed the identity $\cos 2\theta = 2\cos^2\theta - 1$, substituting $\theta = \cos^{-1} x$. More generally, use the cosine addition formula

$$\cos\alpha + \cos\beta = 2\cos\left(\frac{\alpha+\beta}{2}\right)\cos\left(\frac{\alpha-\beta}{2}\right)$$

to get the identity

$$\cos\big((n+1)\theta\big) = 2\cos\theta\cos n\theta - \cos\big((n-1)\theta\big).$$

This formula implies, for $n \geq 2$,

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x),$$

a formula related to the three term recurrence used to construct orthogonal polynomials.

Chebyshev polynomials exhibit a wealth of interesting properties, of which we mention just three.

Furthermore, it doesn't apply if $|x| > 1$. For such x one can define the Chebyshev polynomials using hyperbolic trigonometric functions, $T_n(x) = \cosh(n \cosh^{-1} x)$. Indeed, using hyperbolic trigonometric identities, one can show that this expression generates for $x \notin [-1, 1]$ the same polynomials we get for $x \in [-1, 1]$ from the standard trigonometric identities. We discuss this point in more detail at the end of the section.

In fact, Chebyshev polynomials are orthogonal polynomials on $[-1, 1]$ with respect to the inner product

$$\langle f, g\rangle = \int_a^b \frac{f(x)g(x)}{\sqrt{1-x^2}}\,\mathrm{d}x,$$

a fact we will use when studying Gaussian quadrature later in the semester.

Theorem 2.4. Let T_n be the degree-n Chebyshev polynomial

$$T_n(x) = \cos(n \cos^{-1} x)$$

for $x \in [-1, 1]$.

- $|T_n(x)| \leq 1$ for $x \in [-1, 1]$.
- The roots of T_n are the n points $\xi_j = \cos \frac{(2j-1)\pi}{2n}$, $j = 1, \ldots, n$.
- For $n \geq 1$, $|T_n(x)|$ is maximized on $[-1, 1]$ at the $n + 1$ points $\eta_j = \cos(j\pi/n)$, $j = 0, \ldots, n$:

$$T_n(\eta_j) = (-1)^j.$$

Proof. These results follow from direct calculations. For $x \in [-1, 1]$, $T_n(x) = \cos(n \cos^{-1}(x))$ cannot exceed one in magnitude because cosine cannot exceed one in magnitude. To verify the formula for the roots, compute

$$T_n(\xi_j) = \cos\left(n \cos^{-1} \cos\left(\frac{(2j-1)\pi}{2n}\right)\right) = \cos\left(\frac{(2j-1)\pi}{2}\right) = 0,$$

since cosine is zero at half-integer multiples of π. Similarly,

$$T_n(\eta_j) = \cos\left(n \cos^{-1} \cos\left(\frac{j\pi}{n}\right)\right) = \cos(j\pi) = (-1)^j.$$

Since $T_n(\eta_j)$ is a nonzero degree-n polynomial, it cannot attain more than $n + 1$ extrema on $[-1, 1]$, including the endpoint: we have thus characterized all the maxima of $|T_n|$ on $[-1, 1]$. ∎

Figure 2.5 shows Chebyshev polynomials T_n for nine different values of n.

2.3.1 *Interpolation at Chebyshev Points*

Finally, we are ready to solve the key minimax problem that will reveal optimal interpolation points. Looking at the above plots of Chebyshev polynomials, with their striking equioscillation properties, perhaps you have already guessed the solution yourself.

We defined the Chebyshev polynomials so that

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$$

with $T_0(x) = 1$ and $T_1(x) = x$. Thus T_{n+1} has the leading coefficient 2^n for $n \geq 0$. Define

$$\widehat{T}_{n+1} = 2^{-n} T_{n+1}$$

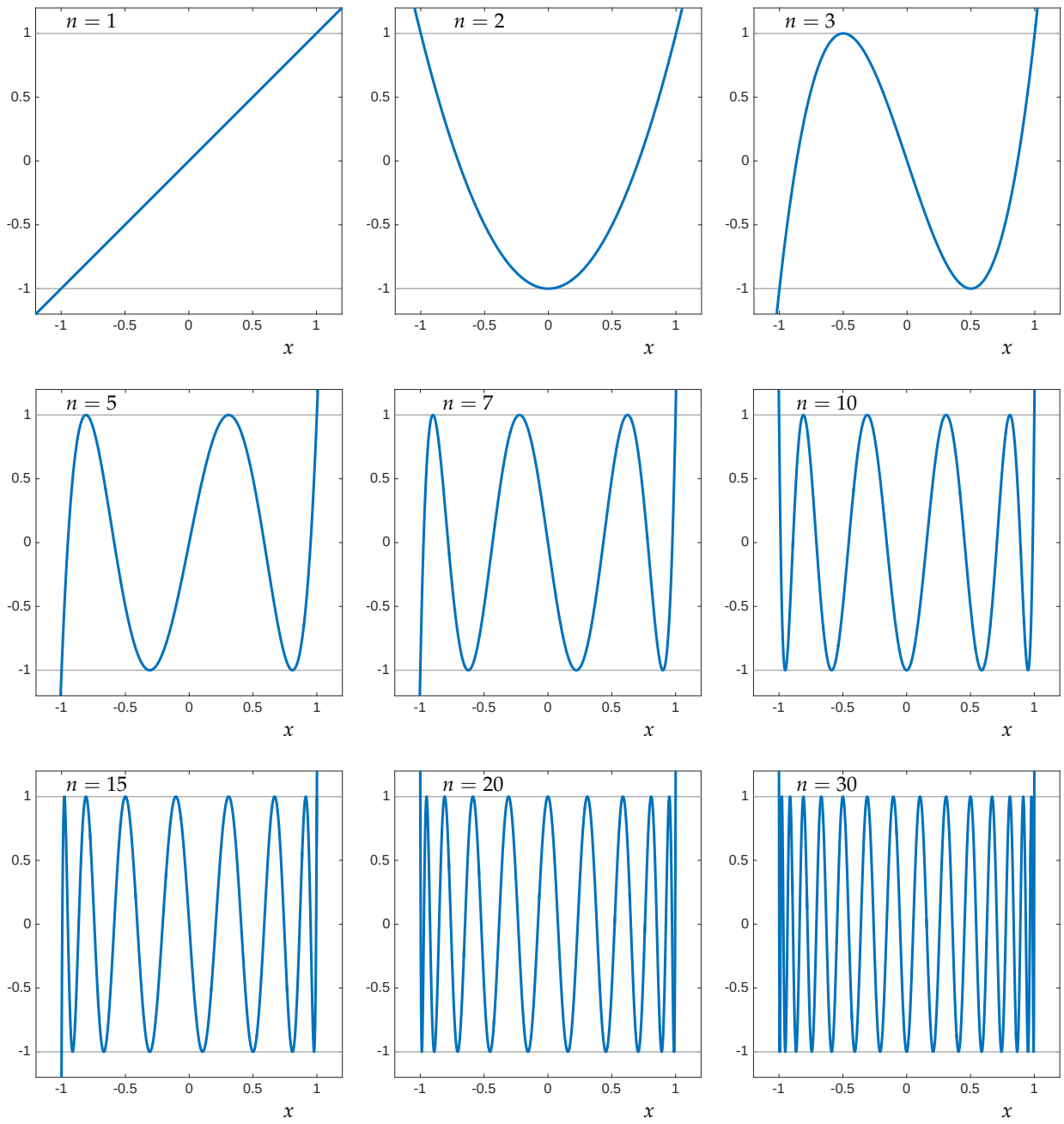


Figure 2.5: Chebyshev polynomials T_n of degree $n = 1, 2, 3$ (top), $n = 5, 7, 10$ (middle), and $n = 15, 20, 30$ (bottom). Note how rapidly these polynomials grow outside the interval $[-1, 1]$.

for $n \geq 0$, with $\widehat{T}_0(x) = 1$. These *normalized* Chebyshev polynomials are *monic*, i.e., the leading term in $\widehat{T}_{n+1}(x)$ is x^{n+1}, rather than $2^n x^{n+1}$ as for $T_{n+1}(x)$. Thus, we can write

$$\widehat{T}_{n+1}(x) = x^{n+1} - r_n(x)$$

for some polynomial $r_n(x) = x^{n+1} - \widehat{T}_{n+1}(x) \in \mathcal{P}_n$. We do not especially care about the particular coefficients of this r_n; our quarry will be the *roots* of $\widehat{T}_{n+1}$, the optimal interpolation points.

For $n \geq 0$, the polynomials $\widehat{T}_{n+1}(x)$ oscillate between $\pm 2^{-n}$ for $x \in [-1, 1]$, with the maximal values attained at

$$\eta_j = \cos\left(\frac{j\pi}{n+1}\right)$$

for $j = 0, \ldots, n+1$. In particular,

$$\widehat{T}_{n+1}(\eta_j) = (\eta_j)^{n+1} - r_n(\eta_j) = (-1)^j 2^{-n}.$$

Thus, we have found a polynomial $r_n \in \mathcal{P}_n$, together with $n+2$ distinct points, $\eta_j \in [-1, 1]$ where the maximum error

$$\max_{x \in [-1,1]} |x^{n+1} - r_n(x)| = 2^{-n}$$

is attained with alternating sign. Thus, by the oscillation theorem, we have found the minimax approximation to x^{n+1}.

Theorem 2.5 (Optimal approximation of x^{n+1})**.**

The optimal approximation to x^{n+1} from $\mathcal{P}_n$ for $x \in [-1, 1]$ is

$$r_n(x) = x^{n+1} - \widehat{T}_{n+1}(x) = x^{n+1} - 2^{-n} T_{n+1}(x) \in \mathcal{P}_n.$$

Thus, the optimal interpolation points are those $n+1$ roots of $x^{n+1} - r_n(x)$, that is, the roots of the degree-$(n+1)$ Chebyshev polynomial:

$$\xi_j = \cos\left(\frac{(2j+1)\pi}{2n+2}\right), \quad j = 0, \ldots, n.$$

For generic intervals $[a, b]$, a change of variable demonstrates that the same points, appropriately shifted and scaled, will be optimal.

Similar properties hold if interpolation is performed at the $n+1$ points

$$\eta_j = \cos\left(\frac{j\pi}{n}\right), \quad j = 0, \ldots, n,$$

which are also called Chebyshev points and are perhaps more popular due to their slightly simpler formula. (We used these points to successfully interpolate Runge's function, scaled to the interval $[-5, 5]$.) While these points differ from the roots of the Chebyshev polynomial, they *have the same distribution* as $n \to \infty$. That is the key.

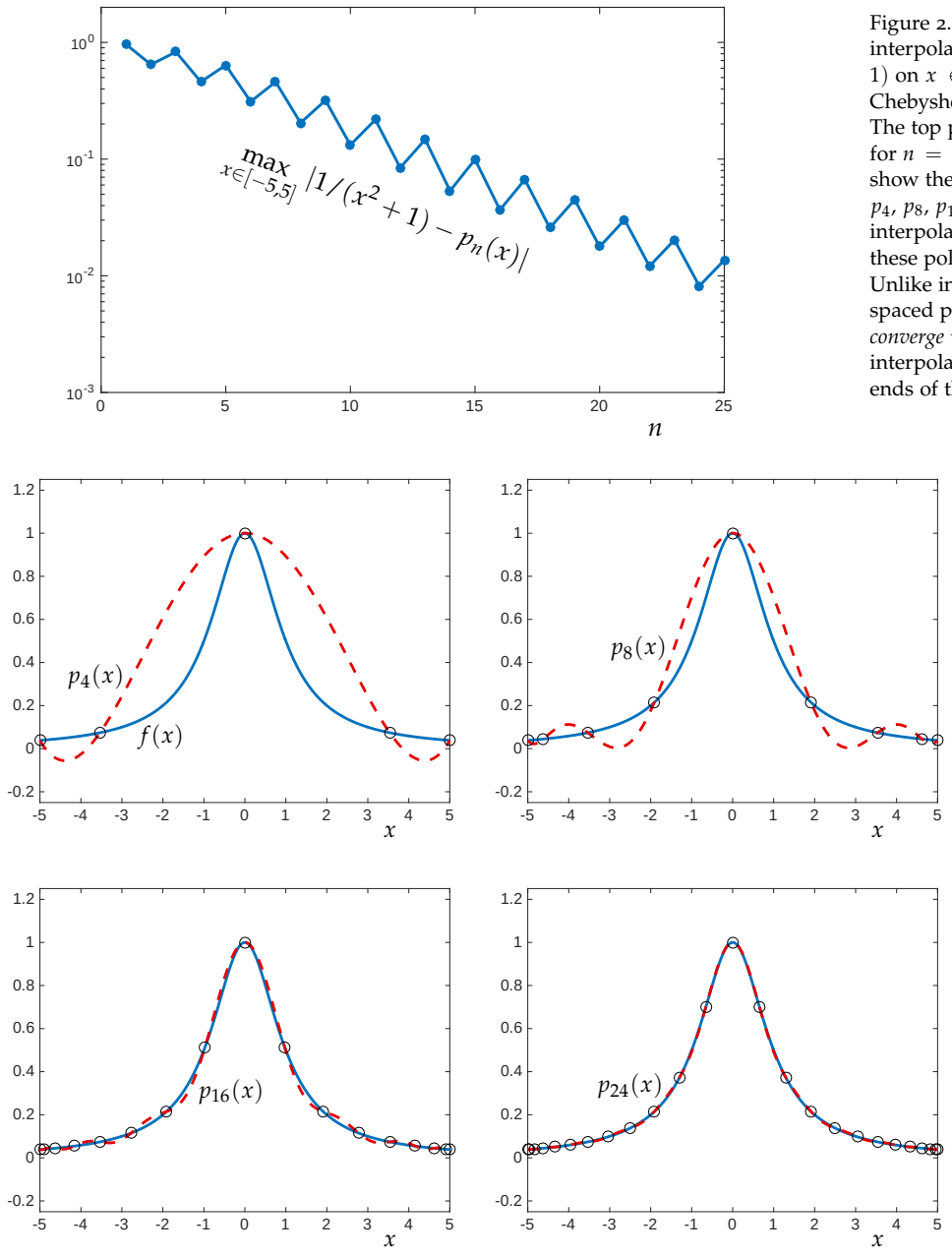


Figure 2.6: Repetition of Figure 1.8, interpolating Runge's function $1/(x^2+1)$ on $x \in [-5,5]$, but now using Chebyshev points $x_j = 5\cos(j\pi/n)$. The top plot shows this convergence for $n = 0,\ldots,25$; the bottom plots show the interpolating polynomials p_4, p_8, p_{16}, and p_{24}, along with the interpolation points that determine these polynomials (black circles). Unlike interpolation at uniformly spaced points, these interpolants *do converge* to f as $n \to \infty$. Notice how the interpolation points cluster toward the ends of the domain $[-5,5]$.

We emphasize the utility of interpolation at Chebyshev points by quoting the following result from Trefethen's excellent *Approximation Theory and Approximation Practice* (SIAM, 2013). Trefethen emphasizes that worst-case results like Faber's theorem (Theorem 1.4) give misleadingly pessimistic concerns about interpolation. If the function $f \in C[a,b]$ has just a bit of smoothness (i.e., bounded derivatives), interpolation in Chebyshev points is 'bulletproof'. The following theorem consolidates aspects of Theorem 7.2 and 8.2 in Trefethen's book.

The results are stated for $[a,b] = [-1,1]$ but can be adapted to any real interval.

Theorem 2.6 (Convergence of Interpolants at Chebyshev Points)**.**
For any $n > 0$, let p_n denote the interpolant to $f \in C[-1,1]$ at the Chebyshev points

$$x_j = \cos\left(\frac{j\pi}{n}\right), \quad j = 0, \ldots, n.$$

- Suppose $f \in C^\nu[-1,1]$ for some $\nu \geq 1$, with $f^{(\nu)}$ having variation $V(\nu)$, i.e.,

$$V(\nu) := \max_{x\in[-1,1]} f^{(\nu)}(x) - \min_{x\in[-1,1]} f^{(\nu)}(x).$$

  Then for any $n > \nu$,

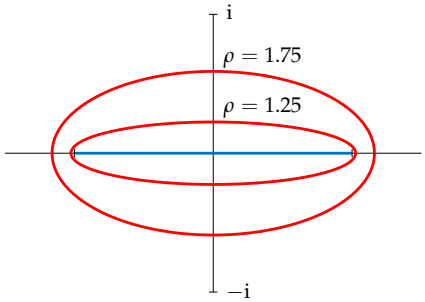
$$\|f - p_n\|_\infty \leq \frac{4V(\nu)}{\pi(\nu(n-\nu)^\nu)}.$$

- Suppose f is *analytic* on $[-1,1]$ and can be analytically continued (into the complex plane) onto the region bounded by the ellipse

$$E_\rho := \left\{\frac{\rho e^{i\theta} + e^{-i\theta}/\rho}{2} : \theta \in [0, 2\pi)\right\}.$$

  Suppose further that $|f(z)| \leq M$ on and inside E_ρ. Then

$$\|f - p_n\|_\infty \leq \frac{2M\rho^{-n}}{\rho - 1}.$$



Interval $[-1,1]$ (blue), with two ellipses E_ρ for $\rho = 1.25$ and $\rho = 1.75$.

For example, the first part of this theorem implies that if f' exists and is bounded, then $\|f - p_n\|_\infty$ must converge at least as fast as $1/n$ as $n \to \infty$. While that is not such a fast rate, it does indeed show convergence of the interpolant. The second part of the theorem ensures that if f is well behaved in the region of the complex plane around $[-1,1]$, the convergence will be extremely fast: the larger the area of $\mathbb{C}$ in which f is well behaved, the faster the convergence.

2.3.2 *Chebyshev polynomials beyond* $[-1,1]$

Another way of interpreting the equioscillating property of Chebyshev polynomials is that T_n solves the approximation problem

$$\|T_n\|_\infty = \min_{\substack{p\in\mathcal{P}_n \\ p \text{ monic}}} \|p\|_\infty,$$

over the interval $[-1,1]$, where a polynomial is *monic* if it has the form $x^n + q(x)$ for $q \in \mathcal{P}_{n-1}$.

In some applications, such as the analysis of iterative methods for solving large-scale systems of linear equations, one needs to bound the size of the Chebyshev polynomial *outside the interval* $[-1,1]$. Figure 2.5 shows that T_n grows very quickly outside $[-1,1]$, even for modest values of n. How fast?

To describe Chebyshev polynomials outside $[-1,1]$, we must replace the trigonometric functions in the definition $T_n(x) = \cos(n\cos^{-1}x)$ with hyperbolic trigonometric functions:

$$T_n(x) = \cosh(n\cosh^{-1}x), \qquad x \notin (-1,1). \tag{2.10}$$

Is this definition is consistent with

$$T_n(x) = \cos(n\cos^{-1}x), \qquad x \in [-1,1]$$

used previously? Trivially one can see that the new definition also gives $T_0(x) = 1$ and $T_1(x) = x$. Like standard trigonometric functions, the hyperbolic functions also satisfy the addition formula

$$\cosh\alpha + \cosh\beta = 2\cosh\left(\frac{\alpha+\beta}{2}\right)\cosh\left(\frac{\alpha-\beta}{2}\right),$$

and so

$$\cosh\big((n+1)\theta\big) = 2\cosh\theta\cosh n\theta - \cosh\big((n-1)\theta\big),$$

leading to the same three-term recurrence as before:

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x).$$

Thus, the definitions are consistent.

We would like a more concrete formula for $T_n(x)$ for $x \notin [-1,1]$ than we could obtain from the formula (**??**). Thankfully Chebyshev polynomials have infinitely many interesting properties to lean on. Consider the change of variables

$$x = \frac{w + w^{-1}}{2},$$

which allows us to write

$$x = \frac{e^{\log w} + e^{-\log w}}{2} = \cosh(\log w).$$

Thus work from the definition to obtain

$$\begin{aligned} T_n(x) &= \cosh(n\cosh^{-1}(x)) \\ &= \cosh(n\log w) \\ &= \cosh(\log w^n) = \frac{e^{\log(w^n)} + e^{-\log(w^n)}}{2} = \frac{w^n + w^{-n}}{2}. \end{aligned}$$

We emphasize this last important formula:

$$T_n(x) = \frac{w^n + w^{-n}}{2}, \qquad x = \frac{w + w^{-1}}{2} \notin (-1,1). \tag{2.11}$$

We have thus shown that $|T_n(x)|$ *will grow exponentially in* n *for any* $x \notin (-1,1)$ *for which* $|w| \neq 1$. When does $|w| = 1$? Only when $x = \pm 1$. Hence,

$|T_n(x)|$ grows exponentially in n for all $x \notin [-1,1]$.

Example 2.3. We want to evaluate $T_n(2)$ as a function of n. First, find w such that $2 = (w + w^{-1})/2$, i.e.,

$$w^2 - 4w + 1 = 0.$$

Solve this quadratic for

$$w_\pm = 2 \pm \sqrt{3}.$$

We take $w = 2 + \sqrt{3} = 3.7320\ldots$. Thus by (2.11)

$$T_n(2) = \frac{(2+\sqrt{3})^n + (2-\sqrt{3})^n}{2} \approx \frac{(2+\sqrt{3})^n}{2}$$

as $n \to \infty$, since $(2-\sqrt{3})^n = (0.2679\ldots)^n \to 0$.

Which $\pm$ choice should you make? It does not matter. Notice that $(2-\sqrt{3})^{-1} = 2+\sqrt{3}$, and this happens in general: $w_\pm = 1/w_\mp$.

Take a moment to reflect on this: We have a beautifully concrete way to write down $|T_n(x)|$ that does not involve any hyperbolic trigonometric formulas, or require use of the Chebyshev recurrence relation. Formulas of this type can be very helpful for analysis in various settings. You will see one such example on Problem Set 3.

LECTURE 16: *Introduction to Least Squares Approximation*

2.4 Least squares approximation

The minimax criterion is an intuitive objective for approximating a function. However, in many cases it is more appealing (for both computation and for the given application) to find an approximation to f that *minimizes the integral of the square of the error*.

Given $f \in C[a,b]$, find $P_* \in \mathcal{P}_n$ such that

$$\Big(\int_a^b (f(x)-P_*(x))^2\,\mathrm{d}x\Big)^{1/2} = \min_{p\in\mathcal{P}_n}\Big(\int_a^b (f(x)-p(x))^2\,\mathrm{d}x\Big)^{1/2}. \tag{2.12}$$

This is an example of a *least squares problem*.

2.4.1 Inner products for function spaces

To facilitate the development of least squares approximation theory, we introduce a formal structure for $C[a,b]$. First, recognize that $C[a,b]$ is a *linear space*: any linear combination of continuous functions on $[a,b]$ must itself be continuous on $[a,b]$.

Definition 2.2. The *inner product* of the functions $f, g \in C[a,b]$ is

$$\langle f, g\rangle = \int_a^b f(x)g(x)\,\mathrm{d}x.$$

The inner product satisfies the following basic axioms:

- $\langle \alpha f + g, h\rangle = \alpha\langle f, h\rangle + \langle g, h\rangle$ for all $f, g, h \in C[a,b]$ and all $\alpha \in \mathbb{R}$;
- $\langle f, g\rangle = \langle g, f\rangle$ for all $f, g \in C[a,b]$;
- $\langle f, f\rangle \geq 0$ for all $f \in C[a,b]$.

With this inner product we associate the norm

$$\|f\|_2 := \langle f, f\rangle^{1/2} = \Big(\int_a^b f(x)^2\,\mathrm{d}x\Big)^{1/2}.$$

This is often called the 'L^2 norm,' where the superscript '2' in L^2 refers to the fact that the integrand involves the *square* of the function f; the L stands for *Lebesgue*, coming from the fact that this inner product can be generalized from $C[a,b]$ to the set of all functions that are *square-integrable*, in the sense of Lebesgue integration. By restricting our attention to continuous functions, we dodge the measure-theoretic complexities.

For simplicity we are assuming that f and g are real-valued. To handle complex-valued functions, one generalizes the inner product to

$$\langle f, g\rangle = \int_a^b f(x)\overline{g(x)}\,\mathrm{d}x,$$

which then gives $\langle f, g\rangle = \overline{\langle g, f\rangle}$.

The Lebesgue theory gives a more robust definition of the integral than the conventional Riemann approach. With such notions one can extend least squares approximation beyond $C[a,b]$, to more exotic function spaces.

2.4.2 *Least squares minimization via calculus*

We are now ready to solve the least squares problem. We shall call the optimal polynomial $P_* \in \mathcal{P}_n$, i.e.,

$$\|f - P_*\|_2 = \min_{p \in \mathcal{P}_n} \|f - p\|_2.$$

We can solve this minimization problem using basic calculus. Consider this example for $n = 1$, where we optimize the error over polynomials of the form $p(x) = c_0 + c_1 x$. The polynomial that minimizes $\|f - p\|_2$ will also minimize its square, $\|f - p\|_2^2$. For any given $p \in \mathcal{P}_1$, define the error function

$$\begin{aligned} E(c_0, c_1) := \|f(x) - (c_0 + c_1 x)\|_{L^2}^2 &= \int_a^b (f(x) - c_0 - c_1 x)^2 \, \mathrm{d}x \\ &= \int_a^b \Big(f(x)^2 - 2f(x)(c_0 + c_1 x) + (c_0^2 + 2c_0 c_1 x + c_1^2 x^2) \Big) \, \mathrm{d}x \\ &= \int_a^b f(x)^2 \, \mathrm{d}x - 2c_0 \int_a^b f(x) \, \mathrm{d}x - 2c_1 \int_a^b x f(x) \, \mathrm{d}x \\ &\qquad + c_0^2 (b - a) + c_0 c_1 (b^2 - a^2) + \tfrac{1}{3} c_1^2 (b^3 - a^3). \end{aligned}$$

To find the optimal polynomial, P_*, optimize E over c_0 and c_1, i.e., find the values of c_0 and c_1 for which

$$\frac{\partial E}{\partial c_0} = \frac{\partial E}{\partial c_1} = 0.$$

First, compute

$$\begin{aligned} \frac{\partial E}{\partial c_0} &= -2 \int_a^b f(x) \, \mathrm{d}x + 2c_0 (b - a) + c_1 (b^2 - a^2) \\ \frac{\partial E}{\partial c_1} &= -2 \int_a^b x f(x) \, \mathrm{d}x + c_0 (b^2 - a^2) + c_1 \tfrac{2}{3} (b^3 - a^3). \end{aligned}$$

Setting these partial derivatives equal to zero yields

$$\begin{aligned} 2c_0 (b - a) + c_1 (b^2 - a^2) &= 2 \int_a^b f(x) \, \mathrm{d}x \\ c_0 (b^2 - a^2) + c_1 \tfrac{2}{3} (b^3 - a^3) &= 2 \int_a^b x f(x) \, \mathrm{d}x. \end{aligned}$$

These equations, linear in the unknowns c_0 and c_1, can be written in the matrix form

$$\begin{bmatrix} 2(b - a) & b^2 - a^2 \\ b^2 - a^2 & \frac{2}{3}(b^3 - a^3) \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} = \begin{bmatrix} 2 \int_a^b f(x) \, \mathrm{d}x \\ 2 \int_a^b x f(x) \, \mathrm{d}x \end{bmatrix}.$$

When $b \neq a$ this system always has a unique solution. The resulting c_0 and c_1 are the coefficients for the monomial-basis expansion of the least squares approximation $P_* \in \mathcal{P}_1$ to f on $[a, b]$.

Example 2.4 ($f(x) = e^x$)**.** Apply this result to $f(x) = e^x$ for $x \in [0,1]$. Since

$$\int_0^1 e^x \, dx = e - 1, \qquad \int_0^1 x e^x \, dx = \left[e^x (x-1) \right]_{x=0}^1 = 1,$$

we must solve the system

$$\begin{bmatrix} 2 & 1 \\ 1 & \frac{2}{3} \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} = \begin{bmatrix} 2e - 2 \\ 2 \end{bmatrix}.$$

The desired solution is

$$c_0 = 4e - 10, \qquad c_1 = 18 - 6e.$$

Figure 2.7 compares f to this least squares approximation P_* and the minimax approximation p_* computed earlier.

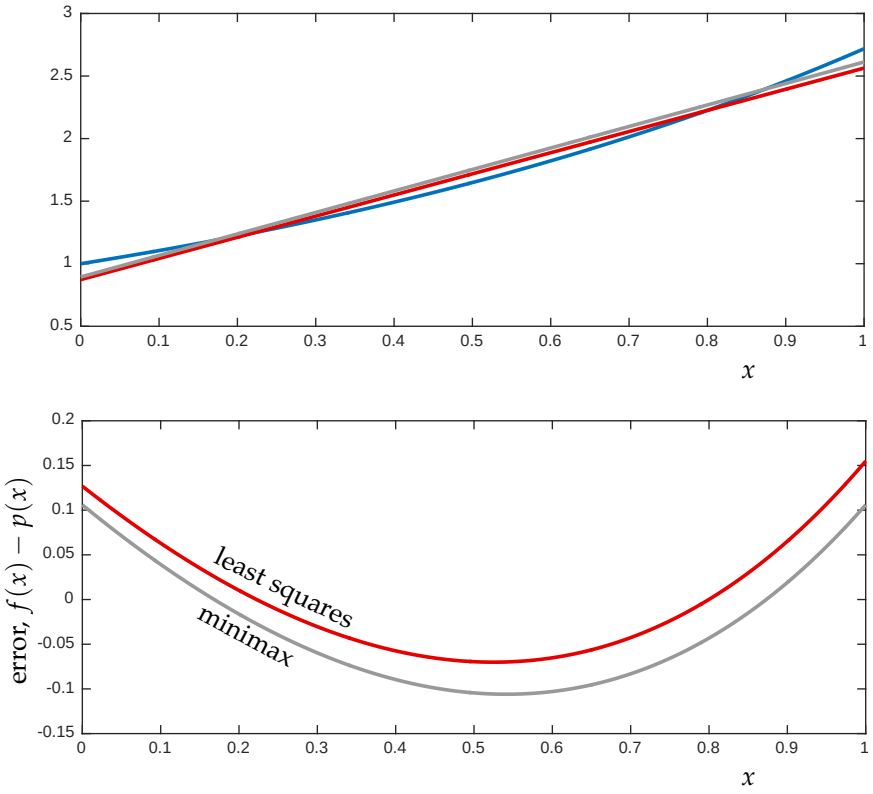


Figure 2.7: Top: Approximation of $f(x) = e^x$ (blue) over $x \in [0,1]$ via least squares (P_*, shown in red) and minimax (p_*, shown as a gray line).

Bottom: Error curves for least squares, $f - P_*$ (red), and minimax, $f - p_*$ (gray) approximation. While the curves have similar shape, note that the red curve does not attain its maximum deviation from f at $n + 2 = 3$ points, while the gray one does.

We can see from the plots in Figure 2.7 that the approximation looks decent to the eye, but the error is not terribly small. We can decrease that error by increasing the degree of the approximating polynomial. Just as we used a 2-by-2 linear system to find the best linear approximation, a general $(n+1)$-by-$(n+1)$ linear system can be constructed to yield the degree-n least squares approximation.

In fact, $\|f - P_*\|_2 = 0.06277\ldots$. This is indeed smaller than the 2-norm error of the minimax approximation p_*: $\|f - p_*\|_2 = 0.07228\ldots$.

2.4.3 General polynomial bases

Note that we performed the above minimization in the monomial basis: $p(x) = c_0 + c_1 x$ is a linear combination of 1 and x. Our experience with interpolation suggests that different choices for the basis may yield approximation algorithms with superior numerical properties. Thus, we develop the form of the approximating polynomial in an arbitrary basis.

Suppose $\{\phi_k\}_{k=0}^n$ is a basis for $\mathcal{P}_n$. Any $p \in \mathcal{P}_n$ can be written as

$$p(x) = \sum_{k=0}^{n} c_k \phi_k(x).$$

The error expression takes the form

$$E(c_0, \ldots, c_n) := \|f(x) - p(x)\|_{L^2}^2 = \int_a^b \Big(f(x) - \sum_{k=0}^{n} c_k \phi_k(x)\Big)^2 \, \mathrm{d}x$$

$$= \langle f, f\rangle - 2\sum_{k=0}^{n} c_k \langle f, \phi_k\rangle + \sum_{k=0}^{n}\sum_{\ell=0}^{n} c_k c_\ell \langle \phi_k, \phi_\ell\rangle.$$

To minimize E, we seek critical values of $\mathbf{c} = [c_0, \ldots, c_{n+1}]^T \in \mathbb{R}^{n+1}$, i.e., we want coefficients where the gradient of E with respect to $\mathbf{c}$ is zero: $\nabla_{\mathbf{c}} E = \mathbf{0}$. To compute this gradient, evaluate $\partial E/\partial c_j$ for $j = 0, \ldots, n$:

$$\frac{\partial E}{\partial c_j} = \frac{\partial}{\partial c_j}\langle f, f\rangle - \frac{\partial}{\partial c_j}\Big(2\sum_{k=0}^{n} c_k \langle f, \phi_k\rangle\Big) + \frac{\partial}{\partial c_j}\Big(\sum_{k=0}^{n}\sum_{\ell=0}^{n} c_k c_\ell \langle \phi_k, \phi_\ell\rangle\Big)$$

$$= 0 - 2\langle f, \phi_j\rangle + \frac{\partial}{\partial c_j}\Big(c_j^2\langle \phi_j, \phi_j\rangle + \sum_{\substack{k=0\\k\neq j}}^{n} c_k c_j \langle \phi_k, \phi_j\rangle + \sum_{\substack{\ell=0\\\ell\neq j}}^{n} c_j c_\ell \langle \phi_j, \phi_\ell\rangle + \sum_{\substack{k=0\\k\neq j}}^{n}\sum_{\substack{\ell=0\\\ell\neq j}}^{n} c_k c_\ell \langle \phi_k, \phi_\ell\rangle\Big)$$

In this last line, we have broken the double sum on the previous line into four parts: one that contains c_j^2, two that contain c_j ($c_k c_j$ for $k \neq j$; $c_j c_\ell$ for $\ell \neq j$), and one (the double sum) that does not involve c_j at all. This decomposition makes it easier to compute the derivative:

$$\frac{\partial}{\partial c_j}\Big(c_j^2\langle \phi_j, \phi_j\rangle \;+ \sum_{\substack{k=0\\k\neq j}}^{n} c_k c_j \langle \phi_k, \phi_j\rangle + \sum_{\substack{\ell=0\\\ell\neq j}}^{n} c_j c_\ell \langle \phi_j, \phi_\ell\rangle + \sum_{\substack{k=0\\k\neq j}}^{n}\sum_{\substack{\ell=0\\\ell\neq j}}^{n} c_k c_\ell \langle \phi_k, \phi_\ell\rangle\Big)$$

$$= 2c_j\langle \phi_j, \phi_j\rangle + \sum_{\substack{k=0\\k\neq j}}^{n} c_k \langle \phi_k, \phi_j\rangle + \sum_{\substack{\ell=0\\\ell\neq j}}^{n} c_\ell \langle \phi_j, \phi_\ell\rangle + 0$$

$$= 2c_j\langle \phi_j, \phi_j\rangle + 2\sum_{\substack{k=0\\k\neq j}}^{n} c_k \langle \phi_k, \phi_j\rangle.$$

These terms contribute to $\partial E/\partial c_j$ to give

$$\frac{\partial E}{\partial c_j} = -2\langle f, \phi_j \rangle + 2 \sum_{k=0}^{n} c_k \langle \phi_k, \phi_j \rangle. \tag{2.13}$$

To minimize E, set $\partial E/\partial c_j = 0$ for $j = 0, \ldots, n$, which gives the $n+1$ equations

$$\sum_{k=0}^{n} c_k \langle \phi_k, \phi_j \rangle = \langle f, \phi_j \rangle, \quad j = 0, \ldots, n, \tag{2.14}$$

in the $n+1$ unknowns $c_0, \ldots, c_n$. Since these equations are linear in the unknowns, write them in matrix form:

$$\begin{bmatrix} \langle \phi_0, \phi_0 \rangle & \langle \phi_0, \phi_1 \rangle & \cdots & \langle \phi_0, \phi_n \rangle \\ \langle \phi_1, \phi_0 \rangle & \langle \phi_1, \phi_1 \rangle & \cdots & \langle \phi_1, \phi_n \rangle \\ \vdots & \vdots & \ddots & \vdots \\ \langle \phi_n, \phi_0 \rangle & \langle \phi_n, \phi_1 \rangle & \cdots & \langle \phi_n, \phi_n \rangle \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} \langle f, \phi_0 \rangle \\ \langle f, \phi_1 \rangle \\ \vdots \\ \langle f, \phi_n \rangle \end{bmatrix}, \tag{2.15}$$

which we denote $\mathbf{G}\mathbf{c} = \mathbf{b}$. The matrix $\mathbf{G}$ is called the *Gram matrix*. Using this matrix-vector notation, we can accumulate the partial derivatives formulas (2.13) for E into the gradient

$$\nabla_{\mathbf{c}} E = 2\big(\mathbf{G}\mathbf{c} - \mathbf{b}\big).$$

Since $\mathbf{c}$ is a critical point if and only if $\nabla_{\mathbf{c}} E(\mathbf{c}) = \mathbf{0}$, we must ask:

- How many critical points are there? Equivalently, how many $\mathbf{c}$ solve $\mathbf{G}\mathbf{c} = \mathbf{b}$?
- If $\mathbf{c}$ is a critical point, is it a (local or even global) minimum?

We will answer the first question by showing that $\mathbf{G}$ is invertible, and hence E has a unique critical point. To answer the second question, we must inspect the Hessian

$$\nabla_{\mathbf{c}}^2 E = \nabla_{\mathbf{c}}(\nabla_{\mathbf{c}} E) = 2\mathbf{G}.$$

The critical point $\mathbf{c}$ is local minimum if and only if the Hessian is *symmetric positive definite*.

A matrix $\mathbf{G}$ is *positive definite* provided $\mathbf{z}^*\mathbf{G}\mathbf{z} > 0$ for all $\mathbf{z} \neq \mathbf{0}$.

The symmetry of the inner product implies $\langle \phi_j, \phi_k \rangle = \langle \phi_k, \phi_j \rangle$, and hence $\mathbf{G}$ is symmetric. (In this case, symmetry also follows from the equivalence of mixed partial deritivates.) The following theorem confirms that $\mathbf{G}$ is indeed positive definite.

LECTURE 17: *Fundamentals of Least Squares Approximation, Part I*
LECTURE 18: *Fundamentals of Least Squares Approximation, Part II*

Theorem 2.7. If $\phi_0, \ldots, \phi_n$ are linearly independent, the Gram matrix $\mathbf{G}$ is positive definite.

This proof is very general: we are thinking of $\phi_0, \ldots, \phi_n$ being a basis for $\mathcal{P}_n$ (and hence linearly independent), but the same proof applies to any linearly independent set of vectors in a general inner product space.

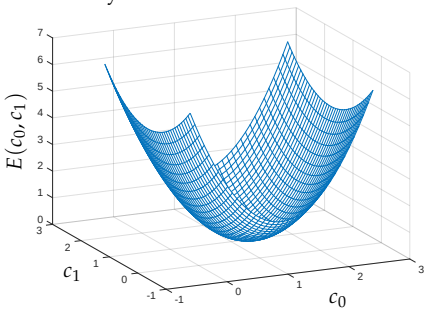
Proof. For a generic $\mathbf{z} \in \mathbb{R}^{n+1}$, consider the product

$$\mathbf{z}^*\mathbf{G}\mathbf{z} = \begin{bmatrix} z_0 & z_1 & \cdots & z_n \end{bmatrix} \begin{bmatrix} \langle \phi_0, \phi_0 \rangle & \langle \phi_0, \phi_1 \rangle & \cdots & \langle \phi_0, \phi_n \rangle \\ \langle \phi_1, \phi_0 \rangle & \langle \phi_1, \phi_1 \rangle & \cdots & \langle \phi_1, \phi_n \rangle \\ \vdots & & \ddots & \vdots \\ \langle \phi_n, \phi_0 \rangle & \langle \phi_n, \phi_1 \rangle & \cdots & \langle \phi_n, \phi_n \rangle \end{bmatrix} \begin{bmatrix} z_0 \\ z_1 \\ \vdots \\ z_n \end{bmatrix}$$

$$= \begin{bmatrix} z_0 & z_1 & \cdots & z_n \end{bmatrix} \begin{bmatrix} \sum_{k=0}^n z_j \langle \phi_0, \phi_k \rangle \\ \sum_{k=0}^n z_j \langle \phi_1, \phi_k \rangle \\ \vdots \\ \sum_{k=0}^n z_j \langle \phi_n, \phi_k \rangle \end{bmatrix} = \sum_{j=0}^n \sum_{k=0}^n z_j z_k \langle \phi_j, \phi_k \rangle.$$

Eigenvalues illuminate. The surfaces below visualize $E(c_0, c_1)$ for best approximation of $f(x) = e^x$ from $\mathcal{P}_1$ over $x \in [-1, 1]$ (top) and $x \in [0, 1]$.

For $[-1, 1]$, the eigenvalues of $\mathbf{G}$ are relatively large, and the error surface looks very bowl-like.



Eigenvalues of $\mathbf{G}$: $\lambda_1 = 2$, $\lambda_2 = 2/3$

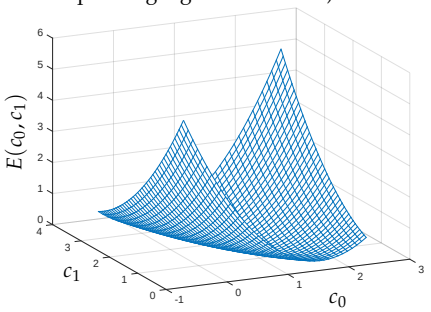
Now use linearity of the inner product to write

$$\mathbf{z}^*\mathbf{G}\mathbf{z} = \sum_{j=0}^n \sum_{k=0}^n z_j z_k \langle \phi_j, \phi_k \rangle = \left\langle \sum_{j=0}^n z_j \phi_j, \sum_{k=0}^n z_k \phi_k \right\rangle = \left\| \sum_{j=0}^n z_j \phi_j \right\|^2.$$

Thus, by nonnegativity of the norm, $\mathbf{z}^*\mathbf{G}\mathbf{z} \geq 0$. This is enough to show that $\mathbf{G}$ is *positive semidefinite*. To show that $\mathbf{G}$ is *positive definite*, we must show that $\mathbf{z}^*\mathbf{G}\mathbf{z} > 0$ if $\mathbf{z} \neq \mathbf{0}$. Now since $\phi_0, \ldots, \phi_n$ are linearly independent, $\sum_{j=0}^n c_j \phi_j = 0$ if and only if $c_0 = \cdots = c_n = 0$, i.e., if and only if $\mathbf{z} = \mathbf{0}$. Thus, if $\mathbf{z} \neq \mathbf{0}$, $\mathbf{z}^*\mathbf{G}\mathbf{z} > 0$. ∎

This answers the second question posed above, and also makes the answer to the first trivial.

For $[0, 1]$, $\mathbf{G}$ has a small eigenvalue: the error surface is much more 'shallow' in one direction. (The orientation of the trough can be found from the corresponding eigenvector of $\mathbf{G}$.)



Eigenvalues of $\mathbf{G}$: $\lambda_1 = (4 + \sqrt{13})/6 = 1.26759\ldots$
$\lambda_2 = (4 - \sqrt{13})/6 = 0.06574\ldots$

Corollary 2.1. If $\phi_0, \ldots, \phi_n$ are linearly independent, the Gram matrix $\mathbf{G}$ is invertible.

Proof. The matrix $\mathbf{G}$ is invertible if $\mathbf{G}\mathbf{z} = \mathbf{0}$ implies $\mathbf{z} = \mathbf{0}$, i.e., $\mathbf{G}$ has a trivial null space. If $\mathbf{G}\mathbf{z} = \mathbf{0}$, then $\mathbf{z}^*\mathbf{G}\mathbf{z} = 0$. Theorem 2.7 ensures that $\mathbf{G}$ is positive definite, so $\mathbf{z}^*\mathbf{G}\mathbf{z} = 0$ implies $\mathbf{z} = \mathbf{0}$. Hence, $\mathbf{G}$ has a trivial null space, and is thus invertible. ∎

We can summarize our findings as follows.

Given any basis $\phi_0, \ldots, \phi_n$, the least squares approximation P_* to $f \in C[a,b]$ is unique and can be expressed as

$$P_* = \sum_{j=0}^{n} c_j \phi_j,$$

where the coefficients $\mathbf{c}$ are computed as the unique solution of $\mathbf{Gc} = \mathbf{b}$.

As with the interpolation problem studied earlier, different choices of basis will give different linear algebra problems, but ultimately result in the same overall approximation P_*. We shall study several choices for the basis in Sections 2.4.5 and 2.4.6. Before doing so, we establish a fundamental property of least squares approximation.

2.4.4 *Orthogonality of the error*

This is the fundamental theorem of linear least squares problems:

The error $f - P_*$ is orthogonal to the approximating subspace $\mathcal{P}_n$.

Having worked hard to characterize the optimal approximation, the formula $\mathbf{Gc} = \mathbf{b}$ makes the proof of this result trivial.

Theorem 2.8. The function $P_* \in \mathcal{P}_n$ is the least squares approximation to $f \in C[a,b]$ if and only if the error $f - P_*$ is orthogonal to the subspace $\mathcal{P}_n$ from which the approximation was drawn:

$$\langle f - P_*, q \rangle = 0, \qquad \text{for all } q \in \mathcal{P}_n.$$

Proof. First suppose that P_* is the least squares approximation. Thus given any basis $\phi_0, \ldots, \phi_n$ for $\mathcal{P}_n$, we can express $P_* = c_0\phi_0 + \cdots + c_n\phi_n$, where the coefficients solve $\mathbf{Gc} = \mathbf{b}$. Now for any basis function ϕ_j, use the linearity of the inner product to compute

$$\begin{aligned}\langle f - P_*, \phi_j \rangle &= \langle f, \phi_j \rangle - \langle P_*, \phi_j \rangle \\ &= \langle f, \phi_j \rangle - \Big\langle \sum_{k=0}^{n} c_k \phi_k, \phi_j \Big\rangle = \langle f, \phi_j \rangle - \sum_{k=0}^{n} c_k \langle \phi_k, \phi_j \rangle.\end{aligned}$$

Recall that the jth row of the equation $\mathbf{Gc} = \mathbf{b}$ (see (2.14) is precisely

$$\sum_{k=0}^{n} c_k \langle \phi_k, \phi_j \rangle = \langle f, \phi_j \rangle,$$

so since the least squares approximation must satisfy $\mathbf{Gc} = \mathbf{b}$, conclude that $\langle f - P_*, \phi_j \rangle = 0$. Since this orthogonality holds for

$j = 0, \ldots, n$, the error $f - P_*$ is orthogonal to the entire basis for $\mathcal{P}_n$, and hence it is orthogonal to any vector $q = \sum_{j=0}^{n} d_j \phi_j \in \mathcal{P}_n$, since

$$\langle f - P_*, q \rangle = \Big\langle f - P_*, \sum_{j=0}^{n} d_j \phi_j \Big\rangle = \sum_{k=0}^{n} d_j \langle f - P_*, \phi_j \rangle = \sum_{k=0}^{n} d_j \cdot 0 = 0.$$

Thus, the least squares error $f - P_*$ is orthogonal to all $q \in \mathcal{P}_n$.

On the other hand suppose that $p \in \mathcal{P}_n$ gives an error $f - p$ that is orthogonal to all $q \in \mathcal{P}_n$, i.e.,

$$\langle f - p, q \rangle = 0, \qquad \text{for all } q \in \mathcal{P}_n. \tag{2.16}$$

Let $\phi_0, \ldots, \phi_n$ be a basis for $\mathcal{P}_n$. Then we can find $c_0, \ldots, c_n$ so that $p = c_0\phi_0 + \cdots + c_n\phi_n$. The orthogonality of $f - p$ to $\mathcal{P}_n$ in (**??**) implies in particular that $\langle f - p, \phi_j \rangle = 0$ for all $j = 0, \ldots, n$, i.e., using linearity of the inner product,

$$0 = \Big\langle f - \sum_{k=0}^{n} c_k \phi_k, \phi_j \Big\rangle = \langle f, \phi_j \rangle - \sum_{k=0}^{n} c_k \langle \phi_k, \phi_j \rangle$$

for $j = 0, \ldots, n$. Thus orthogonality of the error implies that

$$\sum_{k=0}^{n} c_k \langle \phi_k, \phi_j \rangle = \langle f, \phi_j \rangle, \qquad j = 0, \ldots, n,$$

and these $n + 1$ equations precisely give $\mathbf{Gc} = \mathbf{b}$: since the coefficients of p satisfy the linear system that characterizes the (unique) least squares approximation, p must be that least squares approximation, $p = P_*$. Thus, orthogonality of the error $f - p$ with all $q \in \mathcal{P}_n$ implies that p is the least squares approximation. ∎

2.4.5 *Monomial basis*

Suppose we apply this method on the interval $[a, b] = [0, 1]$ with the monomial basis, $\phi_k(x) = x^k$. In that case,

$$\langle \phi_k, \phi_j \rangle = \langle x^k, x^j \rangle = \int_0^1 x^{j+k} \, \mathrm{d}x = \frac{1}{j + k + 1},$$

and the coefficient matrix has an elementary structure. In fact, this is a form of the notorious *Hilbert matrix*. It is exceptionally difficult to obtain accurate solutions with this matrix in floating point arithmetic, reflecting the fact that the monomials are a poor basis for $\mathcal{P}_n$ on $[0, 1]$. Let $\mathbf{G}$ denote the $n + 1$-dimensional Hilbert matrix, and suppose $\mathbf{b}$ is constructed so that the exact solution to the system $\mathbf{Gc} = \mathbf{b}$ is $\mathbf{c} = [1, 1, \ldots, 1]^T$. Let $\widehat{\mathbf{c}}$ denote computed solution to the system in MATLAB. Ideally the forward error $\|\mathbf{c} - \widehat{\mathbf{c}}\|_2$ will be nearly zero (if the rounding errors incurred while constructing $\mathbf{b}$ and solving the

See M.-D. Choi, 'Tricks or treats with the Hilbert matrix,' *American Math. Monthly* 90 (1983) 301–312.

system are small). Unfortunately, this is not the case – the condition number of $\mathbf{G}$ grows exponentially in the dimension n, and the accuracy of the computed solution to the linear system quickly degrades as n increases.

n	$\|\mathbf{G}\|\|\mathbf{G}^{-1}\|$	$\|\mathbf{c} - \widehat{\mathbf{c}}\|$
5	1.495×10^{7}	7.548×10^{-11}
10	1.603×10^{14}	0.01288
15	4.380×10^{17}	12.61
20	1.251×10^{18}	46.9

The last few *condition numbers* $\|\mathbf{G}\|\|\mathbf{G}^{-1}\|$ are in fact *smaller* than they ought to be: MATLAB computes the condition number based as the ratio of the largest to smallest singular values of $\mathbf{G}$; the smallest singular value can only be determined accurately if it is larger than about $\|\mathbf{G}\|\varepsilon_{\text{mach}}$, where $\varepsilon_{\text{mach}} \approx 2.2 \times 10^{-16}$. Thus, if the true condition number is larger than about $1/\varepsilon_{\text{mach}}$, we should not expect MATLAB to compute it accurately.

Clearly these errors are not acceptable!

In summary: *The monomial basis forms an ill-conditioned basis for $\mathcal{P}_n$ over the real interval $[a, b]$.*

2.4.6 Orthogonal basis

In the search for a basis for $\mathcal{P}_n$ that will avoid the numerical difficulties, let the structure of the equation $\mathbf{Gc} = \mathbf{b}$ be our guide. What choice of basis would make the matrix $\mathbf{G}$, written out in (2.15), as simple as possible? If the basis vectors are *orthogonal*, i.e.,

Section 2.5 will derive a procedure for computing an orthogonal basis for $\mathcal{P}_n$.

$$\langle \phi_j, \phi_k \rangle \begin{cases} \neq 0, & j = k; \\ = 0, & j \neq k, \end{cases}$$

then $\mathbf{G}$ only has nonzeros on the main *diagonal*, giving the system

$$\begin{bmatrix} \langle \phi_0, \phi_0 \rangle & 0 & \cdots & 0 \\ 0 & \langle \phi_1, \phi_1 \rangle & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & \langle \phi_n, \phi_n \rangle \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} \langle f, \phi_0 \rangle \\ \langle f, \phi_1 \rangle \\ \vdots \\ \langle f, \phi_n \rangle \end{bmatrix}.$$

This system decouples into $n + 1$ scalar equations $\langle \phi_j, \phi_j \rangle c_j = \langle f, \phi_j \rangle$ for $j = 0, \ldots, n$. Solve these scalar equations to get

$$c_j = \frac{\langle \phi_j, \phi_j \rangle}{\langle f, \phi_j \rangle}, \qquad j = 0, \ldots, n.$$

Thus, with respect to the orthogonal basis the least squares approximation to f is given by

$$P_*(x) = \sum_{j=0}^{n} c_j \phi_j(x) = \sum_{j=0}^{n} \frac{\langle f, \phi_j \rangle}{\langle \phi_j, \phi_j \rangle} \phi_j(x). \tag{2.17}$$

The formula (2.17) has an outstanding property: if we wish to extend approximation from $\mathcal{P}_n$ one degree higher to $\mathcal{P}_{n+1}$, we simply

add in one more term. If we momentarily use the notation $P_{*,k}$ for the least squares approximation from $\mathcal{P}_k$, then

$$P_{*,n+1}(x) = P_{*,n}(x) + \frac{\langle f, \phi_{n+1}\rangle}{\langle \phi_{n+1}, \phi_{n+1}\rangle}\phi_{n+1}(x).$$

In contrast, to increase the degree of the least squares approximation in the monomial basis, one would need to extend the $\mathbf{G}$ matrix by one row and column, and re-solve form $\mathbf{Gc} = \mathbf{b}$: *increasing the degree changes all the old coefficients in the monomial basis.*

An orthogonal basis also permits a beautifully simple formula for the norm of the error, $\|f - P_*\|_2$.

Theorem 2.9. Let $\phi_0, \ldots, \phi_n$ denote an orthogonal basis for $\mathcal{P}_n$. Then for any $f \in \mathbb{C}[a,b]$, the norm of the error $f - P_*$ of the least squares approximation $P_* \in \mathcal{P}_n$ is

$$\|f - P_*\|_2 = \sqrt{\|f\|_2^2 - \sum_{j=0}^{n} \frac{\langle f, \phi_j\rangle^2}{\langle \phi_j, \phi_j\rangle}}. \tag{2.18}$$

This result is closely related to *Parseval's identity*, which essentially says that if $\phi_0, \phi_1, \ldots$ forms an orthogonal basis for the (possibly infinite dimensional) vector space V, then for any $f \in V$,

$$\|f\|^2 = \sum_j \frac{\langle f, \phi_j\rangle^2}{\langle \phi_j, \phi_j\rangle}.$$

To put the utility of the formula (2.18) in context, think about minimax approximation. We have various bounds, like de la Vallée Poussin's theorem, on the minimax error, but no easy formula exists to give you that error directly.

Proof. First, use the formula (2.17) for P_* to compute

$$\begin{aligned}\|P_*\|_2^2 &= \left\langle \sum_{j=0}^{n} \frac{\langle f, \phi_j\rangle}{\langle \phi_j, \phi_j\rangle}\phi_j, \sum_{k=0}^{n} \frac{\langle f, \phi_k\rangle}{\langle \phi_k, \phi_k\rangle}\phi_k \right\rangle \\ &= \sum_{j=0}^{n}\sum_{k=0}^{n} \frac{\langle f, \phi_j\rangle}{\langle \phi_j, \phi_j\rangle}\frac{\langle f, \phi_k\rangle}{\langle \phi_k, \phi_k\rangle}\langle \phi_j, \phi_k\rangle,\end{aligned}$$

using linearity of the inner product. Since the basis polynomials are orthogonal, $\langle \phi_j, \phi_k\rangle = 0$ for $j \neq k$, which reduces the double sum to

$$\|P_*\|_2^2 = \sum_{j=0}^{n} \frac{\langle f, \phi_j\rangle}{\langle \phi_j, \phi_j\rangle}\frac{\langle f, \phi_j\rangle}{\langle \phi_j, \phi_j\rangle}\langle \phi_j, \phi_j\rangle = \sum_{j=0}^{n} \frac{\langle f, \phi_j\rangle^2}{\langle \phi_j, \phi_j\rangle}.$$

This calculation simplifies our primary concern:

$$\begin{aligned}\|f - P_*\|_2^2 = \langle f - P_*, f - P_*\rangle &= \langle f, f\rangle - \langle f, P_*\rangle - \langle P_*, f\rangle + \langle P_*, P_*\rangle \\ &= \langle f, f\rangle - 2\langle f, P_*\rangle + \langle P_*, P_*\rangle \\ &= \langle f, f\rangle - 2\left\langle f, \sum_{j=0}^{n} \frac{\langle f, \phi_j\rangle}{\langle \phi_j, \phi_j\rangle}\phi_j \right\rangle + \langle P_*, P_*\rangle \\ &= \langle f, f\rangle - 2\sum_{j=0}^{n} \frac{\langle f, \phi_j\rangle^2}{\langle \phi_j, \phi_j\rangle} + \sum_{j=0}^{n} \frac{\langle f, \phi_j\rangle^2}{\langle \phi_j, \phi_j\rangle} \\ &= \|f\|_2^2 - \sum_{j=0}^{n} \frac{\langle f, \phi_j\rangle^2}{\langle \phi_j, \phi_j\rangle},\end{aligned}$$

as required. ∎

2.4.7 Coda: Connection to discrete least squares (Optional Section)

Studies of numerical linear algebra inevitably address the *discrete least squares* problem: Given $\mathbf{A} \in \mathbb{R}^{m\times n}$ and $\mathbf{b} \in \mathbb{R}^m$ with $m \geq n$, solve

$$\min_{\mathbf{x}\in\mathbb{R}^n} \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2, \tag{2.19}$$

using the Euclidean norm $\|\mathbf{v}\|_2 = \sqrt{\mathbf{v}^*\mathbf{v}}$. One can show that the minimizing $\mathbf{x}$ solves the linear system

$$\mathbf{A}^*\mathbf{A}\mathbf{x} = \mathbf{A}^*\mathbf{b}, \tag{2.20}$$

which are called the *normal equations*. If $\text{rank}(\mathbf{A}) = n$ (i.e., the columns of $\mathbf{A}$ are linearly independent), then $\mathbf{A}^*\mathbf{A} \in \mathbb{R}^{n\times n}$ is invertible, and

$$\mathbf{x} = (\mathbf{A}^*\mathbf{A})^{-1}\mathbf{A}^*\mathbf{b}. \tag{2.21}$$

One learns that, for purposes of numerical stability, it is preferable to compute the *QR factorization*

$$\mathbf{A} = \mathbf{Q}\mathbf{R},$$

where the columns of $\mathbf{Q} \in \mathbb{R}^{m\times n}$ are orthonormal, $\mathbf{Q}^*\mathbf{Q} = \mathbf{I}$, and $\mathbf{R} \in \mathbb{R}^{n\times n}$ is upper triangular ($r_{j,k} = 0$ if $j > k$) and invertible if the columns of $\mathbf{A}$ are linearly independent. Substituting $\mathbf{QR}$ for $\mathbf{A}$ reduces the solution formula (2.21) to

$$\mathbf{x} = \mathbf{R}^{-1}\mathbf{Q}^*\mathbf{b}. \tag{2.22}$$

How does this "least squares problem" relate to the polynomial approximation problem in this section? We consider two perspectives.

2.4.8 Discrete least squares as subspace approximation

Notice that the he problem (2.19) can be viewed as

$$\min_{\mathbf{x}\in\mathbb{R}^n} \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2 = \min_{\mathbf{v}\in\text{Ran}(\mathbf{A})} \|\mathbf{b} - \mathbf{v}\|_2, \tag{2.23}$$

i.e., the discrete least squares problem seeks to approximate $\mathbf{b}$ with some vector $\mathbf{v} = \mathbf{A}\mathbf{x}$ from the subspace $\text{Ran}(\mathbf{A}) \subset \mathbb{R}^m$. Writing

$\text{Ran}(\mathbf{A}) = \{\mathbf{A}\mathbf{x} : \mathbf{x} \in \mathbb{R}^n\}$ is the range (column space) of $\mathbf{A}$.

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_1 & \cdots & \mathbf{a}_n \end{bmatrix}$$

for $\mathbf{a}_1, \ldots, \mathbf{a}_n \in \mathbb{R}^m$, we seek

$$\mathbf{v} = \mathbf{A}\mathbf{x} = x_1\mathbf{a}_1 + \cdots x_n\mathbf{a}_n \in \mathbb{R}^m$$

to approximate $\mathbf{b} \in \mathbb{R}^m$.

Viewing $\mathbf{a}_1, \ldots, \mathbf{a}_n$ as a basis for the approximating subspace $\mathrm{Ran}(\mathbf{A})$, one can develop the least squares theory precisely as we have earlier in this section, using the inner product

$$\langle \mathbf{a}_j, \mathbf{a}_k \rangle = \mathbf{a}_k^* \mathbf{a}_j.$$

Minimizing the error function

$$E(x_1, \ldots, x_n) = \|\mathbf{b} - (x_1\mathbf{a}_1 + \cdots + x_n\mathbf{a}_n)\|_2^2$$

with respect to $x_1, \ldots, x_n$ just as in the previous development leads to the Gram matrix problem

$$\begin{bmatrix} \langle \mathbf{a}_1, \mathbf{a}_1 \rangle & \langle \mathbf{a}_1, \mathbf{a}_2 \rangle & \cdots & \langle \mathbf{a}_1, \mathbf{a}_n \rangle \\ \langle \mathbf{a}_2, \mathbf{a}_1 \rangle & \langle \mathbf{a}_2, \mathbf{a}_2 \rangle & \cdots & \langle \mathbf{a}_2, \mathbf{a}_n \rangle \\ \vdots & \vdots & \ddots & \vdots \\ \langle \mathbf{a}_n, \mathbf{a}_1 \rangle & \langle \mathbf{a}_n, \mathbf{a}_2 \rangle & \cdots & \langle \mathbf{a}_n, \mathbf{a}_n \rangle \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} \langle \mathbf{b}, \mathbf{a}_1 \rangle \\ \langle \mathbf{b}, \mathbf{a}_2 \rangle \\ \vdots \\ \langle \mathbf{b}, \mathbf{a}_n \rangle \end{bmatrix}, \tag{2.24}$$

which is a perfect analogue of (2.15). In fact, notice that (2.24) is nothing other than

$$\mathbf{A}^*\mathbf{A}\mathbf{x} = \mathbf{A}^*\mathbf{b},$$

the familiar normal equations! What role does the QR factorization play? The columns of $\mathbf{Q}$ form an orthonormal basis for $\mathrm{Ran}(\mathbf{A})$:

$$\mathrm{Ran}(\mathbf{A}) = \mathrm{span}\{\mathbf{a}_1, \ldots, \mathbf{a}_n\} = \mathrm{span}\{\mathbf{q}_1, \ldots, \mathbf{q}_n\} = \mathrm{Ran}(\mathbf{Q}).$$

So the approximation problem (2.23) is equivalent to

$$\begin{aligned} \min_{\mathbf{x} \in \mathbb{R}^n} \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2 &= \min_{\mathbf{v} \in \mathrm{Ran}(\mathbf{Q})} \|\mathbf{b} - \mathbf{v}\|_2 \\ &= \min_{c_1, \ldots, c_n} \|\mathbf{b} - (c_1\mathbf{q}_1 + \cdots + c_n\mathbf{q}_n)\|_2. \end{aligned}$$

The Gram matrix system with respect to this basis is

$$\begin{bmatrix} \langle \mathbf{q}_1, \mathbf{q}_1 \rangle & \langle \mathbf{q}_1, \mathbf{q}_2 \rangle & \cdots & \langle \mathbf{q}_1, \mathbf{q}_n \rangle \\ \langle \mathbf{q}_2, \mathbf{q}_1 \rangle & \langle \mathbf{q}_2, \mathbf{q}_2 \rangle & \cdots & \langle \mathbf{q}_2, \mathbf{q}_n \rangle \\ \vdots & \vdots & \ddots & \vdots \\ \langle \mathbf{q}_n, \mathbf{q}_1 \rangle & \langle \mathbf{q}_n, \mathbf{q}_2 \rangle & \cdots & \langle \mathbf{q}_n, \mathbf{q}_n \rangle \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} \langle \mathbf{b}, \mathbf{q}_1 \rangle \\ \langle \mathbf{b}, \mathbf{q}_2 \rangle \\ \vdots \\ \langle \mathbf{b}, \mathbf{q}_n \rangle \end{bmatrix}. \tag{2.25}$$

The orthonormality of the vectors $\mathbf{q}_1, \ldots, \mathbf{q}_n$ means that $\mathbf{q}_j^*\mathbf{q}_k = 0$ if $j \neq k$ and $\mathbf{q}_j^*\mathbf{q}_j = \|\mathbf{q}_j\|_2^2 = 1$, and so the matrix in (2.25) is the identity. Hence

$$c_j = \langle \mathbf{b}, \mathbf{q}_j \rangle,$$

so we can write

$$\mathbf{c} = \mathbf{Q}^*\mathbf{b}.$$

The approximation $\mathbf{v}$ to $\mathbf{b}$ is then

$$\mathbf{v} = c_1\mathbf{q}_1 + \cdots + c_n\mathbf{q}_n = \mathbf{Q}\mathbf{c} = \mathbf{Q}\mathbf{Q}^*\mathbf{b}.$$

Now using the fact that $\mathbf{Q}^*\mathbf{Q} = \mathbf{I}$,

$$\begin{aligned}\mathbf{Q}\mathbf{Q}^* &= (\mathbf{A}\mathbf{R}^{-1})(\mathbf{A}\mathbf{R}^{-1})^* \\ &= \mathbf{A}(\mathbf{R}^*\mathbf{R})^{-1}\mathbf{A}^* = \mathbf{A}(\mathbf{A}^*\mathbf{A})^{-1}\mathbf{A}^*.\end{aligned}$$

Thus the least squares approximation to $\mathbf{b}$ is

$$\mathbf{v} = \mathbf{Q}\mathbf{Q}^*\mathbf{b} = \mathbf{A}\big((\mathbf{A}^*\mathbf{A})^{-1}\mathbf{A}^*\mathbf{b}\big) = \mathbf{A}\mathbf{x},$$

where $\mathbf{x}$ solves the original least squares problem (2.19).

Thus, the orthogonal basis for the approximating space $\mathrm{Ran}(\mathbf{A})$ leads to an easy formula for the approximation, in just the same fashion that orthogonal polynomials made quick work of the polynomial least squares problem in (2.17).

2.4.9 *Discrete least squares for polynomial approximation*

Now we turn the tables for another view of the connection between the the polynomial approximation problem and the matrix least squares problem (2.19).

Suppose we only know how to solve discrete least squares problems like (2.19), and want to use that technology to construct some polynomial $p \in P_n$ that approximates $f \in C[a,b]$ over $x \in [a,b]$.

We could sample f at, say, $m+1$ discrete points $x_0, \ldots, x_m$ uniformly distributed over $[a,b]$: set $h_m := (b-a)/m$ and let

$$x_k = a + kh_m.$$

We then want to solve

$$\min_{p\in\mathcal{P}_n} \sum_{j=0}^{m} |f(x_j) - p(x_j)|^2. \tag{2.26}$$

This least squares error, when scaled by h_m, takes the form of a Riemann sum that, in the $m \to \infty$ limit, approximates an integral:

$$\lim_{m\to\infty}\ h_m \sum_{k=0}^{m} (f(x_k) - p(x_k))^2 = \int_a^b (f(x) - p(x))^2\,\mathrm{d}x.$$

That is, as we take more and more approximation points, the error (2.26) that we are minimizing better and better approximates the integral error formulation (2.12).

To solve (2.26), represent $p \in \mathcal{P}_n$ using the monomial basis,

$$p(x) = c_0 + c_1 x + \cdots + c_n x^n.$$

Then write (2.26) as

$$\min_{c_0,\ldots,c_n} \|\mathbf{f} - \mathbf{A}\mathbf{c}\|_2^2,$$

where

$$\mathbf{A} = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ 1 & x_2 & x_2^2 & \cdots & x_2^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_m & x_m^2 & \cdots & x_m^n \end{bmatrix}, \qquad \mathbf{c} = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix}, \qquad \mathbf{f} = \begin{bmatrix} f(x_0) \\ f(x_1) \\ f(x_2) \\ \vdots \\ f(x_m) \end{bmatrix}.$$

This discrete problem can be solved via the *normal equations*, i.e., find $\mathbf{c} \in \mathbb{R}^{n+1}$ to solve the matrix equation

$$\mathbf{A}^*\mathbf{A}\mathbf{c} = \mathbf{A}^*\mathbf{f}.$$

Compute the right-hand side as

$$\mathbf{A}^*\mathbf{f} = \begin{bmatrix} \sum_{k=0}^n f(x_k) \\ \sum_{k=0}^n x_k f(x_k) \\ \sum_{k=0}^n x_k^2 f(x_k) \\ \vdots \\ \sum_{k=0}^n x_k^n f(x_k) \end{bmatrix} \in \mathbb{R}^{n+1}.$$

Notice that if $m+1$ approximation points are uniformly spaced over $[a,b]$, $x_k = a + kh_m$ for $h_m = (b-a)/m$, then

$$\lim_{m\to\infty} h_m \mathbf{A}^*\mathbf{f} = \begin{bmatrix} \int_a^b f(x)\,\mathrm{d}x \\ \int_a^b x f(x)\,\mathrm{d}x \\ \int_a^b x^2 f(x)\,\mathrm{d}x \\ \vdots \\ \int_a^b x^n f(x)\,\mathrm{d}x \end{bmatrix} = \begin{bmatrix} \langle f, 1\rangle \\ \langle f, x\rangle \\ \langle f, x^2\rangle \\ \vdots \\ \langle f, x^n\rangle \end{bmatrix},$$

which is precisely the right hand side vector $\mathbf{b} \in \mathbb{R}^{n+1}$ obtained for the original least squares problem at the beginning of this section in (2.15). Similarly, the $(j+1, k+1)$ entry of $\mathbf{A}^*\mathbf{A} \in \mathbb{R}^{(n+1)\times(n+1)}$ for the discrete problem can be formed as

$$(\mathbf{A}^*\mathbf{A})_{j+1,k+1} = \sum_{\ell=0}^m x_\ell^j x_\ell^k = \sum_{\ell=0}^m x_\ell^{j+k},$$

and thus for uniformly spaced approximation points,

$$\lim_{m\to\infty} h_m (\mathbf{A}^*\mathbf{A})_{j+1,k+1} = \int_a^b x^{j+k}\,\mathrm{d}x = \langle x^j, x^k\rangle.$$

Thus in aggregate we have

$$\lim_{m\to\infty} h_m \mathbf{A}^*\mathbf{A} = \mathbf{G},$$

where $\mathbf{G}$ is the same Gram matrix in (2.15).

We arrive at the following beautiful conclusion: The normal equations $\mathbf{A}^*\mathbf{A}\mathbf{c} = \mathbf{A}^*\mathbf{f}$ formed for polynomial approximation by *discrete least squares* converges to *exactly the same* $(n+1)\times(n+1)$ *system* $\mathbf{G}\mathbf{c} = \mathbf{b}$ that we independently derived for the polynomial approximation problem (2.12) with the integral form of the error.

LECTURE 19: *Orthogonal Polynomials, Part I*
LECTURE 20: *Orthogonal Polynomials, Part II*

2.5 Systems of orthogonal polynomials

Given a basis for $\mathcal{P}_n$, we can obtain the least squares approximation to $f \in C[a,b]$ by solving the linear system $\mathbf{Gc} = \mathbf{b}$ as described in Section 2.4.3. In particular, we could expand polynomials in any basis $\{\phi_k\}_{k=0}^n$ for $\mathcal{P}_n$,

$$p = \sum_{k=0}^{n} c_k \phi_k,$$

and then solve the system

$$\begin{bmatrix} \langle \phi_0, \phi_0 \rangle & \langle \phi_0, \phi_1 \rangle & \cdots & \langle \phi_0, \phi_n \rangle \\ \langle \phi_1, \phi_0 \rangle & \langle \phi_1, \phi_1 \rangle & & \vdots \\ \vdots & & \ddots & \vdots \\ \langle \phi_n, \phi_0 \rangle & \langle \phi_n, \phi_1 \rangle & \cdots & \langle \phi_n, \phi_n \rangle \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} \langle f, \phi_0 \rangle \\ \langle f, \phi_1 \rangle \\ \vdots \\ \langle f, \phi_n \rangle \end{bmatrix}.$$

If the basis is orthogonal, so that $\langle \phi_j, \phi_k \rangle = 0$ when $j \neq k$, the Gram matrix is diagonal, and the coefficients take the simple form

$$c_j = \frac{\langle f, \phi_j \rangle}{\langle \phi_j, \phi_j \rangle}.$$

The simplicity of this solution is compelling, but some work is required to construct a basis that has this orthogonality property. This section derives an efficient method to build this basis. In fact, for later use we slightly generalize the inner product to incorporate a *weight function*.

Definition 2.3. Given a function $w \in C[a,b]$ with $w(x) > 0$, the inner product of $f, g \in C[a,b]$ with respect to the weight w is

$$\langle f, g \rangle = \int_a^b f(x) g(x) w(x) \, \mathrm{d}x.$$

Generalizations are possible: for example, we can allow $w(x) = 0$ on a set of measure zero (e.g., finitely many points on $[a,b]$), and we can take $[a,b]$ to be the unbounded interval $[0,\infty)$ or $(-\infty,\infty)$, provided we are willing to restrict $C[a,b]$ to functions that have finite norm on these intervals.

One can confirm that this definition is consistent with the axioms required of an inner product that were described on page 97. For any such inner product, we then have the following definitions.

Definition 2.4. The functions $f, g \in C[a,b]$ are *orthogonal* if $\langle f, g \rangle = 0$.

Definition 2.5. A set of functions $\{\phi_k\}_{k=0}^n$ is a system of *orthogonal polynomials* provided:

- ϕ_k is a polynomial of exact degree k (with $\phi_0 \neq 0$);
- $\langle \phi_j, \phi_k \rangle = 0$ when $j \neq k$.

Be sure not to overlook the first property, that ϕ_k has exact degree k; it ensures the following result.

Lemma 2.2. The system of orthogonal polynomials $\{\phi_k\}_{k=0}^{\ell}$ is a basis for $\mathcal{P}_\ell$, for all $\ell = 0, \ldots, n$.

The proof follows by observing that the exact degree property ensures that $\phi_0, \ldots, \phi_\ell$ are $\ell + 1$ linearly independent vectors in the $\ell + 1$-dimensional subspace $\mathcal{P}_n$.We can apply it to derive the next lemma, one we will use repeatedly.

Lemma 2.3. Let $\{\phi_j\}_{j=0}^{n}$ be a system of orthogonal polynomials. Then $\langle p, \phi_n \rangle = 0$ for any $p \in \mathcal{P}_{n-1}$.

Proof. Lemma 2.2 ensures that $\{\phi_k\}_{j=0}^{n-1}$ is a basis for $\mathcal{P}_{n-1}$. Thus for any $p \in \mathcal{P}_{n-1}$, one can determine constants $c_0, \ldots, c_{n-1}$ such that

$$p = \sum_{j=0}^{n-1} c_j \phi_j.$$

The linearity of the inner product and orthogonality of $\{\phi_j\}_{j=0}^{n}$ imply

$$\langle p, \phi_n \rangle = \left\langle \sum_{j=0}^{n-1} c_j \phi_j, \phi_n \right\rangle = \sum_{j=0}^{n-1} c_j \langle \phi_j, \phi_n \rangle = \sum_{j=0}^{n-1} 0 = 0,$$

as required. ∎

We need a mechanism for constructing orthogonal polynomials. The Gram–Schmidt process used to orthogonalize vectors in $\mathbb{R}^n$ can readily be generalized to the present setting. Suppose that we have some $(n + 1)$-dimensional subspace $\mathcal{S}$ with the basis $p_0, p_1, \ldots, p_n$. Then the classical Gram–Schmidt algorithm takes the following form.

Algorithm 2.1 (Gram–Schmidt orthogonalization, prototype)**.**

Given a basis $\{p_0, \ldots, p_n\}$ for some subspace $\mathcal{S}$, the following algorithm constructs an orthogonal basis $\{\phi_0, \ldots, \phi_n\}$ for $\mathcal{S}$:

$\phi_0 := p_0$
for $k = 1, \ldots, n$
  $\phi_k := p_k - \Big($least squares approximation to p_k from span$\{\phi_0, \ldots, \phi_{k-1}\}\Big)$
end

Focus on the construction of ϕ_k: By the orthogonality of the error in least squares approximation (Theorem 2.8, which holds for the general inner products described above), ϕ_k is orthogonal to $\phi_0, \ldots, \phi_{k-1}$. Moreover, ϕ_k is not zero, since the linear independence of $p_0, \ldots, p_n$ ensures

$$p_k \notin \text{span}\{p_0, \ldots, p_{k-1}\} = \text{span}\{\phi_0, \ldots, \phi_{k-1}\}.$$

Since the basis $\phi_0, \ldots, \phi_{k-1}$ is orthogonal, the best approximation to p_k can be easily written down via Theorem 2.9 as

$$\sum_{j=0}^{k-1} \frac{\langle p_k, \phi_j \rangle}{\langle \phi_j, \phi_j \rangle} \phi_j.$$

Thus the Gram–Schmidt process takes this more precise formulation.

Algorithm 2.2 (Gram–Schmidt orthogonalization, general basis)**.**

Given a basis $\{p_0, \ldots, p_n\}$ for some subspace $\mathcal{S}$, the following algorithm constructs an orthogonal basis $\{\phi_0, \ldots, \phi_n\}$ for $\mathcal{S}$:

$\phi_0 := p_0$
for $k = 1, \ldots, n$
    $\phi_k := p_k - \sum_{j=0}^{k-1} \frac{\langle p_k, \phi_j \rangle}{\langle \phi_j, \phi_j \rangle} \phi_j$
end

This is a convenient process, but like the vector Gram–Schmidt process the amount of work required at each step grows as k increases, as p_k must be orthogonalized against more ϕ_j polynomials. Fortunately, there is a slick way to choose the initial basis $\{p_0, \ldots, p_n\}$ for which the Gram–Schmidt process takes a much simpler form.

Suppose one has a set of orthogonal polynomials, $\{\phi_j\}_{j=0}^{k-1}$, and seeks the next orthogonal polynomial, ϕ_k. Since ϕ_{k-1} has exact degree $k-1$, the polynomial $x\phi_{k-1}(x)$ has exact degree k, and hence $x\phi_{k-1}(x) \in \mathcal{P}_k$ but

$$x\phi_{k-1}(x) \notin \text{span}\{\phi_0, \ldots, \phi_{k-1}\}.$$

Thus, we could apply a Gram–Schmidt step to orthogonalize $x\phi_{k-1}(x)$ against $\phi_0, \ldots, \phi_{k-1}$ to get a new orthogonal basis vector, ϕ_k, giving

$$\mathcal{P}_k = \text{span}\{\phi_0, \ldots, \phi_k\}.$$

What is special about the special choice $p_k(x) = x\phi_{k-1}(x)$? It will gives an essential simplification to the customary Gram–Schmidt recurrence

$$\phi_k(x) = x\phi_{k-1}(x) - \sum_{j=0}^{k-1} \frac{\langle x\phi_{k-1}(x), \phi_j(x) \rangle}{\langle \phi_j, \phi_j \rangle} \phi_j(x).$$

The trick is that the x in $x\phi_{k-1}(x)$ can be flipped to the other side of the inner product,

$$\begin{aligned}
\langle x\phi_{k-1}(x), \phi_j(x) \rangle &= \int_a^b \Big(x\phi_{k-1}(x) \Big) \phi_j(x) w(x) \, \mathrm{d}x \\
&= \int_a^b \phi_{k-1}(x) \Big(x\phi_j(x) \Big) w(x) \, \mathrm{d}x \\
&= \langle \phi_{k-1}(x), x\phi_j(x) \rangle.
\end{aligned}$$

Now, recall from Lemma 2.3 that ϕ_{k-1} is orthogonal to all polynomials of degree less that $k-1$. Thus since $x\phi_j(x) \in \mathcal{P}_{j+1}$, if $j+1 < k-1$,

$$\langle x\phi_{k-1}(x), \phi_j(x)\rangle = \langle \phi_{k-1}(x), x\phi_j(x)\rangle = 0,$$

allowing us to neglect all terms in the Gram–Schmidt sum for which $j < k-2$:

$$\sum_{j=0}^{k-1} \frac{\langle x\phi_{k-1}(x), \phi_j(x)\rangle}{\langle \phi_j, \phi_j\rangle}\phi_j = \sum_{j=k-2}^{k-1} \frac{\langle x\phi_{k-1}(x), \phi_j(x)\rangle}{\langle \phi_j, \phi_j\rangle}\phi_j.$$

Thus we can compute orthogonal polynomials efficiently, even if the necessary polynomial degree is large. This fact has vital implications in numerical linear algebra: indeed, it is a reason that the iterative conjugate gradient method for solving $\mathbf{Ax} = \mathbf{b}$ often executes with blazing speed, but that is a story for another class.

The Gram–Schmidt process does not simplify to a short recurrence in all settings. We used the key fact $\langle x\phi_n, \phi_k\rangle = \langle \phi_n, x\phi_k\rangle$, *which does not hold in general inner product spaces*, but works perfectly well in our present setting because *our polynomials are real valued on* $[a,b]$. The short recurrence does not hold, for example, if you compute orthogonal polynomials over a general complex domain, instead of the real interval $[a,b]$.

Theorem 2.10 (Three-Term Recurrence for Orthogonal Polynomials)**.** Given a weight function $w(x)$ ($w(x) \geq 0$ for all $x \in (a,b)$, and $w(x) = 0$ only on a set of measure zero), a real interval $[a,b]$, and an associated real inner product

$$\langle f, g\rangle = \int_a^b w(x)f(x)g(x)\,\mathrm{d}x,$$

then a system of (monic) orthogonal polynomials $\{\phi_k\}_{k=0}^n$ can be generated as follows:

$$\phi_0(x) = 1,$$
$$\phi_1(x) = x - \frac{\langle x, 1\rangle}{\langle 1, 1\rangle},$$
$$\phi_k(x) = x\phi_{k-1}(x) - \frac{\langle x\phi_{k-1}(x), \phi_{k-1}(x)\rangle}{\langle \phi_{k-1}(x), \phi_{k-1}(x)\rangle}\phi_{k-1}(x) - \frac{\langle x\phi_{k-1}(x), \phi_{k-2}(x)\rangle}{\langle \phi_{k-2}(x), \phi_{k-2}(x)\rangle}\phi_{k-2}(x), \qquad \text{for } k \geq 2.$$

The above process constructs *monic* orthogonal polynomials, i.e., ϕ_k has leading term x^k. Other normalizations can be imposed with simple modifications to the Gram–Schmidt algorithm that preserve the three-term recurrence structure. In some settings other normalizations are more convenient, e.g., $\|\phi_k\|^2 = \langle \phi_k, \phi_k\rangle = 1$ or $\phi(0) = 1$.

We next illustrate these ideas for a particularly important class of orthogonal polynomials that use the constant weight $w(x) = 1$.

2.5.1 *Legendre polynomials*

On the interval $[a,b] = [-1,1]$ with weight $w(x) = 1$ for all x, the orthogonal polynomials are known as *Legendre polynomials*. Start with $\phi_0(x) = 1$, and then orthogonalize $\phi_1(x) = x\phi_0(x) = x$; since ϕ_0 is

even and ϕ_1 is odd over $x \in [-1,1]$, $\langle x, 1\rangle = 0$, giving $\phi_1 = x$. This begins an inductive cascade: in the Gram–Schmidt process, for all k,

$$\langle x\phi_{k-1}(x), \phi_{k-1}\rangle = 0,$$

since if ϕ_{k-1} is even, $x\phi_{k-1}$ will be odd (or vice versa), and the inner product of even and odd functions with $w(x) = 1$ over $x \in [-1,1]$ is always zero. Thus for Legendre polynomials the conventional three-term recurrence in Theorem 2.10 reduces to

$$\phi_k(x) = x\phi_{k-1}(x) - \frac{\langle x\phi_{k-1}(x), \phi_{k-2}(x)\rangle}{\langle \phi_{k-2}(x), \phi_{k-2}(x)\rangle}\phi_{k-2}(x).$$

Legendre polynomials enjoy many nice properties and identities; with some extra work, one can simplify the coefficient multiplying ϕ_{k-2} to

$$\phi_k(x) = x\phi_{k-1}(x) - \frac{(k-1)^2}{4(k-1)^2 - 1}\phi_{k-2}(x).$$

The first few Legendre polynomials $\phi_0, \ldots, \phi_6$ are presented below (and plotted in the margin):

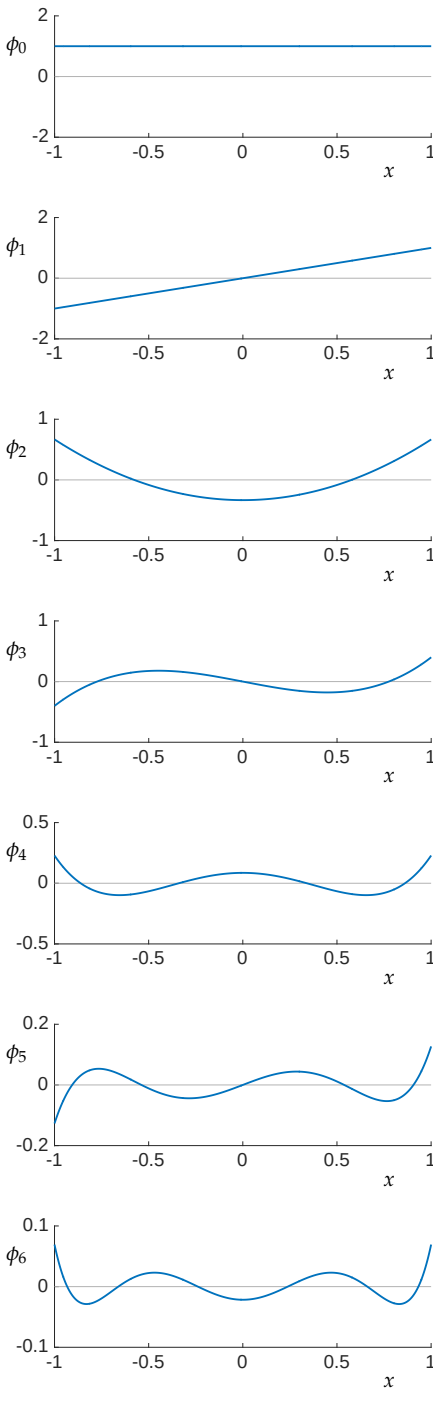
$$\begin{aligned}
\phi_0(x) &= 1 \\
\phi_1(x) &= x \\
\phi_2(x) &= x^2 - \tfrac{1}{3} \\
\phi_3(x) &= x^3 - \tfrac{3}{5}x \\
\phi_4(x) &= x^4 - \tfrac{6}{7}x^2 + \tfrac{3}{35} \\
\phi_5(x) &= x^5 - \tfrac{10}{9}x^3 + \tfrac{5}{21}x \\
\phi_6(x) &= x^6 - \tfrac{15}{11}x^4 + \tfrac{5}{11}x^2 - \tfrac{5}{231}.
\end{aligned}$$

Orthogonal polynomials play a key role in a prominent technique for computing integrals known as Gaussian quadrature. In that context, we will see other families of orthogonal polynomials: the Chebyshev, Laguerre, and Hermite polynomials. These polynomials differ in their weight functions and the intervals of $\mathbb{R}$ on which they are posed.

Example 2.5 ($f(x) = e^x$)**.** We repeat our previous example: approximating $f(x) = e^x$ on $[0,1]$ with a linear polynomial. First, we need to construct orthogonal polynomials for this interval. Set $\phi_0(x) = 1$; then a straightforward computation gives $\phi_1(x) = x - 1/2$. We then compute

$$\begin{aligned}
\langle e^x, \phi_0(x)\rangle &= \int_0^1 e^x \, dx = e - 1 \\
\langle e^x, \phi_1(x)\rangle &= \int_0^1 e^x (x - 1/2) \, dx = (3 - e)/2,
\end{aligned}$$

The recurrence for the monic Legendre polynomial is given, e.g., by Dahlquist and Björck, *Numerical Methods in Scientific Computing*, vol. 1, p. 571. In contrast to these monic polynomials, Legendre polynomials are, by longstanding tradition, usually normalized so that $\phi_k(0) = 1$.



and

$$\langle \phi_0, \phi_0 \rangle = \int_0^1 1^2 \, \mathrm{d}x = 1$$
$$\langle \phi_1, \phi_1 \rangle = \int_0^1 (x - 1/2)^2 \, \mathrm{d}x = 1/12.$$

Assemble these inner products to obtain a formulas for P_*:

$$\begin{aligned} P_*(x) &= \frac{\langle \mathrm{e}^x, \phi_0 \rangle}{\langle \phi_0, \phi_0 \rangle} \phi_0(x) + \frac{\langle \mathrm{e}^x, \phi_0 \rangle}{\langle \phi_1, \phi_1 \rangle} \phi_1(x) \\ &= \frac{\mathrm{e} - 1}{1} 1 + \frac{(3 - \mathrm{e})/2}{1/12} (x - 1/2) \\ &= (\mathrm{e} - 1) + (18 - 6\mathrm{e})(x - 1/2) \\ &= 4\mathrm{e} - 10 + x(18 - 6\mathrm{e}). \end{aligned}$$

Note that this is exactly the polynomial we obtained in Example 2.4 using the monomial basis.

With this procedure, one can easily to increase the degree of the approximating polynomial. To increase the degree by one, simply add

$$\frac{\langle f, \phi_{n+1} \rangle}{\langle \phi_{n+1}, \phi_{n+1} \rangle} \phi_{n+1}$$

to the old approximation. Were we using a general (non-orthogonal) basis, to increase the degree of the approximation we would need to solve a new $(n+2)$-by-$(n+2)$ linear system to find the coefficients of the least squares approximation. Indeed, an advantage to the new method is that we express the optimal polynomial in a 'good' basis—the basis of orthonormal polynomials—rather than the monic polynomial basis.

It is true, however, that both these methods for finding the least squares polynomial will generally be more expensive then simply finding a polynomial interpolant.

3
Quadrature

LECTURE 21: *Interpolatory Quadrature Rules*

The past two chapters have developed a suite of tools for polynomial interpolation and approximation. We shall now apply these tools toward the approximation of definite integrals.

To compute the least squares approximations discussed in Section 2.4, one needs to compute integrals for the inner products

$$\langle f, \phi_j \rangle = \int_a^b f(x)\phi_j(x)\,\mathrm{d}x$$

that form the right-hand side of the Gram matrix equation $\mathbf{G}\mathbf{c} = \mathbf{b}$. Of course, many other applications require the evaluation of definite integrals; integrals across (many) different variables pose additional challenges.

Many definite integrals are difficult or impossible to evaluate exactly, so our next charge is to develop algorithms that approximate such integrals quickly and accurately. This field is known as *quadrature*, a name that suggests the approximation of the area under a curve by area of subtending quadrilaterals. (a "Riemann sum").

The term *quadrature* is used to distinguish the numerical approximation of a definite integral from the numerical solution of an ordinary differential equation, which is often called *numerical integration*. Approximation of a double integral is sometimes called *cubature*.

For more details on quadrature rules, see Süli and Mayers, Chapter 7, which has guided many aspects of our presentation here.

3.1 *Interpolatory Quadrature*

Given $f \in C[a, b]$, we seek approximations to the definite integral

$$\int_a^b f(x)\,\mathrm{d}x.$$

All the methods we consider in these notes are variants of *interpolatory quadrature rules*, meaning that they approximate the integral of f by the exact integral of a polynomial interpolant to f:

$$\int_a^b f(x)\,\mathrm{d}x \approx \int_a^b p_n(x)\,\mathrm{d}x,$$

where $p_n \in \mathcal{P}_n$ interpolates f at $n+1$ points in $[a,b]$. Will such rules produce a reasonable estimate to the integral? Of course, that depends on properties of f and the interpolation points.

Our goal in this section is to develop a convenient formula for the approximation

$$\int_a^b p_n(x)\,\mathrm{d}x$$

that will not require the explicit construction of p_n. As is often the case, the task becomes direct and simple if we express the interpolant in the correct basis. Recall the Lagrange form of the interpolant presented in Section 1.5: Given $n+1$ distinct interpolation points

$$x_0, \ldots, x_n \in [a,b],$$

the interpolant can be written as

$$p_n(x) = \sum_{j=0}^{n} f(x_j)\ell_j(x), \tag{3.1}$$

where the basis functions $\ell_0, \ldots, \ell_n$ take the familiar form

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

The integral of p_n can then be computed in terms of the integral of the basis functions:

$$\int_a^b p_n(x)\,\mathrm{d}x = \int_a^b \sum_{j=0}^{n} f(x_j)\ell_j(x)\,\mathrm{d}x = \sum_{j=0}^{n} f(x_j) \int_a^b \ell_j(x)\,\mathrm{d}x.$$

Why is the Lagrange basis special? Could you not do the same kind of expansion with the monomial or Newton bases? Yes indeed: but then you would need to compute the coefficients c_j that multiply these basis functions in the expansion $p_n(x) = \sum c_j \phi_j(x)$, which requires the solution of a (nontrivial) linear system. The beauty of the Lagrange approach is that these coefficients are instantly available by evaluating f at the quadrature nodes: $c_j = f(x_j)$.

In the nomenclature of quadrature rules, the integrals of the basis functions are called *weights*, denoted

$$w_j := \int_a^b \ell_j(x)\,\mathrm{d}x.$$

The degree-n interpolatory quadrature rule at distinct *nodes* $x_0, \ldots, x_n$ takes the form

$$\int_a^b f(x)\,\mathrm{d}x \approx \sum_{j=0}^{n} w_j f(x_j),$$

for the *weights*

$$w_j = \int_a^b \ell_j(x)\,\mathrm{d}x.$$

It is worth stating an obvious theorem, which we will revisit in future lectures.

Theorem 3.1 (Exactness of Interpolatory Quadrature)**.**
The degree-n interpolatory quadrature rule at distinct points $x_0, \ldots, x_n \in [a, b]$ is exact for any polynomial of degree n or less: if $f \in \mathcal{P}_n$, then

$$\int_a^b f(x)\,\mathrm{d}x = \sum_{j=0}^{n} w_j f(x_j).$$

The proof is simple: If $f \in \mathcal{P}_n$, its polynomial interpolant $p_n \in \mathcal{P}_n$ is exactly f, and so the exact integral of p_n is the same thing as the exact integral of f. However, the result is not inconsequential. There are some circumstances in numerical computations where it is easier to use a quadrature rule to evaluate the integral of a polynomial, rather than computing the integral directly from the polynomial coefficients.

Finite element methods give one such setting, where in some cases f is represented by its values $f(x_j)$ on a computational mesh, rather than by its coefficients.

It is no surprise that a quadrature rule based on a degree-n interpolant will exactly integrate $f \in \mathcal{P}_n$. However, in special cases a degree-n interpolant will *exactly integrate polynomials of higher degree*. This motivates the next definition.

Definition 3.1. An interpolatory quadrature rule has *degree of exactness* m if for all $f \in \mathcal{P}_m$,

$$\int_a^b f(x)\,\mathrm{d}x = \sum_{j=0}^{n} w_j f(x_j).$$

By Theorem 3.1, a degree-n quadrature rule has degree of exactness $m \geq n$. Thus it will be particularly interesting to see circumstances in which this degree of exactness is exceeded.

LECTURE 22: *Newton–Cotes quadrature*

3.2 *Newton–Cotes quadrature*

You encountered the most basic method for approximating an integral when you learned calculus: the Riemann integral is motivated by approximating the area under a curve by the area of rectangles that touch that curve, which gives a rough estimate that becomes increasingly accurate as the width of those rectangles shrinks. This amounts to approximating the function f by a piecewise constant interpolant, and then computing the exact integral of the interpolant. When only one rectangle is used to approximate the entire integral, we have the most simple *Newton–Cotes* formula; see Figure 3.1.

Newton–Cotes formulas are interpolatory quadrature rules where the quadrature notes $x_0, \ldots, x_n$ are uniformly spaced over $[a, b]$,

$$x_j = j\left(\frac{b-a}{n}\right).$$

Given the lessons we learned about polynomial interpolation at uniformly spaced points in Section 1.6, you should rightly be suspicious of applying this idea with large n (i.e., high degree interpolants). A more reliable way to increase accuracy follows the lead of basic Riemann sums: partition $[a, b]$ into smaller subintervals, and use low-degree interpolants to approximate the integral on each of these smaller domains. Such methods are called *composite* quadrature rules.

In some cases, the function f may be fairly regular over most of the domain $[a, b]$, but then have some small region of rapid growth or oscillation. Modern *adaptive* quadrature rules are composite rules on which the subintervals of $[a, b]$ vary in size, depending on estimates of how rapidly f is changing in a given part of the domain. Such methods seek to balance the competing goals of highly accurate approximate integrals and as few evaluations of f as possible.We shall not dwell much on these sophisticated quadrature procedures here, but rather start by understanding some methods you were probably introduced to in your first calculus class.

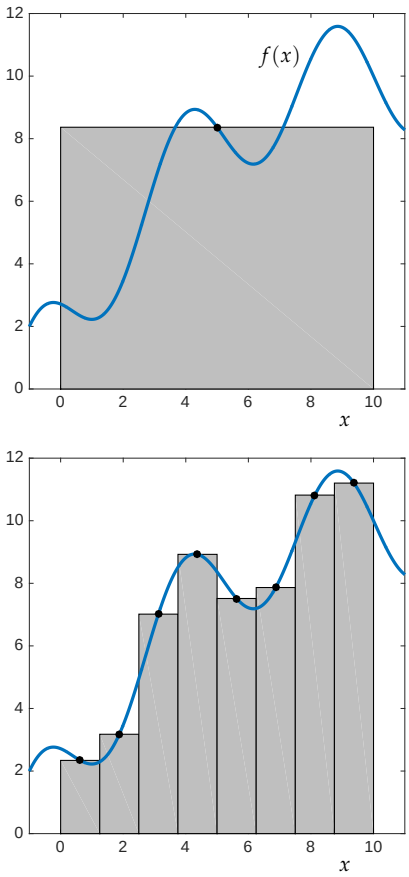


Figure 3.1: Estimates of $\int_0^{10} f(x)\,\mathrm{d}x$, shown in gray: the first approximates f by a constant interpolant; the second, a *composite rule*, uses a piecewise constant interpolant. You probably have encountered this second approximation as a *Riemann sum*.

3.2.1 *The trapezoid rule*

The trapezoid rule is a simple improvement over approximating the integral by the area of a single rectangle. A linear interpolant to f can be constructed, requiring evaluation of f at the interval end points $x_0 = a$ and $x_1 = b$. Using the interpolatory quadrature methodology

described in the last section, we write

$$p_1(x) = f(a)\left(\frac{x-b}{a-b}\right) + f(b)\left(\frac{x-a}{b-a}\right),$$

and compute its integral as

$$\begin{aligned}\int_a^b p_1(x)\,\mathrm{d}x &= \int_a^b f(a)\left(\frac{x-b}{a-b}\right) + f(b)\left(\frac{x-a}{b-a}\right)\mathrm{d}x \\ &= f(a)\int_a^b \frac{x-x_1}{x_0-x_1}\,\mathrm{d}x + f(b)\int_a^b \frac{x-x_1}{x_0-x_1}\,\mathrm{d}x \\ &= f(a)\left(\frac{b-a}{2}\right) + f(b)\left(\frac{b-a}{2}\right).\end{aligned}$$

In summary,

Trapezoid rule:

$$\int_a^b f(x)\,\mathrm{d}x \approx \frac{b-a}{2}\Big(f(a)+f(b)\Big).$$

The procedure behind the trapezoid rule is illustrated in Figure 3.2 where the area approximating the integral is colored gray.

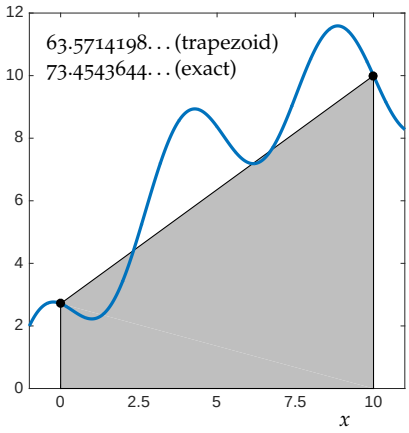


Figure 3.2: Trapezoid rule estimate of $\int_0^{10} f(x)\,\mathrm{d}x$, shown in gray.

To derive an error bound for the trapezoid rule, simply integrate the fundamental interpolation error formula in Theorem 1.3. That gave, for each $x \in [a,b]$, some $\xi \in [a,b]$ such that

$$f(x) - p_1(x) = \tfrac{1}{2}f''(\xi)(x-a)(x-b).$$

Note that ξ will vary with x, which we emphasize by writing $\xi(x)$. Integrate this formula to obtain

$$\begin{aligned}\int_a^b f(x)\,\mathrm{d}x - \int_a^b p_1(x)\,\mathrm{d}x &= \int_a^b \frac{1}{2}f''(\xi(x))(x-a)(x-b)\,\mathrm{d}x \\ &= \frac{1}{2}f''(\eta)\int_a^b (x-a)(x-b)\,\mathrm{d}x \\ &= \frac{1}{2}f''(\eta)\left(\frac{1}{6}a^3 - \frac{1}{2}a^2b + \frac{1}{2}ab^2 - \frac{1}{6}b^3\right) \\ &= -\frac{1}{12}f''(\eta)(b-a)^3\end{aligned}$$

for some $\eta \in [a,b]$. The second step follows from the mean value theorem for integrals.

In a forthcoming lecture we shall develop a much more general theory, based on the *Peano kernel*, from which we can derive this error

The mean value theorem for integrals states that if $h, g \in C[a,b]$ and h *does not change sign on* $[a,b]$, then there exists some $\eta \in [a,b]$ such that $\int_a^b g(t)h(t)\,\mathrm{d}t = g(\eta)\int_a^b h(t)\,\mathrm{d}t$. The requirement that h not change sign is essential. For example, if $g(t) = h(t) = t$ then $\int_{-1}^1 g(t)h(t)\,\mathrm{d}t = \int_{-1}^1 t^2\,\mathrm{d}t = 2/3$, yet $\int_{-1}^1 h(t)\,\mathrm{d}t = \int_{-1}^1 t\,\mathrm{d}t = 0$, so for all $\eta \in [-1,1]$, $g(\eta)\int_{-1}^1 h(t)\,\mathrm{d}t = 0 \neq \int_{-1}^1 g(t)h(t)\,\mathrm{d}t = 2/3$.

bound, plus bounds for more complicated schemes, too. For now, we summarize the bound in the following Theorem.

Theorem 3.2. Let $f \in C^2[a,b]$. The error in the trapezoid rule is

$$\int_a^b f(x)\,\mathrm{d}x - \left(\frac{b-a}{2}\left(f(a)+f(b)\right)\right) = -\frac{1}{12}f''(\eta)(b-a)^3$$

for some $\eta \in [a,b]$.

This bound has an interesting feature: if we are integrating over the small interval, $b - a = h \ll 1$, then the error in the trapezoid rule approximation is $\mathcal{O}(h^3)$ as $h \to 0$, while the error in the linear interpolant upon which this quadrature rule is based is only $\mathcal{O}(h^2)$ (from Theorem 1.3).

Example 3.1 ($f(x) = \mathrm{e}^x(\cos x + \sin x)$). Here we demonstrate the difference between the error for linear interpolation of a function, $f(x) = \mathrm{e}^x(\cos x + \sin x)$, between two points, $x_0 = 0$ and $x_1 = h$, and the trapezoid rule applied to the same interval. The theory reveals that linear interpolation will have an $\mathcal{O}(h^2)$ error as $h \to 0$, while the trapezoid rule has $\mathcal{O}(h^3)$ error, as confirmed in Figure 3.3.

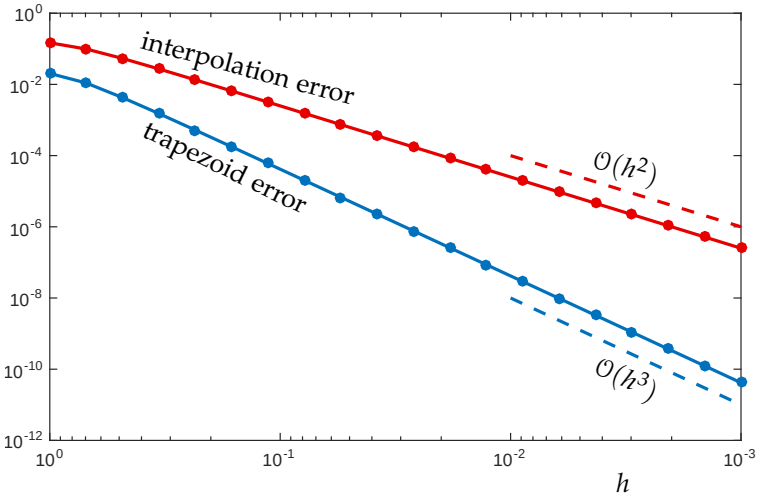


Figure 3.3: Error of linear interpolation and trapezoid rule approximation for $f(x) = \mathrm{e}^x(\cos x + \sin x)$ for $x \in [0,h]$ as $h \to 0$.

3.2.2 *Simpson's rule*

To improve the accuracy of the trapezoid rule, increment the degree of the interpolating polynomial. This will increase the number of evaluations of f (often very costly), but hopefully will significantly

decrease the error. Indeed it does – by an even greater margin than we might expect.

Simpson's rule integrates the quadratic interpolant $p_2 \in \mathcal{P}_2$ to f at the uniformly spaced points

$$x_0 = a, \qquad x_1 = (a+b)/2, \qquad x_2 = b.$$

Using the interpolatory quadrature formulation of the last section,

$$\int_a^b p_2(x)\,\mathrm{d}x = w_0 f(a) + w_1 f(\tfrac{1}{2}(a+b)) + w_2 f(c),$$

where

$$w_0 = \int_a^b \left(\frac{x-x_1}{x_0-x_1}\right)\left(\frac{x-x_2}{x_0-x_2}\right)\mathrm{d}x = \frac{b-a}{6}$$

$$w_1 = \int_a^b \left(\frac{x-x_0}{x_1-x_0}\right)\left(\frac{x-x_2}{x_1-x_2}\right)\mathrm{d}x = \frac{2(b-a)}{3}$$

$$w_2 = \int_a^b \left(\frac{x-x_0}{x_2-x_0}\right)\left(\frac{x-x_1}{x_2-x_1}\right)\mathrm{d}x = \frac{b-a}{6}.$$

In summary:

Simpson's rule:

$$\int_a^b f(x)\,\mathrm{d}x \approx \frac{b-a}{6}\left(f(a) + 4f(\tfrac{1}{2}(a+b)) + f(b)\right).$$

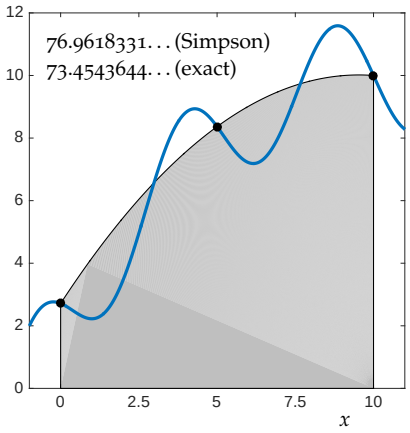


Figure 3.4: Simpson's rule estimate of $\int_0^{10} f(x)\,\mathrm{d}x$, shown in gray.

Simpson's rule enjoys a remarkable feature: though it only approximates f by a quadratic, *it integrates any cubic polynomial exactly*! One can verify this by directly applying Simpson's rule to a generic cubic polynomial. Write $f(x) = \alpha x^3 + q(x)$, where $q \in \mathcal{P}_2$. Let $I(f) = \int_a^b f(x)\,\mathrm{d}x$ and let $I_2(f)$ denote the Simpson's rule approximation. Then, by linearity of the integral,

$$I(f) = \alpha I(x^3) + I(q)$$

and, by linearity of Simpson's rule,

$$I_2(f) = \alpha I_2(x^3) + I_2(q).$$

Since Simpson's rule is an interpolatory quadrature rule based on quadratic polynomials, its degree of exactness must be at least 2 (Theorem 3.1), i.e., it exactly integrates q: $I_2(q) = I(q)$. Thus

$$I(f) - I_2(f) = \alpha\Big(I(x^3) - I_2(x^3)\Big).$$

So Simpson's rule will be exact for all cubics if it is exact for x^3. A simple computation gives

$$\begin{aligned} I_2(x^3) &= \frac{b-a}{6}\left(a^3 + 4\Big(\frac{a+b}{2}\Big)^3 + b^3\right) \\ &= \frac{b-a}{12}\left(3a^3 + 3a^2b + 3ab^2 + 3b^3\right) = \frac{b^4 - a^4}{4} = I(x^3), \end{aligned}$$

confirming that Simpson's rule is exact for x^3, and hence for all cubics. For now we simply state an error bound for Simpson's rule, which we will prove in a future lecture.

In fact, Newton–Cotes formulas based on approximating f by an even-degree polynomial always exactly integrate polynomials one degree higher.

Theorem 3.3. Let $f \in C^4[a,b]$. The error in the Simpson's rule is

$$\int_a^b f(x)\,\mathrm{d}x - \left(\frac{b-a}{6}\left(f(a) + 4f((a+b)/2) + f(b)\right)\right) = -\frac{1}{90}f^{(4)}(\eta)(b-a)^5$$

for some $\eta \in [a,b]$.

This error formula captures the fact that Simpson's rule is exact for cubics, since it features the fourth derivative $f^{(4)}(\eta)$, two derivatives greater than $f''(\eta)$ in the trapezoid rule bound, even though the degree of the interpolant has only increased by one. Perhaps it is helpful to visualize the exactness of Simpson's rule for cubics. Figure 3.5 shows $f(x) = x^3$ (blue) and its quadratic interpolant (red). On the left, the area under f is colored gray: its area is the integral we seek. On the right, the area under the interpolant is colored gray. Accounting area below the x axis as negative, both integrals give an identical value even though the functions are quite different. It is remarkable that this is the case for *all* cubics.

Typically one does not see Newton–Cotes rules based on polynomials of degree higher than two (i.e., Simpson's rule). Because it can be fun to see numerical mayhem, we give an example to emphasize why high-degree Newton–Cotes rules can be a bad idea. Recall that Runge's function $f(x) = 1/(1+x^2)$ gave a nice example for which the polynomial interpolant at uniformly spaced points over $[-5,5]$ fails to converge uniformly to f. This fact suggests that Newton–Cotes quadrature will also fail to converge as the degree of the interpolant grows. The exact value of the integral we seek is

Integrating the cubic interpolant at four uniformly spaced points is called *Simpson's three-eighths rule*.

$$\int_{-5}^{5} \frac{1}{1+x^2}\,\mathrm{d}x = 2\tan^{-1}(5) = 2.75680153\ldots.$$

Just as the interpolant at uniformly spaced points diverges, so too does the Newton–Cotes integral. Figure 3.6 illustrates this divergence, and shows that integrating the interpolant at Chebyshev

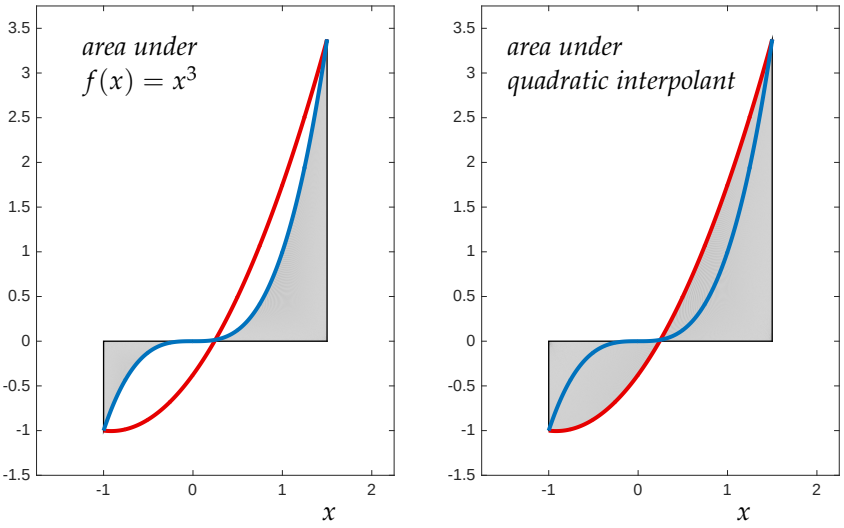


Figure 3.5: Simpson's rule applied to $f(x) = x^3$ on $x \in [-1, 3/2]$. The areas under $f(x)$ (blue) and its quadratic interpolant (red) are the same, even though the functions are quite different.

points, called *Clenshaw–Curtis quadrature*, does indeed converge. Section 3.3 describes this latter quadrature in more detail. Before discussing it, we describe a way to make Newton–Cotes rules more robust: integrate low-degree polynomials over subintervals of $[a, b]$.

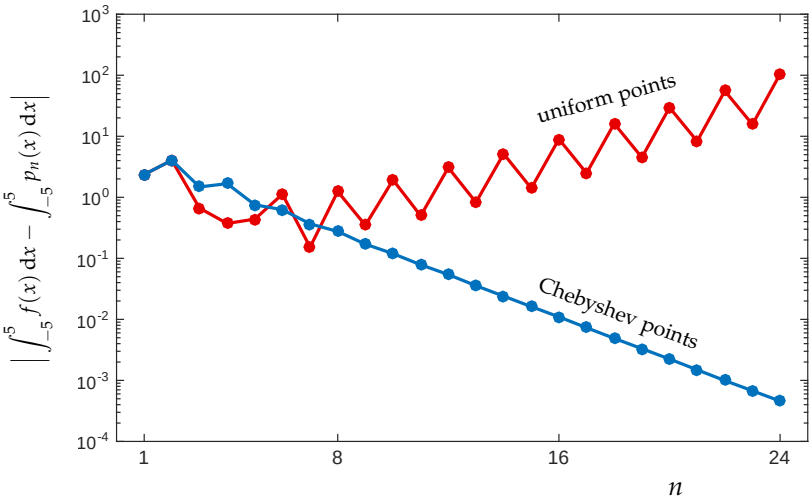


Figure 3.6: Integrating interpolants p_n at $n + 1$ uniformly spaced points (red) and at Chebyshev points (blue) for Runge's function, $f(x) = 1/(1 + x^2)$ over $x \in [-5, 5]$.

3.2.3 *Composite rules*

As an alternative to integrating a high-degree polynomial, one can pursue a simpler approach that is often very effective: Break the interval $[a, b]$ into subintervals, then apply a standard Newton–Cotes rule (e.g., trapezoid or Simpson) on each subinterval. Applying the

trapezoid rule on n subintervals gives

$$\int_a^b f(x)\,\mathrm{d}x = \sum_{j=1}^{n} \int_{x_{j-1}}^{x_j} f(x)\,\mathrm{d}x \approx \sum_{j=1}^{n} \frac{(x_j - x_{j-1})}{2}\Big(f(x_{j-1} + f(x_j)\Big).$$

The standard implementation assumes that f is evaluated at uniformly spaced points between a and b, $x_j = a + jh$ for $j = 0, \ldots, n$ and $h = (b-a)/n$, giving the following famous formulation:

Composite Trapezoid rule:

$$\int_a^b f(x)\,\mathrm{d}x \approx \frac{h}{2}\Big(f(a) + 2\sum_{j=1}^{n-1} f(a+jh) + f(b)\Big).$$

(Of course, one can readily adjust this rule by partitioning $[a,b]$ into subintervals of different sizes.) The error in the composite trapezoid rule can be derived by summing up the error in each application of the trapezoid rule:

$$\begin{aligned}\int_a^b f(x)\,\mathrm{d}x - \frac{h}{2}\Big(f(a) + 2\sum_{j=1}^{n-1} f(a+jh) + f(b)\Big) &= \sum_{j=1}^{n}\Big(-\tfrac{1}{12}f''(\eta_j)(x_j - x_{j-1})^3\Big) \\ &= -\frac{h^3}{12}\sum_{j=1}^{n} f''(\eta_j)\end{aligned}$$

for $\eta_j \in [x_{j-1}, x_j]$. We can simplify these f'' terms by noting that $\frac{1}{n}(\sum_{j=1}^{n} f''(\eta_j))$ is the average of n values of f'' evaluated at points in the interval $[a,b]$. Naturally, this average cannot exceed the maximum or minimum value that f'' assumes on $[a,b]$, so there exist points $\xi_1, \xi_2 \in [a,b]$ such that

$$f''(\xi_1) \le \frac{1}{n}\sum_{j=1}^{n} f''(\eta_j) \le f''(\xi_2).$$

Thus the intermediate value theorem guarantees the existence of some $\eta \in [a,b]$ such that

$$f''(\eta) = \frac{1}{n}\sum_{j=1}^{n} f''(\eta_j).$$

We arrive at a bound on the error in the composite trapezoid rule.

Theorem 3.4. Let $f \in C^2[a,b]$. The error in the composite trapezoid rule over n intervals of uniform width $h = (b-a)/n$ is

$$\int_a^b f(x)\,\mathrm{d}x - \frac{h}{2}\Big(f(a) + 2\sum_{j=1}^{n-1} f(a+jh) + f(b)\Big) = -\frac{h^2}{12}(b-a)f''(\eta).$$

for some $\eta \in [a,b]$.

This error analysis has an important consequence: *the error for the composite trapezoid rule is only* $\mathcal{O}(h^2)$, not the $\mathcal{O}(h^3)$ we saw for the usual trapezoid rule (in which case $b - a = h$ since $n = 1$).

A similar construction leads to the composite Simpson's rule. We now must ensure that n is even, since each interval on which we apply the standard Simpson's rule has width $2h$. Simple algebra leads to the following formula.

Composite Simpson's rule:

$$\int_a^b f(x)\,\mathrm{d}x \approx \frac{h}{3}\Big(f(a) \;+\; 4\sum_{j=1}^{n/2} f(a+(2j-1)h) \;+\; 2\sum_{j=1}^{n/2-1} f(a+2jh) \;+\; f(b)\Big).$$

Now use Theorem 3.3 to derive an error formula for the composite Simpson's rule, using the same approach as for the composite trapezoid rule.

Theorem 3.5. Let $f \in C^2[a,b]$. The error in the composite Simpson's rule over $n/2$ intervals of uniform width $2h = 2(b-a)/n$ is

$$\int_a^b f(x)\,\mathrm{d}x - \frac{h}{3}\Big(f(a) \;+\; 4\sum_{j=1}^{n/2} f(a+(2j-1)h) + \quad 2\sum_{j=1}^{n/2-1} f(a+2jh) \;+\; f(b)\Big)$$
$$= -\frac{h^4}{180}(b-a)f^{(4)}(\eta)$$

for some $\eta \in [a,b]$.

The illustrations in Figure 3.7 compare the composite trapezoid and Simpson's rules for the same number of function evaluations. One can see that Simpson's rule, in this typical case, gives considerably better accuracy.

Reflect for a moment. Suppose you are willing to evaluate f a fixed number of times. How can you get the most bang for your buck? If f is smooth, a rule based on a high-order interpolant (such as the Clenshaw–Curtis and Gaussian quadrature rules we will present in a few lectures) are likely to give the best result. If f is not smooth (e.g., with kinks, discontinuous derivatives, etc.), then a robust composite rule would be a good option. (A famous special case: If the function f is sufficiently smooth and is periodic with period $b - a$, then the trapezoid rule converges *exponentially*.)

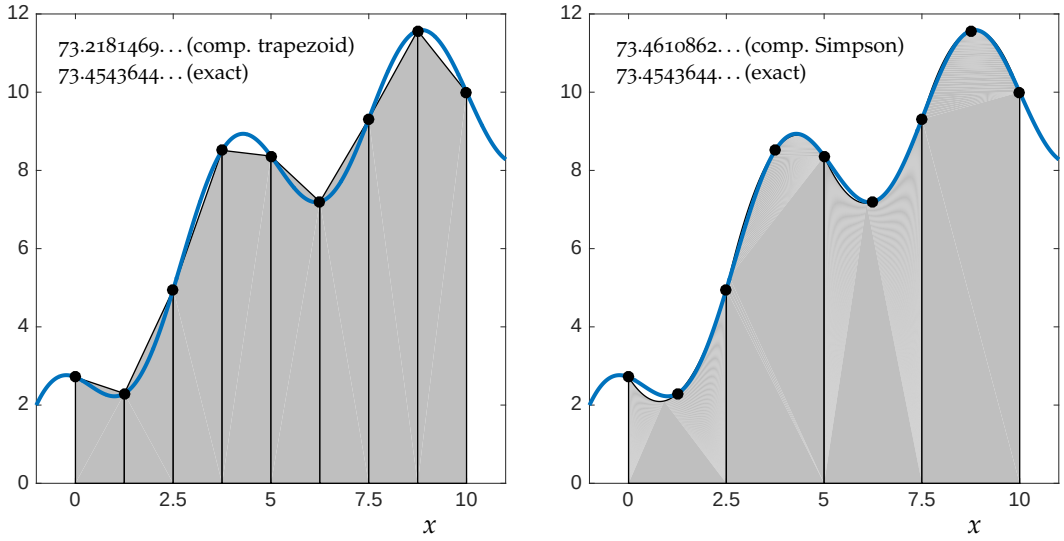


Figure 3.7: Composite trapezoid rule (left) and composite Simpson's rule (right).

3.2.4 *Adaptive Quadrature*

If f is continuous, we can attain arbitrarily high accuracy with composite rules by taking the spacing between function evaluations, h, to be sufficiently small. This might be necessary to resolve regions of rapid growth or oscillation in f. If such regions only make up a small proportion of the domain $[a, b]$, then uniformly reducing h over the entire interval will be unnecessarily expensive. One wants to concentrate function evaluations in the region where the function is the most ornery. Robust quadrature software adjusts the value of h locally to handle such regions. To learn more about such techniques, which are not foolproof, see W. Gander and W. Gautschi, "Adaptive quadrature—revisited," *BIT* 40 (2000) 84–101.

This paper criticizes the routines `quad` and `quad8` that were included in MATLAB version 5. In light of this analysis MATLAB improved its software, essentially incorporating the two routines suggested in this paper starting in version 6 as the routines `quad` (adaptive Simpson's rule) and `quadl` (an adaptive Gauss–Lobatto rule).

LECTURE 23: *Clenshaw–Curtis quadrature*

3.3 *Clenshaw–Curtis quadrature*

To get faster convergence for a fixed number of function evaluations, one might wish to increase the degree of the approximating polynomial further still, then integrate that high-degree polynomial. As evidenced in the discussion of polynomial interpolation in Section 1.6, the success of such an approach depends significantly on the choice of the interpolation points and the nature of the function being interpolated. For example, we would not expect the integral of a high degree polynomial interpolant to Runge's function $f(x) = (x^2+1)^{-1}$ over uniformly spaced points on $[-5, 5]$ to accurately approximate

$$\int_{-5}^{5} \frac{1}{x^2+1}\,\mathrm{d}x,$$

since the underlying interpolants fail to converge for this example.

However, recall that Theorem 2.6 ensures that the polynomials that interpolate f *at Chebyshev points* will converge, provided f is just a bit smooth. This suggests that the integrals of such interpolating polynomials will also be accurate as n increases, and indeed this is the case.

This procedure of *integrating interpolants at Chebyshev points* is known as *Clenshaw–Curtis quadrature*. If f is smooth, this method typically converges much faster than the composite trapezoid and Simpson's rules, which are only based on low-degree polynomial interpolants. In fact, the Clenshaw–Curtis method competes very well with the Gaussian quadrature schemes discussed in the next sections, although those Gaussian quadrature schemes have historically received much greater attention.

See L. N. Trefethen, 'Is Gauss Quadrature Better than Clenshaw–Curtis?', *SIAM Review* 50 (2008) 67–87.

Suppose we wish to integrate a function over $[-1, 1]$. Then Clenshaw–Curtis quadrature evaluates f at the $n+1$ Chebyshev points

$$x_j = \cos\left(\frac{j\pi}{n}\right), \quad j = 0, \ldots, n.$$

The Lagrange form of the polynomial interpolant at these points takes the form

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

where

$$\ell_j(x) = \prod_{k=0}^{n} \frac{x - x_k}{x_j - x_k}$$

are the usual Lagrange basis functions. Since the Clenshaw–Curtis quadrature rule will integrate the polynomial interpolant at the Chebyshev points, the rule will give

$$\int_{-1}^{1} f(x)\,\mathrm{d}x \approx \int_{-1}^{1} p_n(x)\,\mathrm{d}x = \sum_{j=0}^{n} f(x_j) \int_{-1}^{1} \ell_j(x)\,\mathrm{d}x.$$

Thus, defining the *weights* to be

$$w_j := \int_{-1}^{1} \ell_j(x)\,\mathrm{d}x,$$

the Clenshaw–Curtis quadrature rule takes the following compact form.

Clenshaw–Curtis rule:

$$\int_{-1}^{1} f(x)\,\mathrm{d}x \approx \sum_{j=0}^{n} w_j f(x_j),$$

where $x_j = \cos(j\pi/n)$ and $w_j = \int_{-1}^{1} \ell_j(x)\,\mathrm{d}x$.

Connecting interpolation at Chebyshev points to trigonometric interpolation leads to a convenient algorithm for computing the weights w_j using a fast Fourier transform, which is much more stable and convenient than integrating the Lagrange interpolating polynomials directly. We shall not go into details here. Interested readers can consult Trefethen's paper, 'Is Gauss Quadrature Better than Clenshaw–Curtis?' (2008) or his book *Spectral Methods in MATLAB* (2000).

LECTURE 24: *Gaussian quadrature rules: fundamentals*

3.4 Gaussian quadrature

It is clear that the trapezoid rule,

$$\frac{b-a}{2}\big(f(a)+f(b)\big),$$

exactly integrates linear polynomials, but not all quadratics. In fact, one can show that *no* quadrature rule of the form

$$w_a f(a) + w_b f(b)$$

will exactly integrate all quadratics over $[a, b]$, regardless of the choice of constants w_a and w_b. However, notice that a general quadrature rule with two points,

$$w_0 f(x_0) + w_1 f(x_1),$$

has four parameters (w_0, x_0, w_1, x_1). We might then hope that we could pick these four parameters in such a fashion that the quadrature rule is exact for a four-dimensional subspace of functions, $\mathcal{P}_3$. This section explores generalizations of this question.

3.4.1 A special 2-point rule

Suppose we consider a more general class of 2-point quadrature rules, where we do not initially fix the points at which the integrand f is evaluated:

$$I(f) = w_0 f(x_0) + w_1 f(x_1)$$

for unknowns *nodes* $x_0, x_1 \in [a, b]$ and *weights* w_0 and w_1. We wish to pick x_0, x_1, w_0, and w_1 so that the quadrature rule exactly integrates all polynomials of the largest degree possible. Since this quadrature rule is linear, it will suffice to check that it is exact on monomials. There are four unknowns; to get four equations, we will require $I(f)$ to exactly integrate 1, x, x^2, x^3.

$$\begin{aligned}
f(x)=1: &\quad \int_a^b 1\,\mathrm{d}x = I(1) &\implies &\quad b-a &= w_0 + w_1\\
f(x)=x: &\quad \int_a^b x\,\mathrm{d}x = I(x) &\implies &\quad \tfrac{1}{2}(b^2-a^2) &= w_0x_0 + w_1x_1\\
f(x)=x^2: &\quad \int_a^b x^2\,\mathrm{d}x = I(x^2) &\implies &\quad \tfrac{1}{3}(b^3-a^3) &= w_0x_0^2 + w_1x_1^2\\
f(x)=x^3: &\quad \int_a^b x^3\,\mathrm{d}x = I(x^3) &\implies &\quad \tfrac{1}{4}(b^4-a^4) &= w_0x_0^3 + w_1x_1^3
\end{aligned}$$

Three of these constraints are *nonlinear* equations of the unknowns x_0, x_1, w_0, and w_1: thus questions of existence and uniqueness of solutions becomes a bit more subtle than for the linear equations we so often encounter.

In this case, a solution *does* exist:

$$w_0 = w_1 = \tfrac{1}{2}(b-a),$$

$$x_0 = \tfrac{1}{2}(b+a) - \tfrac{\sqrt{3}}{6}(b-a), \qquad x_1 = \tfrac{1}{2}(b+a) + \tfrac{\sqrt{3}}{6}(b-a).$$

Notice that $x_0, x_1 \in [a,b]$: If this were not the case, we could not use these points as quadrature nodes, since f might not be defined outside $[a,b]$. When $[a,b] = [-1,1]$, the interpolation points are $\pm 1/\sqrt{3}$, giving the quadrature rule

$$I(f) = f(-1/\sqrt{3}) + f(1/\sqrt{3}).$$

3.4.2 *Generalization to higher degrees*

Emboldened by the success of this humble 2-point rule, we consider generalizations to higher degrees. If some two-point rule ($n+1$ integration nodes, for $n=1$) will exactly integrate all cubics ($3 = 2n+1$), one might anticipate the existence of rules based on $n+1$ points that exactly integrate all polynomials of degree $2n+1$, for general values of n. Toward this end, consider quadrature rules of the form

$$I_n(f) = \sum_{j=0}^{n} w_j f(x_j),$$

for which we will choose the nodes $\{x_j\}$ and weights $\{w_j\}$ (a total of $2n+2$ variables) to maximize the degree of polynomial that is integrated exactly.

The primary challenge is to find satisfactory quadrature nodes. Once these are found, the weights follow easily: in theory, one could obtain them by integrating the polynomial interpolant at the nodes, though better methods are available in practice. In particular, this procedure for assigning weights ensures, at a minimum, that $I_n(f)$ will exactly integrate all polynomials of degree n. This assumption will play a key role in the coming development.

Orthogonal polynomials, introduced in Section 2.5, play a central role in this exposition, and they suggest a generalization of the interpolatory quadrature procedures we have studied up to this point.

Let $\{\phi_j\}_{j=0}^{n+1}$ be a system of orthogonal polynomials with respect to the inner product

$$\langle f, g \rangle = \int_a^b f(x) g(x) w(x)\, \mathrm{d}x$$

for some weight function $w \in C(a,b)$ that is non-negative over (a,b) and takes the value of zero only on a set of measure zero.

This weight function plays an essential role in the discussion: it defines the inner product, and so it dictates what it means for two functions to be orthogonal. Change the weight function, and you will change the orthogonal polynomials.

Now we wish to construct an interpolatory quadrature rule for an integral that incorporates the weight function $w(x)$ in the integrand:

$$I_n(f) = \sum_{j=0}^{n} w_j f(x_j) \approx \int_a^b f(x)w(x)\,\mathrm{d}x.$$

In the Section 3.4.4 we shall see some useful examples of weight functions.

It is our aim to make $I_n(p)$ exact for all $p \in \mathcal{P}_{2n+1}$. First, we will show that any interpolatory quadrature rule I_n will at least be exact for the weighted integral of degree-n polynomials. Showing this is a simple modification of the argument made in Section 3.1 for unweighted integrals.

Given a set of distinct nodes $x_0, \ldots, x_n$, construct the polynomial interpolant to f at those nodes:

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

where $\ell_j(x)$ is the usual Lagrange basis function for polynomial interpolation. The interpolatory quadrature rule will exactly integrate the *weighted integral* of the interpolant p_n:

$$\begin{aligned}\int_a^b f(x)w(x)\,\mathrm{d}x \approx \int_a^b p_n(x)w(x)\,\mathrm{d}x &= \int_a^b \left(\sum_{j=0}^{n} f(x_j)\ell_j(x)\right) w(x)\,\mathrm{d}x \\ &= \sum_{j=0}^{n} f(x_j) \int_a^b \ell_j(x)w(x)\,\mathrm{d}x.\end{aligned}$$

Thus we define the *quadrature weights* for the *weighted integral* to be

$$w_j := \int_a^b \ell_j(x)w(x)\,\mathrm{d}x,$$

giving the rule

$$I_n(f) = \sum_{j=0}^{n} w_j f(x_j) \approx \int_a^b f(x)w(x)\,\mathrm{d}x.$$

Apply this rule to a degree-n polynomial, p. Since $p \in \mathcal{P}_n$, it is its own degree-n polynomial interpolant, so the integral of the interpolant delivers the exact weighted integral of p:

Note that the weight function $w(x)$ can include all sorts of nastiness, all of which is absorbed in the quadrature weights $w_0, \ldots, w_n$.

$$\int_a^b p(x)w(x)\,\mathrm{d}x = \sum_{j=0}^{n} w_j p(x_j) = I_n(p).$$

This is the case regardless of how the (distinct) nodes $x_0, \ldots, x_n$ were chosen. Now we seek a way to choose the nodes so that the quadature rule is exactly for a higher degree polynomials.

To begin, consider an arbitrary $p \in \mathcal{P}_{2n+1}$. Using polynomial division, we can always write

$$p(x) = \phi_{n+1}(x)q(x) + r(x)$$

for some $q, r \in \mathcal{P}_n$ that depend on p. Integrating this p, we obtain

$$\begin{aligned}\int_a^b p(x)w(x)\,\mathrm{d}x &= \int_a^b \phi_{n+1}(x)q(x)w(x)\,\mathrm{d}x + \int_a^b r(x)w(x)\,\mathrm{d}x \\ &= \langle \phi_{n+1}, q\rangle + \int_a^b r(x)w(x)\,\mathrm{d}x \\ &= \int_a^b r(x)w(x)\,\mathrm{d}x.\end{aligned}$$

The last step is a consequence that important basic fact, proved in Section 2.5, that the orthogonal polynomial ϕ_{n+1} is orthogonal to all $q \in \mathcal{P}_n$.

Now apply the quadrature rule to p, and attempt to pick the interpolation nodes $\{x_j\}$ to yield the value of the exact integral computed above. In particular,

$$\begin{aligned}I_n(p) = \sum_{j=0}^n w_j p(x_j) &= \sum_{j=0}^n w_j \phi_{n+1}(x_j)q(x_j) + \sum_{j=0}^n w_j r(x_j) \\ &= \sum_{j=0}^n w_j \phi_{n+1}(x_j)q(x_j) + \int_a^b r(x)w(x)\,\mathrm{d}x.\end{aligned}$$

This last statement is a consequence of the fact that $I_n(\cdot)$ will exactly integrate all $r \in \mathcal{P}_n$. This will be true regardless of our choice for the distinct nodes $\{x_j\} \subset [a, b]$. (Recall that the quadrature rule is constructed so that it exactly integrates a degree-n polynomial interpolant to the integrand, and in this case the integrand, r, is a degree n polynomial. Hence $I_n(r)$ will be exact.)

Notice that we can force agreement between $I_n(p)$ and $\int_a^b p(x)w(x)\,\mathrm{d}x$ provided

$$\sum_{j=0}^n w_j \phi_{n+1}(x_j)q(x_j) = 0.$$

We cannot make assumptions about $q \in \mathcal{P}_n$, as this polynomial will vary with the choice of p, but we can exploit properties of ϕ_{n+1}. Since ϕ_{n+1} has exact degree $n+1$ (recall this property of all orthogonal polynomials), it must have $n+1$ roots. If we choose the interpolation nodes $\{x_j\}$ to be the roots of ϕ_{n+1}, then $\sum_{j=0}^n w_j\, \phi_{n+1}(x_j)q(x_j) = 0$ as required, and we have a quadrature rule that is exact for all polynomials of degree $2n+1$.

Before we can declare victory, though, we must exercise some caution. Perhaps ϕ_{n+1} has repeated roots (so that the nodes $\{x_j\}$ are not distinct), or perhaps these roots lie at points in the complex plane

where f may not even be defined. Since we are integrating f over the interval $[a, b]$, it is crucial that ϕ_{n+1} has $n+1$ distinct roots in $[a, b]$. Fortunately, this is one of the many beautiful properties enjoyed by orthogonal polynomials.

Theorem 3.6 (Roots of Orthogonal Polynomials)**.**
Let $\{\phi_k\}_{k=0}^{n+1}$ be a system of orthogonal polynomials on $[a, b]$ with respect to the weight function $w(x)$. Then ϕ_k has k distinct real roots, $\{x_j^{(k)}\}_{j=1}^{k}$, with $x_j^{(k)} \in [a, b]$ for $j = 1, \ldots, k$.

Proof. The result is trivial for ϕ_0. Fix any $k \in \{1, \ldots, n+1\}$. Suppose that ϕ_k, a polynomial of exact degree k, changes sign at $j < k$ distinct roots $\{x_\ell^{(k)}\}_{\ell=1}^{j}$, in the interval $[a, b]$. Then define

$$q(x) = (x - x_1^{(k)})(x - x_2^{(k)}) \cdots (x - x_j^{(k)}) \in \mathcal{P}_j.$$

This function changes sign at exactly the same points as ϕ_k does on $[a, b]$. Thus, the product of these two functions, $\phi_k(x)q(x)$, *does not change sign* on $[a, b]$. See the illustration in Figure 3.8.

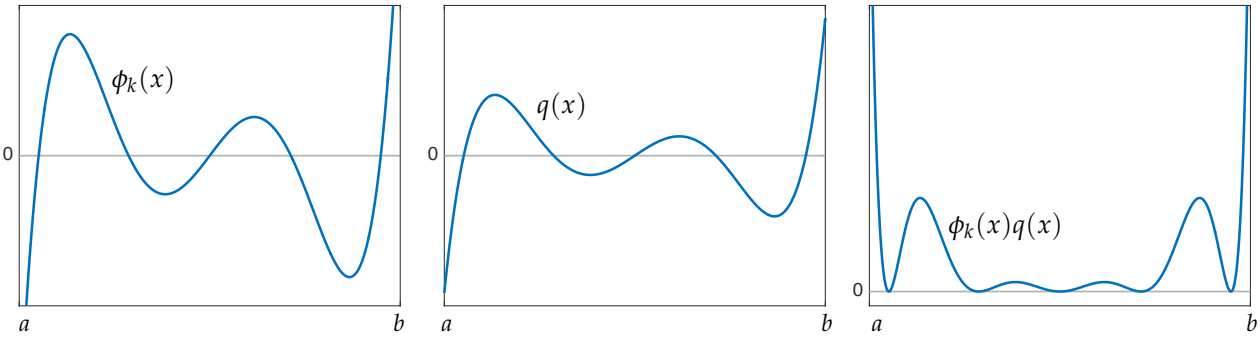


Figure 3.8: The functions ϕ_k, q, and $\phi_k q$ from the proof of Theorem 3.9.

As the weight function $w(x)$ is nonnegative on $[a, b]$, it must also be that $\phi_k q w$ does not change sign on $[a, b]$. However, the fact that $q \in \mathcal{P}_j$ for $j < k$ implies that

$$\int_a^b \phi_k(x)q(x)w(x)\,\mathrm{d}x = \langle \phi_k, q \rangle = 0,$$

since ϕ_k is orthogonal to all polynomials of degree $k-1$ or lower (Lemma 2.3). Thus, we conclude that the integral of some continuous nonzero function $\phi_k q w$ that never changes sign on $[a, b]$ must be zero. This is a contradiction, as the integral of such a function must always be positive. Thus, ϕ_k must have at least k distinct zeros in $[a, b]$. As ϕ_k is a polynomial of degree k, it can have no more than k zeros. ∎

We have arrived at *Gaussian quadrature rules*: Integrate the polynomial that interpolates f at the roots of the orthogonal polynomial ϕ_{n+1}. What are the weights $\{w_j\}$? Write the interpolant, p_n, in the Lagrange basis,

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

where the basis polynomials ℓ_j are defined as usual,

$$\ell_j(x) = \prod_{k=0, k\neq j}^{n} \frac{(x - x_k)}{(x_j - x_k)}. \tag{3.2}$$

Integrating this interpolant gives

$$I_n(f) = \int_a^b p_n(x)w(x)\,\mathrm{d}x = \int_a^b \sum_{j=0}^{n} f(x_j)\ell_j(x)w(x)\,\mathrm{d}x = \sum_{j=0}^{n} f(x_j)\int_a^b \ell_j(x)w(x)\,\mathrm{d}x,$$

revealing a formula for the quadrature weights:

$$w_j = \int_a^b \ell_j(x)w(x)\,\mathrm{d}x.$$

This construction proves the following result.

Theorem 3.7. Suppose $I_n(f)$ is the Gaussian quadrature rule

$$I_n(f) = \sum_{j=0}^{n} w_j f(x_j),$$

where the nodes $\{x_j\}_{j=0}^{n}$ are the $n+1$ roots of a degree-$(n+1)$ orthogonal polynomial on $[a, b]$ with weight function w, and $w_j = \int_a^b \ell_j(x)w(x)\,\mathrm{d}x$. Then

$$I_n(f) = \int_a^b f(x)w(x)\,\mathrm{d}x$$

for all polynomials f of degree $2n+1$.

As a side-effect of this high-degree exactness, we obtain an interesting new formula for the weights in Gaussian quadrature. Since the Lagrange basis polynomial ℓ_k is the product of n linear factors (see (3.2)), $\ell_k \in \mathcal{P}_n$, and

$$(\ell_k)^2 \in \mathcal{P}_{2n} \subseteq \mathcal{P}_{2n+1}.$$

Thus the Gaussian quadrature rule exactly integrates $(\ell_k)^2 w(x)$. We write

$$\begin{aligned}\int_a^b (\ell_k(x))^2 w(x)\mathrm{d}x &= \sum_{j=0}^{n} w_j(\ell_k(x_j))^2 \\ &= w_k(\ell_k(x_k))^2 = w_k,\end{aligned}$$

where we have used the fact that $\ell_k(x_j) = 0$ if $j \neq k$, and $\ell_k(x_k) = 1$. This leads to another formula for the Gaussian quadrature weights:

$$w_k = \int_a^b \ell_k(x) w(x)\,\mathrm{d}x = \int_a^b (\ell_k(x))^2 w(x)\,\mathrm{d}x. \tag{3.3}$$

This latter formula is more computationally appealing than the former, because it is more numerically reliable to integrate positive-valued integrands. This is a neat fact, but, as described in Section , there is a still-better way to compute these weights: by computing eigenvectors of a symmetric tridiagonal matrix.

One avoids floating point errors that can be introduced by adding quantities that are similar in magnitude but opposite in sign, known as *catastrophic cancellation*.

Of course, in many circumstances we are not simply integrating polynomials, but more complicated functions, so we want better insight about the method's performance than Theorem 3.7 provides. One can prove the following error bound.

See, e.g., Süli and Mayers, pp. 282–283.

Theorem 3.8. Suppose $f \in C^{2n+2}[a,b]$ and let $I_n(f)$ be the usual $(n+1)$-point Gaussian quadrature rule on $[a,b]$ with weight function $w(x)$ and nodes $\{x_j\}_{j=0}^n$. Then

$$\int_a^b f(x)w(x)\,\mathrm{d}x - I_n(f) = \frac{f^{(2n+2)}(\xi)}{(2n+2)!}\int_a^b \psi^2(x)w(x)\,\mathrm{d}x$$

for some $\xi \in [a,b]$ and $\psi(x) = \prod_{j=0}^n (x - x_j)$.

LECTURE 25: *Gaussian quadrature: nodes, weights; examples; extensions*

3.4.3 *Computing Gaussian quadrature nodes and weights*

When first approaching Gaussian quadrature, the complicated characterization of the nodes and weights might seem like a significant drawback. For example, if one approximates an integral with an $(n+1)$-point Gaussian quadrature rule and finds the accuracy insufficient, one must compute an entirely new set of nodes and weights for a larger n from scratch.

Many years ago, one would need to look up pre-computed nodes and weights for a given rule in book of mathematical tables, and one was thus limited to using values of n for which one could easily find tabulated values for the nodes and weights.

However, in a landmark paper of 1969, Gene Golub and John Welsch found a nice characterization of the nodes and weights in terms of a symmetric matrix eigenvalue problem. Given the existence of excellent algorithms for computing such eigenvalues, one can readily compute Gaussian quadrature nodes for arbitrary values of n. Some details of this derivation are left for a homework exercise, but we summarize the results here.

G. H. Golub and J. H. Welsch, "Calculation of Gauss Quadrature Rules," *Math. Comp.* 23 (1969) 221–230.

In general, such algorithms require $\mathcal{O}(n^3)$ operations to compute all eigenvalues and eigenvectors of an $n \times n$ matrix; for the modest values of n most common in practice (n in the tens or at most low hundreds), this expense is not onerous. Exploiting the structure of $\mathbf{J}_n$, this algorithm could be sped up to $\mathcal{O}(n^2)$.

One can show, via the discussion in Section 2.5, that, given values $\phi_{-1}(x) = 0$ and $\phi_0(x) = 1$ the subsequent orthogonal polynomials can be generated via the three-term recurrence relation

$$\phi_{k+1}(x) = x\,\phi_k(x) - \alpha_k \phi_k(x) - \beta_k \phi_{k-1}(x). \tag{3.4}$$

The values of $\alpha_0, \ldots, \alpha_n$ and $\beta_1, \ldots, \beta_n$ follow from the Gram–Schmidt process used in Section 2.5. Later we will also need the definition

$$\beta_0 := \langle 1, 1 \rangle = \int_a^b w(x)\,\mathrm{d}x.$$

A good source for the values of $\{\alpha_k\}$ and $\{\beta_k\}$ is Table 1.1 in Walter Gautschi's *Orthogonal Polynomials*, Oxford University Press, 2004.

Given a fixed value of n, collect the coefficients $\{\alpha_k\}_{k=0}^n$ and $\{\beta_k\}_{k=1}^n$ and use them to populate the *Jacobi matrix*

$$\mathbf{J}_n = \begin{bmatrix} \alpha_0 & \sqrt{\beta_1} & & & \\ \sqrt{\beta_1} & \alpha_1 & \sqrt{\beta_2} & & \\ & \sqrt{\beta_2} & \ddots & \ddots & \\ & & \ddots & \alpha_{n-1} & \sqrt{\beta_n} \\ & & & \sqrt{\beta_n} & \alpha_n \end{bmatrix}. \tag{3.5}$$

The following theorem (whose proof is left for as a homework exercise) gives a ready way to compute the roots of ϕ_{n+1}.

Theorem 3.9. Let $\{\phi_j\}_{j=0}^{n+1}$ denote a sequence of monic orthogonal polynomials generated by the recurrence relation (3.4). Then λ is a root of ϕ_{n+1} if and only if λ is an eigenvalue of $\mathbf{J}_n$ with corresponding eigenvector

$$\mathbf{v}(\lambda) = \begin{bmatrix} \phi_0(\lambda) \\ \phi_1(\lambda)/\sqrt{\beta_1} \\ \phi_2(\lambda)/\sqrt{\beta_1\beta_2} \\ \vdots \\ \phi_n(\lambda)/\sqrt{\beta_1\cdots\beta_n} \end{bmatrix}. \tag{3.6}$$

Theorem 3.9 thus gives a convenient way to compute the nodes of a Gaussian quadrature rule. Given the nodes, one could compute the weights using either of the formulas (3.3) involving Lagrange basis functions. However, Golub and Welsch proved that these same weights could be extracted from the eigenvectors of the Jacobi matrix $\mathbf{J}_n$. Label the eigenvalues of $\mathbf{J}_n$ as

$$\lambda_0 < \lambda_1 < \cdots < \lambda_n$$

and the corresponding eigenvectors, given by the formula (3.6), as

$$\mathbf{v}_0 = \mathbf{v}(\lambda_0), \quad \mathbf{v}_1 = \mathbf{v}(\lambda_1), \quad \ldots, \quad \mathbf{v}_n = \mathbf{v}(\lambda_n).$$

Then, with a bit of work, one can show that the weights for $n+1$-point Gaussian quadrature can be computed as

$$w_j = \beta_0 \frac{1}{\|\mathbf{v}_j\|_2^2}, \tag{3.7}$$

Note: assumes $(\mathbf{v}_j)_1 = \phi_0(\lambda_j) = 1$.

where $\|\cdot\|_2$ is the vector 2-norm.

The formula (3.7) relies on the specific form of the eigenvector in (3.6). If the eigenvector is normalized differently (e.g., MATLAB's `eigs` routine gives unit eigenvectors, $\|\mathbf{v}_j\|_2 = 1$, then one should use the general formula

$$w_j = \beta_0 \frac{(\mathbf{v}_j)_1^2}{\|\mathbf{v}_j\|_2^2}, \tag{3.8}$$

where $(\mathbf{v}_j)_1$ is the first entry in the eigenvector $\mathbf{v}_j$.

3.4.4 *Examples of Gaussian Quadrature*

Let us examine four well-known Gaussian quadrature rules.

method	*interval*, (a, b)	*weight*, $w(x)$
Gauss–Legendre	$(-1, 1)$	1
Gauss–Chebyshev	$(-1, 1)$	$\dfrac{1}{\sqrt{1-x^2}}$
Gauss–Laguerre	$(0, \infty)$	e^{-x}
Gauss–Hermite	$(-\infty, \infty)$	e^{-x^2}

The weight function plays an essential role here. For example, a Gauss–Chebyshev quadrature rule with $n+1 = 5$ points will exactly integrate polynomials of degree $2n+1 = 9$ *times the weight function*. Thus this rule *will* exactly integrate

$$\int_{-1}^{1} \frac{x^9}{\sqrt{1-x^2}}\,\mathrm{d}x,$$

but it *will not* exactly integrate

$$\int_{-1}^{1} x^{10}\,\mathrm{d}x.$$

This is a subtle point that many students overlook when first learning about Gaussian quadrature.

Example 3.2. Gauss–Legendre Quadrature
When numerical analysts speak of "Gaussian quadrature" without further qualification, they typically mean Gauss–Legendre quadrature, i.e., quadrature with the weight function $w(x) = 1$ (perhaps over a transformed domain (a, b); see Section 3.4.5.) As discussed in Section 2.5.1, the orthogonal polynomials for this interval and weight are called *Legendre polynomials*. To construct a Gaussian quadrature rule with $n+1$ points, determine the roots of the degree-$(n+1)$ Legendre polynomial, then find the associated weights.

First consider $n = 1$. The quadratic Legendre polynomial is

$$\phi_2(x) = x^2 - 1/3,$$

and from this polynomial one can derive the 2-point quadrature rule that is exact for cubic polynomials, with roots $\pm 1/\sqrt{3}$. This agrees with the special 2-point rule derived in Section 3.4.1. The values for the weights follow simply, $w_0 = w_1 = 1$, giving the 2-point Gauss–Legendre rule

$$I_n(f) = f(-1/\sqrt{3}) + f(1/\sqrt{3})$$

that exactly integrates polynomials of degree $2n+1 = 3$, i.e., all cubics.

Recall that Simpson's rule also exactly integrates cubics, but it requires *three* f evaluations, rather than the two f evaluations required of this rule.

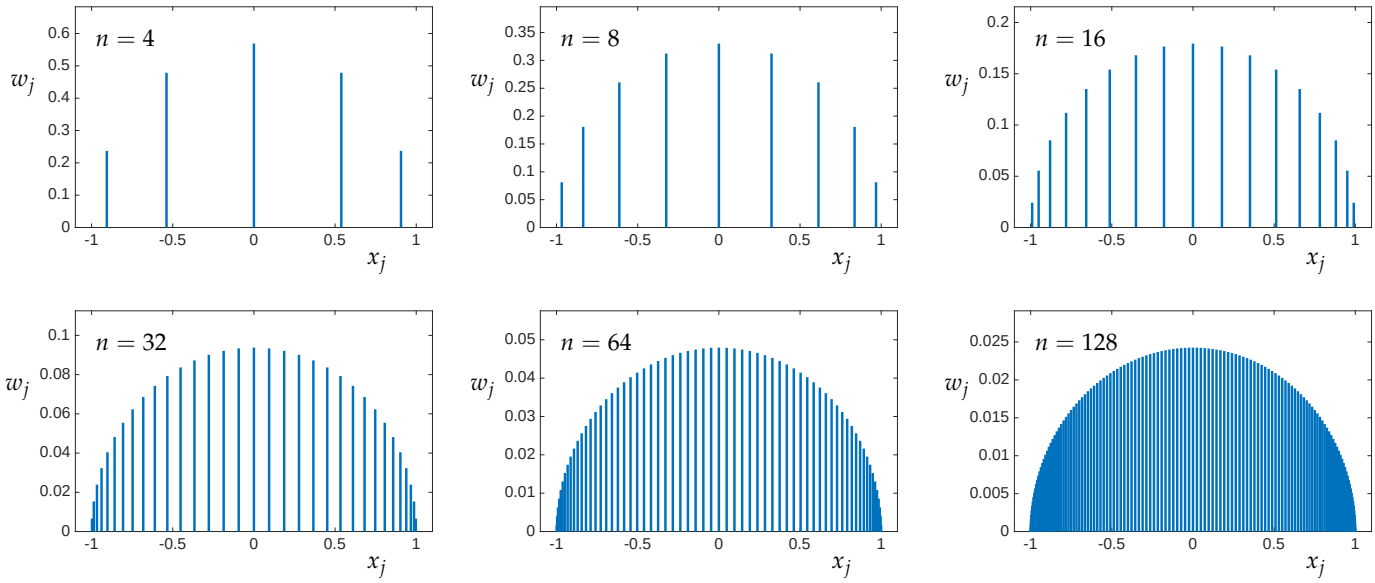


Figure 3.9: Nodes and weights of Gauss–Legendre quadrature, for various values of n. In each case, the location of the vertical line indicates x_j, while the height of the line shows w_j.

For Gauss–Legendre quadrature rules based on larger numbers of points, we can compute the nodes and weights using the symmetric eigenvalue formulation discussed in Section 3.4.3. For this weight, one can show

$$\begin{aligned} \alpha_k &= 0, && k = 0, 1, \ldots; \\ \beta_0 &= 2; \\ \beta_k &= \frac{k^2}{4k^2 - 1}, && k = 1, 2, 3, \ldots. \end{aligned}$$

Figure 3.9 shows the nodes and weights for six values of n, as computed via the eigenvalue problem. Notice that the points are not uniformly spaced, but are slightly more dense at the ends of the interval. Moreover, the weights are smaller at these ends of the interval.

The table below shows nodes and weights for $n = 4$, as computed in MATLAB.

j	*nodes*, x_j	*weights*, w_j
0	−0.906179845938664	0.236926885056189
1	−0.538469310105683	0.478628670499366
2	0.000000000000000	0.568888888888889
3	0.538469310105683	0.478628670499367
4	0.906179845938664	0.236926885056189

Example 3.3. Gauss–Chebyshev quadrature
Another popular class of Gaussian quadrature rules use as their nodes the roots of the Chebyshev polynomials. The standard degree-

k Chebyshev polynomial is defined as

$$T_k(x) = \cos(k \cos^{-1} x),$$

which can be generated by the recurrence relation

$$T_{k+1}(x) = 2xT_k(x) - T_{k-1}(x).$$

with $T_0(x) = 1$ and $T_1(x) = x$. These Chebyshev polynomials are orthogonal on $(-1, 1)$ with respect to the weight function

$$w(x) = \frac{1}{\sqrt{1-x^2}}.$$

The degree-$(n+1)$ Chebyshev polynomial has the roots

$$x_j = \cos\Big(\frac{(j+1/2)\pi}{n+1}\Big), \qquad j = 0, \ldots, n.$$

In this case all the weights work out to be identical; one can show

$$w_j = \frac{\pi}{n+1}$$

for all $j = 0, \ldots n$. Figure 3.10 shows these nodes and weights. One can also define the *monic* Chebyshev polynomials according to the recurrence (3.4) with

See Süli and Mayers, Problem 10.4 for a sketch of a proof.

$$\begin{aligned}
&\alpha_k = 0, && k = 0, 1, \ldots; \\
&\beta_0 = \pi; \\
&\beta_1 = 1/2; \\
&\beta_k = 1/4, && k = 2, 2, 3, \ldots.
\end{aligned}$$

The resulting polynomials are scaled versions of the usual Chebyshev polynomials $T_{k+1}(x)$, and thus have the same roots.

Again, we emphasize that the weight function plays a crucial role: the Gauss–Chebyshev rule based on $n+1$ interpolation nodes will exactly compute integrals of the form

$$\int_{-1}^{1} \frac{p(x)}{\sqrt{1-x^2}}\,\mathrm{d}x$$

for all $p \in \mathcal{P}_{2n+1}$. For a general integral

$$\int_{-1}^{1} \frac{f(x)}{\sqrt{1-x^2}}\,\mathrm{d}x.$$

the quadrature rule should be implemented as

$$I_n(f) = \sum_{j=0}^{n} w_j f(x_j);$$

Note that the $1/\sqrt{1-x^2}$ component of the integrand is not evaluated here; its influence has already been incorporated into the weights $\{w_j\}$.

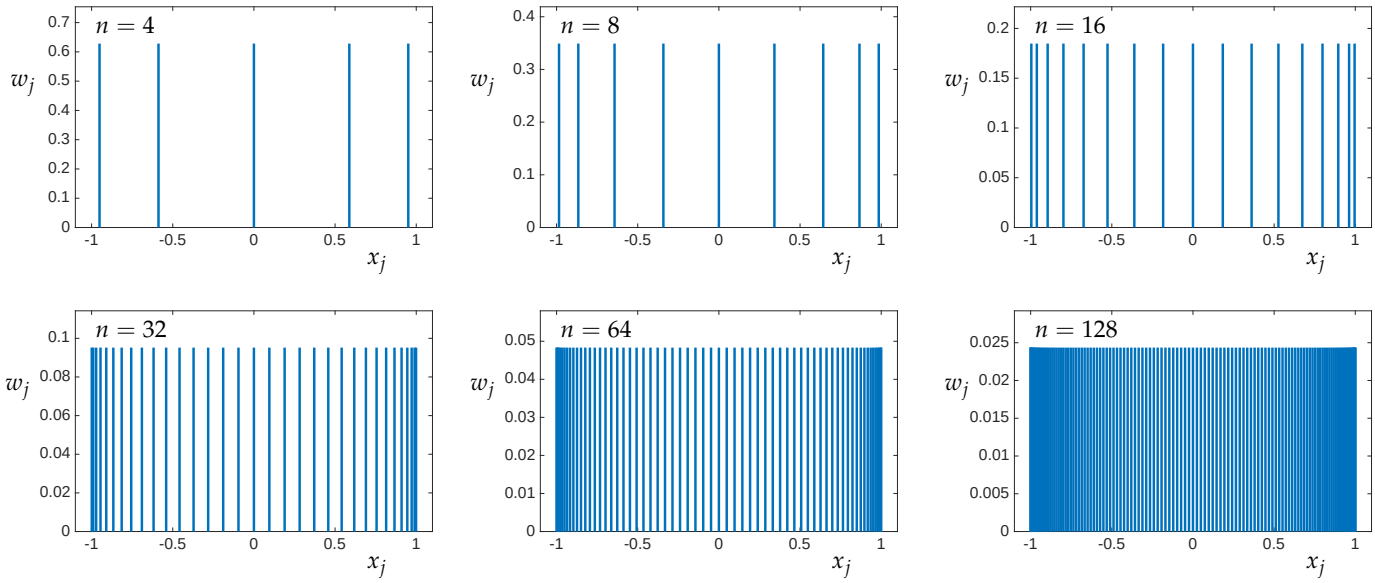


Figure 3.10: Nodes and weights of Gauss–Chebyshev quadrature, for various values of n. In each case, the location of the vertical line indicates x_j, while the height of the line shows w_j. The nodes, roots of Chebyshev polynomials, cluster toward the end of the intervals; the weights are the same for all the nodes, $w_j = \pi/(n+1)$.

The Chebyshev weight function $w(x) = 1/(1-x^2)$ blows up at ± 1, so if the integrand f does not balance this growth, adaptive Newton–Cotes rules will likely have to place many interpolation nodes near these singularities to achieve decent accuracy, while Gauss–Chebyshev quadrature has no problems. Moreover, in this important case, the nodes and weights are trivial to compute, thus allaying the need to solve the eigenvalue problem.

It is worth pointing out that Gauss–Chebyshev quadrature is quite different than Clenshaw–Curtis quadrature. Though both use Chebyshev points as interpolation nodes, only Gauss–Chebyshev incorporates the weight function $w(x) = (1-x^2)^{-1/2}$ in the weights $\{w_j\}$. Thus Clenshaw–Curtis is more appropriately compared to Gauss–Legendre quadrature. Since the Clenshaw–Curtis method is not a Gaussian quadrature formula, it will generally be exact only for all $p \in \mathcal{P}_n$, rather than all $p \in \mathcal{P}_{2n+1}$.

Example 3.4. Gauss–Laguerre quadrature
The Laguerre polynomials form a set of orthogonal polynomials over $(0, \infty)$ with the weight function $w(x) = \mathrm{e}^{-x}$. The accompanying quadrature rule approximates integrals of the form

$$\int_0^\infty f(x)\mathrm{e}^{-x}\,\mathrm{d}x.$$

The recurrence (3.4) uses the coefficients

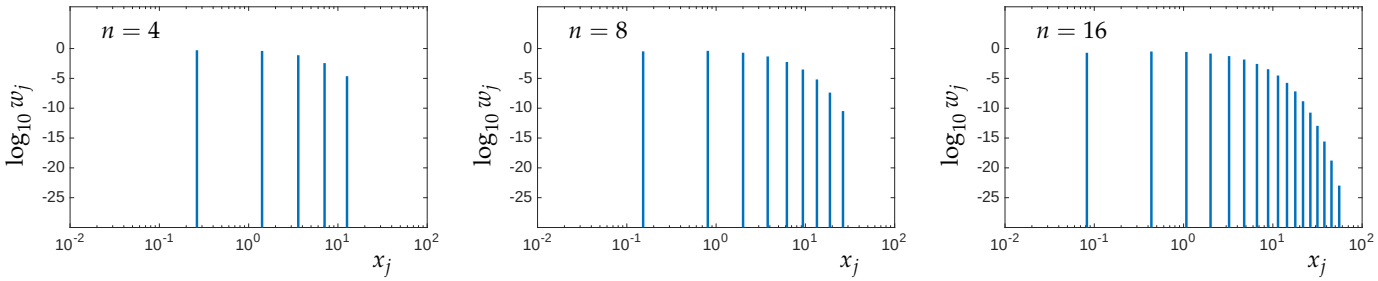


Figure 3.11: Nodes and weights of Gauss–Laguerre quadrature, for various values of n. In each case, the location of the vertical line indicates x_j, while the height of the line shows $\log_{10} w_j$. Note that the horizontal axis is scaled logarithmically. As n increases the quadrature rule includes larger and larger nodes to account for the infinite domain of integration; however, the weights are exceptionally small for the larger nodes. For example for $n = 16$, $w_{16} \approx 10^{-23}$.

$$\begin{aligned} \alpha_k &= 2k+1, \qquad && k = 0, 1, \ldots; \\ \beta_0 &= 1; \\ \beta_k &= k^2, && k = 1, 2, 3, \ldots. \end{aligned}$$

Figure 3.11 shows the nodes and weights for several values of n. Since the domain of integration $(0, \infty)$ is infinite, the quadrature nodes x_j get larger and larger. As the nodes get larger, the corresponding weights decay rapidly. When $w_j < 10^{-15}$, it becomes difficult to reliably compute the weights by solving the Jacobi matrix eigenvalue problem. To get the small weights given here, we have used Chebfun's `lagpts` routine routine, which uses a more efficient algorithm of Glaser, Liu, and Rokhlin (2007).

Example 3.5. Gauss–Hermite quadrature
The Hermite polynomials are orthogonal polynomials over $(-\infty, \infty)$ with the weight function $w(x) = \mathrm{e}^{-x^2}$. This quadrature rule approximates integrals of the form

$$\int_{-\infty}^{\infty} f(x) \mathrm{e}^{-x^2} \, \mathrm{d}x.$$

The Hermite polynomials can be generated using the recurrence (3.4) with coefficients

$$\begin{aligned} \alpha_k &= 0, \qquad && k = 0, 1, \ldots; \\ \beta_0 &= \sqrt{\pi}; \\ \beta_k &= k/2, && k = 1, 2, 3, \ldots. \end{aligned}$$

Figure 3.12 shows nodes and weights for various values of n. Though the interval of integration is infinite, the nodes do not grow as rapidly as for Gauss–Laguerre quadrature, since the Hermite weight $w(x) = \mathrm{e}^{-x^2}$ decays more rapidly than the Laguerre weight $w(x) = \mathrm{e}^{-x}$. (Again, the nodes and weights in the figure were computed with Chebfun's implementation of the Glaser, Liu, and Rokhlin algorithm.)

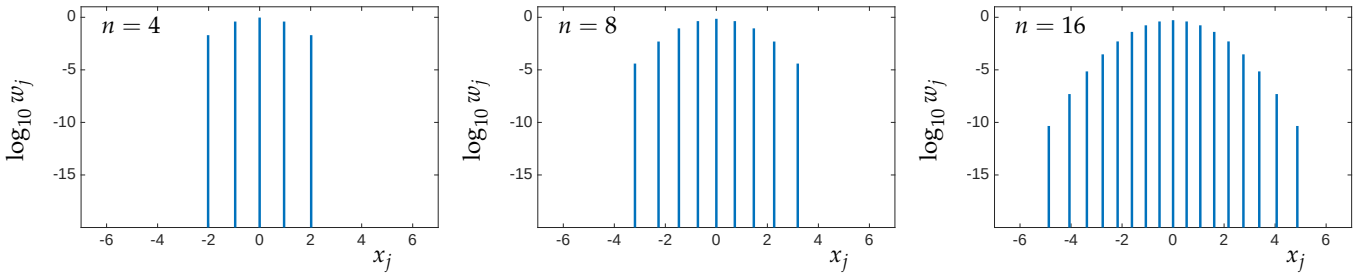


Figure 3.12: Nodes and weights of Gauss–Hermite quadrature, for various values of n. In each case, the location of the vertical line indicates x_j, while the height of the line shows $\log_{10} w_j$. Contrast this plot to Figure 3.11. Though the domain of integration is infinite in both cases, the weight function here, e^{-x^2}, decays much more rapidly than e^{-x}, explaining why the largest nodes are smaller than seen in Figure 3.11 for Gauss–Laguerre quadrature.

3.4.5 *Changing variables to transform domains*

One notable drawback of Gaussian quadrature is the need to precompute (or look up) the requisite weights and nodes. If one has a quadrature rule for the interval $[c, d]$, and wishes to adapt it to the interval $[a, b]$, there is a simple change of variables procedure to eliminate the need to recompute the nodes and weights from scratch. Let τ be a linear transformation taking $[c, d]$ to $[a, b]$,

$$\tau(x) = a + \left(\frac{b-a}{d-c}\right)(x-c)$$

with inverse $\tau^{-1} : [a, b] \to [c, d]$,

$$\tau^{-1}(y) = c + \left(\frac{d-c}{b-a}\right)(y-a).$$

Then we have

$$\begin{aligned}\int_a^b f(x)w(x)\,\mathrm{d}x &= \int_{\tau^{-1}(a)}^{\tau^{-1}(b)} f(\tau(x))w(\tau(x))\tau'(x)\,\mathrm{d}x \\ &= \left(\frac{b-a}{d-c}\right)\int_c^d f(\tau(x))w(\tau(x))\,\mathrm{d}x.\end{aligned}$$

The quadrature rule for $[a, b]$ takes the form

$$\widehat{I}(f) = \sum_{j=0}^{n} \widehat{w}_j f(\widehat{x}_j),$$

for

$$\widehat{w}_j = \left(\frac{b-a}{d-c}\right) w_j, \qquad \widehat{x}_j = \tau^{-1}(x_j),$$

where $\{x_j\}_{j=0}^n$ and $\{w_j\}_{j=0}^n$ are the nodes and weights for the quadrature rule on $[c, d]$.

Be sure to note how this change of variables alters the weight function. The transformed rule will now have a weight function

$$w(\tau(x)) = w(a + (b-a)(x-c)/(d-c)),$$

not simply $w(x)$. To make this concrete, consider Gauss–Chebyshev quadrature, which uses the weight function $w(x) = (1-x^2)^{-1/2}$ on

$[-1,1]$. If one wishes to integrate, for example, $\int_0^1 x(1-x^2)^{-1/2}\,\mathrm{d}x$, it *is not* sufficient just to use the change of variables formula described here. To compute the desired integral, one would have to adjust the nodes and weights to accommodate $w(x) = (1-x^2)^{-1/2}$ on $[0,1]$.

Composite rules Employing this change of variables technique, it is simple to devise a method for decomposing the interval of integration into smaller regions, over which Gauss quadrature rules can be applied. (The most straightforward application is to adjust the Gauss–Legendre quadrature rule, which avoids complications induced by the weight function, since $w(x) = 1$ in this case.) Such techniques can be used to develop Gaussian-based adaptive quadrature rules.

3.4.6 Gauss–Radau and Gauss-Lobatto quadrature

Some applications make it necessary or convenient to force one or both of the end points of the interval of integration to be among the quadrature points. Such methods are known as Gauss–Radau and Gauss–Lobatto quadrature rules, respectively; rules based on $n+1$ interpolation points exactly integrate all polynomials in $\mathcal{P}_{2n}$ or $\mathcal{P}_{2n-1}$: each quadrature node that we fix decreases the optimal order by one.

LECTURE 26: *Richardson extrapolation*

3.5 Richardson extrapolation, Romberg integration

Throughout numerical analysis, one encounters procedures that apply some simple approximation (e.g., linear interpolation) to construct some equally simple algorithm (e.g., differentiate the interpolant to get a finite difference formula (Section 1.7), integrate the interpolant to get the trapezoid rule (Section 3.2). An unfortunate consequence is that such approximations often converge slowly, with errors decaying only like h or h^2, where h is some discretization parameter (e.g., the spacing between interpolation points).

In the case of integration, you might prefer using a higher order method, like Clenshaw–Curtis or Gaussian quadrature. What we talk about here is an alternative to such approaches.

In this lecture we describe a remarkable, fundamental tool of classical numerical analysis. Like alchemists who sought to convert lead into gold, so we will take a sequence of slowly convergent data and extract from it a highly accurate estimate of our solution. This procedure is *Richardson extrapolation*, an essential but easily overlooked technique that should be part of every numerical analyst's toolbox. When applied to quadrature rules, the procedure is called *Romberg integration*.

We begin in a general setting: Suppose we seek some abstract quantity, $\xi \in \mathbb{R}$, which could be the value of a definite integral, a derivative, the solution to a differential equation at a certain point, or something else entirely. Further suppose we cannot compute ξ exactly; we can only access numerical approximations to it, generated by some function (an algorithm) Φ that depends upon a mesh parameter h. We compute $\Phi(h)$ for several values of h, expecting that $\Phi(h) \to \Phi(0) = \xi$ as $h \to 0$. To obtain good accuracy, one naturally seeks to evaluate Φ with increasingly smaller values of h. There are two reasons not to do so:

- Often Φ becomes increasingly expensive to evaluate as h shrinks;
- The numerical accuracy with which we can evaluate Φ may deteriorate as h gets small, due to rounding errors in floating point arithmetic. (For an example of the latter, try computing estimates of $f'(\alpha)$ using the formula $f'(\alpha) \approx (f(\alpha + h) - f(\alpha))/h$ as $h \to 0$.)

For example, computing $\Phi(h/2)$ often requires at least twice as much work as $\Phi(h)$. In some cases, $\Phi(h/2)$ could require 4, or even 8, times as much work at $\Phi(h)$, i.e., the expense of Φ could grow like $1/h$ or $1/h^2$ or $1/h^3$, etc.

Assume that Φ is infinitely continuously differentiable *as a function of* h, thus allowing us to expand $\Phi(h)$ in the Taylor series

$$\Phi(h) = \Phi(0) + h\Phi'(0) + \tfrac{1}{2}h^2\Phi''(0) + \tfrac{1}{6}h^3\Phi'''(0) + \cdots .$$

The derivatives here may seem to complicate matters (e.g., what are the derivatives of a quadrature rule with respect to h?), but we shall not need to compute them: they key is that the function Φ behaves

smoothly in h. Recalling that $\Phi(0) = \xi$, we can rewrite the Taylor series for $\Phi(h)$ as

$$\Phi(h) = \xi + c_1 h + c_2 h^2 + c_3 h^3 + \cdots$$

for some constants $\{c_j\}_{j=1}^{\infty}$. (For example, $c_1 = \Phi'(0)$.)

This expansion implies that taking $\Phi(h)$ as an approximation for ξ incurs an $\mathcal{O}(h)$ error. Halving the parameter h should roughly halve the error, according to the expansion

$$\Phi(h/2) = \xi + c_1 \tfrac{1}{2} h + c_2 \tfrac{1}{4} h^2 + c_3 \tfrac{1}{8} h^3 + \cdots .$$

Here comes the trick that is key to the whole lecture: Combine the expansions for $\Phi(h)$ and $\Phi(h/2)$ in such a way that eliminates the $\mathcal{O}(h)$ term. In particular, define

$$\begin{aligned}
\Psi(h) &:= 2\Phi(h/2) - \Phi(h) \\
&= 2\Big(\xi + c_1 \tfrac{1}{2} h + c_2 \tfrac{1}{4} h^2 + c_3 \tfrac{1}{8} h^3 + \cdots\Big) \\
&\qquad - \Big(\xi + c_1 h + c_2 h^2 + c_3 h^3 + \cdots\Big) \\
&= \xi - c_2 \tfrac{1}{2} h^2 - c_3 \tfrac{3}{4} h^3 + \cdots .
\end{aligned}$$

Thus, $\Psi(h)$ also approximates $\xi = \Psi(0) = \Phi(0)$, but with an $\mathcal{O}(h^2)$ error, rather than the $\mathcal{O}(h)$ error that pollutes $\Phi(h)$. For small h, this $\mathcal{O}(h^2)$ approximation will be considerably more accurate.

Why stop with $\Psi(h)$? Repeat the procedure, combining $\Psi(h)$ and $\Psi(h/2)$ to eliminate the $\mathcal{O}(h^2)$ term. Since

$$\Psi(h/2) = \xi - c_2 \tfrac{1}{8} h^2 - c_3 \tfrac{3}{32} h^3 + \cdots ,$$

we have

$$\Theta(h) := \frac{4\Psi(h/2) - \Psi(h)}{3} = \xi + c_3 \tfrac{1}{8} h^3 + \cdots .$$

To compute $\Theta(h)$, we must have access to both $\Psi(h)$ and $\Psi(h/2)$. These, in turn, require $\Phi(h)$, $\Phi(h/2)$, and $\Phi(h/4)$. In many cases, Φ becomes increasingly expensive to compute as the parameter h is reduced. Thus there is some practical limit to how small we can take h when evaluating $\Phi(h)$.

One could continue this procedure repeatedly, each time improving the accuracy by one order, at the cost of one additional Φ computation with a smaller h. To facilitate generalization and to avoid a further tangle of Greek characters, we adopt a new notation: Define

$$\begin{aligned}
R(j,0) &:= \Phi(h/2^j), && j \geq 0; \\
R(j,k) &:= \frac{2^k R(j,k-1) - R(j-1,k-1)}{2^k - 1}, && j \geq k > 0.
\end{aligned}$$

For the sake of clarity let us discuss a concrete case, elaborated upon in Example 3.6 below. Suppose we wish to compute $\xi = f'(\alpha)$ using the finite difference formula

$$\Phi(h) = \frac{f(\alpha + h) - f(\alpha)}{h}.$$

The quotient rule gives

$$\Phi'(h) = \frac{f(\alpha) - f(\alpha + h)}{h^2} + \frac{f'(\alpha + h)}{h},$$

which will depend smoothly on h provided f is smooth near α. In particular, a Taylor expansion for f gives

$$f(\alpha + h) = f(\alpha) + h f'(\alpha) + \tfrac{1}{2} h^2 f''(\alpha) + \tfrac{1}{6} h^3 f'''(\eta)$$

for some $\eta \in [\alpha, \alpha + h]$. Substitute this formula into the equation for $\Phi'(h)$ and simplify to get

$$\Phi'(h) = \frac{f'(\alpha + h) - f'(\alpha)}{h} - \tfrac{1}{2} f''(\alpha) - \tfrac{1}{6} h f'''(\eta).$$

Now this expression leads to a clean formula for the first coefficient of the Taylor series for $\Phi(h)$:

$$c_1 = \Phi'(0) = \lim_{h \to 0} \Phi'(h) = \tfrac{1}{2} f''(\alpha).$$

The moral of the example: while it might seem strange to take a Taylor series of the "algorithm" Φ, the quantities involved often have a very natural interpretation in terms of the underlying problem at hand.

Thus: $R(0,0) = \Phi(h)$, $R(1,0) = \Phi(h/2)$, and $R(1,1) = \Psi(h)$. This procedure is called *Richardson extrapolation* after the British applied mathematician Lewis Fry Richardson, a pioneer of the numerical solution of partial differential equations, weather modeling, and mathematical models in political science. The numbers $R(j,k)$ are arranged in a triangular *extrapolation table*:

$$\begin{array}{llllll}
R(0,0) & & & & & \\
R(1,0) & R(1,1) & & & & \\
R(2,0) & R(2,1) & R(2,2) & & & \\
R(3,0) & R(3,1) & R(3,2) & R(3,3) & & \\
\vdots & \vdots & \vdots & \vdots & \ddots & \\
\cdots & \cdots & \cdots & \cdots & \cdots & . \\
\uparrow & \uparrow & \uparrow & \uparrow & & \\
\mathcal{O}(h) & \mathcal{O}(h^2) & \mathcal{O}(h^3) & \mathcal{O}(h^4) & &
\end{array}$$

To compute any given element in the table, one must first determine entries above and to the left. Note that only the first column will require significant work; the subsequent columns follow from easy arithmetic. The theory suggests that the bottom-right element in the table will be the most accurate approximation to ξ. Indeed this bottom-right entry will generally be the most accurate, provided the assumption that Φ is infinitely continuously differentiable holds. When floating point roundoff errors spoil what otherwise would have been an infinitely continuously differentiable procedure, the bottom-right entry will suffer acutely from this pollution. Such errors will be apparent in the forthcoming example.

Example 3.6 (Finite difference approximation of the first derivative)**.** We seek $\xi = f'(\alpha)$ for some function continuously differentiable function f. Recall from Section 1.7 the simple finite difference approximation to the first derivative that follows from differentiating the linear interpolant to f through the points $x = \alpha$ and $x = \alpha + h$:

$$f'(\alpha) \approx \frac{f(\alpha + h) - f(\alpha)}{h}.$$

In fact, in Theorem 1.6 we quantified the error to be $\mathcal{O}(h)$ as $h \to 0$:

$$f'(\alpha) = \frac{f(\alpha + h) - f(\alpha)}{h} + \mathcal{O}(h).$$

Thus we define

$$\Phi(h) = \frac{f(\alpha + h) - f(\alpha)}{h}.$$

As a simple test problem, take $f(x) = \mathrm{e}^x$. We will use Φ and Richardson extrapolation to approximate $f'(1) = \mathrm{e} = 2.7182818284\ldots$.

The simple finite difference method produces crude answers:

h	$\Phi(h)$	error
1	4.670774270	1.95249×10^{0}
1/2	3.526814484	8.08533×10^{-1}
1/4	3.088244516	3.69963×10^{-1}
1/8	2.895480164	1.77198×10^{-1}
1/16	2.805025851	8.67440×10^{-2}
1/32	2.761200889	4.29191×10^{-2}
1/64	2.739629446	2.13476×10^{-2}
1/128	2.728927823	1.06460×10^{-2}
1/256	2.723597892	5.31606×10^{-3}
1/512	2.720938130	2.65630×10^{-3}

Even with $h = 1/512 = 0.00195\ldots$ we fail to approximate $f'(1)$ to even three correct digits. As we take h smaller and smaller, finite precision arithmetic eventually causes unacceptable errors; Figure 3.13 shows the error in $\Phi(h)$ as $h \to 0$. (The red line shows what perfect $\mathcal{O}(h)$ convergence would look like.)

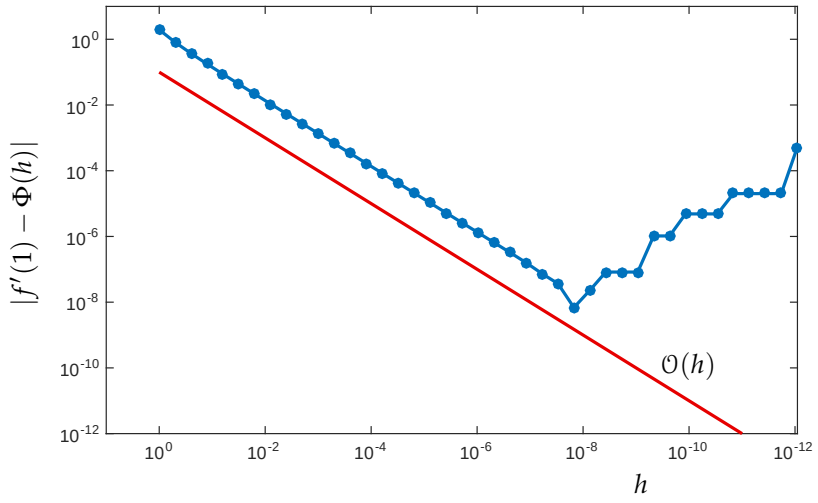


Figure 3.13: Linear convergence of the estimate $\Phi(h)$ to $f'(1)$ (blue line). As h gets small, rounding errors spoil the $\mathcal{O}(h)$ convergence (red line). An accuracy of about 10^{-8} seems to be the best we can do for this method and this problem.

A few steps of Richardson extrapolation on the data in the table above reveals greatly improved solutions, five correct digits in $R(4,4)$:

j	$R(j,0)$	$R(j,1)$	$R(j,2)$	$R(j,3)$	$R(j,4)$
0	4.67077427047160				
1	3.52681448375804	2.38285469704447			
2	3.08824451601118	2.64967454826433	2.73861449867095		
3	2.89548016367188	2.70271581133258	2.72039623235534	2.71779362288168	
4	2.80502585140344	2.71457153913500	2.71852344840247	2.71825590783778	2.71828672683485

The good performance of this method depends on f having sufficiently many smooth derivatives. If higher derivatives are not

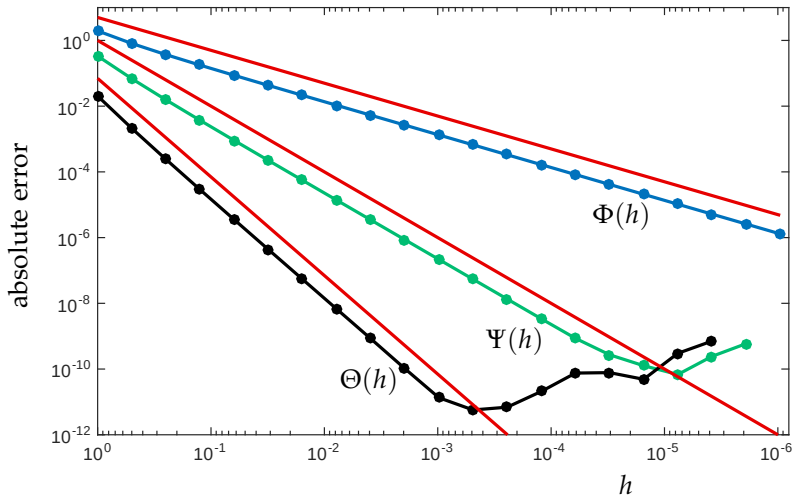


Figure 3.14: The convergence of $\Phi(h)$ (blue) along with its first two Richardson refinements, $\Psi(h)$ (green) and $\Theta(h)$ (black). The red lines show $\mathcal{O}(h)$, $\mathcal{O}(h^2)$, and $\mathcal{O}(h^3)$ convergence. For these values of h, rounding errors are not apparent in the $\Phi(h)$ plot; however, they lurk in the later digits of $\Phi(h)$, enough to interfere with the $\Psi(h)$ and $\Theta(h)$ approximations. Before these errors take hold, $\Theta(h)$ gives several additional orders of magnitude accuracy than was obtained by $\Phi(h)$ with much smaller h, in Figure 3.13.

smooth, then $\Phi(h)$ will not have smooth derivatives, and the accuracy breaks down. The accuracy also eventually degrades because of rounding errors that subtly pollute the initial column of data, as shown in the Figure 3.14.

3.5.1 *Extrapolation for higher order approximations*

In many cases, the initial algorithm $\Phi(h)$ is better than $\mathcal{O}(h)$ accurate, and in this case the formula for $R(j,k)$ should be adjusted to take advantage. Suppose that

$$\Phi(h) = \xi + c_1 h^r + c_2 h^{2r} + c_3 h^{3r} + \cdots$$

Notice that this structure is rather special: for example, if $r = 2$, then the Taylor series for $\Phi(h)$ must avoid all *odd-order* terms.

for some integer $r \geq 1$. Then define

$$R(j,0) := \Phi(h/2^j) \qquad \text{for } j \geq 0$$

$$R(j,k) := \frac{2^{rk} R(j,k-1) - R(j-1,k-1)}{2^{rk} - 1} \qquad \text{for } j \geq k > 0. \tag{3.9}$$

In this case, the $R(:,k)$ column will be $\mathcal{O}(h^{(k+1)r})$ accurate.

3.5.2 *Extrapolating the composite trapezoid rule: Romberg integration*

Suppose $f \in C^\infty[a,b]$, and we wish to approximate $\int_a^b f(x)\,\mathrm{d}x$ with the composite trapezoid rule,

$$T(h) = \frac{h}{2}\Big[f(a) + 2\sum_{j=1}^{n-1} f(a+jh) + f(b)\Big].$$

Notice that $T(h)$ only makes sense (as the composite trapezoid rule) when $h = (b-a)/n$ for some integer n. Notice that $T((b-a)/n)$

If you find this restriction on h distracting, just define $T(h)$ to be a sufficiently smooth interpolation between the values of $T((b-a)/n)$ for $n = 1, 2, \ldots$.

requires $n+1$ evaluations of the function f, and so increasing n (decreasing h) increases the expense.

One can show that for any $f \in C^\infty[a,b]$,

$$T(h) = \int_a^b f(x)\,\mathrm{d}x + c_1 h^2 + c_2 h^4 + c_3 h^6 + \cdots .$$

Now perform the generalized Richardson extrapolation (3.9) on $T(h)$ with $r = 2$:

$$R(j,0) = T(h/2^j) \qquad \text{for } j \geq 0$$

$$R(j,k) = \frac{4^k R(j,k-1) - R(j-1,k-1)}{4^k - 1} \qquad \text{for } j \geq k > 0.$$

This procedure is called *Romberg integration*.

In cases where $f \in C^\infty[a,b]$ (or if f has many continuous derivatives), the Romberg table will converge to high accuracy, though it may be necessary to take h to be relatively small before this is observed. When f does not have many continuous derivatives, each column of the Romberg table will still converge to the true integral, but not at the ever-improving clip we expect for smoother functions.

This procedure's utility is best appreciated through an example.

Example 3.7. For purposes of demonstration, we should use an integral we know exactly, say

$$\int_0^\pi \sin(x)\,\mathrm{d}x = 2.$$

Start the table with $h = \pi$ to generate $R(0,0)$, requiring 2 evaluations of $f(x)$. To build out the table, compute the composite trapezoid approximation based on an increasing number of function evaluations at each step. The final entry in the first column requires 129 function evaluations, and has four digits correct. This may not seem particularly impressive, but after refining these computations through a few steps of Romberg integration, we have an approximation that is accurate to full precision.

Ideally, one would exploit the fact that some grid points used to compute $T(h)$ are also required for $T(h/2)$, etc., thus limiting the number of new function evaluations required at each step.

j	$R(j,0)$	$R(j,1)$	$R(j,2)$	$R(j,3)$	$R(j,4)$	$R(j,5)$	$R(j,6)$
0	0.000000000000						
1	1.570796326795	2.094395102393					
2	1.896118897937	2.004559754984	1.998570731824				
3	1.974231601946	2.000269169948	1.999983130946	2.000005549980			
4	1.993570343772	2.000016591048	1.999999752455	2.000000016288	1.999999994587		
5	1.998393360970	2.000001033369	1.999999996191	2.000000000060	1.999999999996	2.000000000001	
6	1.999598388640	2.000000064530	1.999999999941	2.000000000000	2.000000000000	2.000000000000	2.000000000000

Be warned that Romberg results are not always as clean as this example, but this procedure is important tool to have at hand when high precision integrals are required. The general strategy of Richardson extrapolation can be applied to great effect in a wide variety of numerical settings.

4
Nonlinear Equations

LECTURE 27: *Introduction to Nonlinear Equations*
LECTURE 28: *Bracketing Algorithms*

THE SOLUTION OF NONLINEAR EQUATIONS has been a motivating challenge throughout the history of numerical analysis. We opened these notes with mention of Kepler's equation, $M = E - e \sin E$, where M and e are known, and E is sought. One can view this E as the solution of the equation $f(E) = 0$, where

$$f(x) = M - x + e \sin(x).$$

This is a simple equation in one variable, $x \in \mathbb{R}$, and it turns out that it is not particularly difficult to solve. Other examples are complicated by nastier nonlinearities, multiple solutions (in which case one might like to find them all), ill-conditioned zeros (where $f(x) \approx 0$ for x far from the true zeros of f), solutions in the complex plane, and expensive f evaluations.

Optimization is closely allied to the solution of nonlinear equations, since one finds extrema of $F : \mathbb{R} \to \mathbb{R}$ by solving

$$F'(x) = 0.$$

When $F : \mathbb{R}^n \to \mathbb{R}$, one seeks $\mathbf{x} \in \mathbb{R}^n$ that solves

$$\nabla F(\mathbf{x}) = \mathbf{0},$$

a nonlinear system of n equations in n variables. Handling $n > 1$ variables leads to more sophisticated theory and algorithms. Optimization problems become more difficult when additional constraints are imposed upon $\mathbf{x} \in \mathbb{R}^n$,

$$\min_{\mathbf{x} \in S} F(\mathbf{x}),$$

where $\mathcal{S}$ is the set of feasible solutions (e.g., vectors with nonnegative entries). When F is linear in $\mathbf{x}$ and $\mathbf{x}$ is constrained by simple inequalities, one has the important field of *linear programming*. When the set $\mathcal{S}$ constrains $\mathbf{x}$ to take discrete values (e.g., integers), the optimization problem becomes exceptionally difficult. Such *combinatorial optimization* or *integer programming* problems connect closely to the study of NP-hard problems in computer science.

In this course, we only have time to look at a few of the simplest algorithms for solving nonlinear equations. These will give some brief introduction to some of the over-arching themes in this important field. Further study is highly recommended!

4.1 Bracketing Algorithms for Root Finding

Given a function $f : \mathbb{R} \to \mathbb{R}$, we seek a point $x_* \in \mathbb{R}$ such that $f(x_*) = 0$. This x_* is called a *root* of the equation $f(x) = 0$, or simply a *zero* of f. At first, we only require that f be continuous a interval $[a, b]$ of the real line, $f \in C[a, b]$, and that this interval contains the root of interest. The function f could have many different roots; we will only look for one. In practice, f could be quite complicated (e.g., evaluation of a parameter-dependent integral or differential equation) that is expensive to evaluate (e.g., requiring minutes, hours, ...), so we seek algorithms that will produce a solution that is accurate to high precision while keeping evaluations of f to a minimum.

In some applications, like polynomial root-finding, we know f has multiple zeros, and we might want to find all of them. This particular case is complicated by the fact that the zeros could be located in the complex plane, $x_* \in \mathbb{C}$.

The first algorithms we study require the user to specify a finite interval $[a_0, b_0]$, called a *bracket*, such that $f(a_0)$ and $f(b_0)$ differ in sign, $f(a_0)f(b_0) < 0$. Since f is continuous, the intermediate value theorem guarantees that f has at least one root x_* in the bracket, $x_* \in (a_0, b_0)$.

4.1.1 Bisection

Given a bracket $[a, b]$ for which f takes opposite sign at a and b, the simplest technique for finding x_* is the *bisection* algorithm:

For $k = 0, 1, 2, \ldots$

1. Compute $f(c_k)$ for $c_k = \frac{1}{2}(a_k + b_k)$.
2. If $f(c_k) = 0$, exit; otherwise, repeat with
$$[a_{k+1}, b_{k+1}] := \begin{cases} [a_k, c_k], & \text{if } f(a_k)f(c_k) < 0; \\ [c_k, b_k], & \text{if } f(c_k)f(b_k) < 0. \end{cases}$$
3. Stop when the interval $b_{k+1} - a_{k+1}$ is sufficiently small, or if $f(c_k) = 0$.

How does this method converge? Not bad for such a simple idea. At

the kth stage, there must be a root in the interval $[a_k, b_k]$. Take $c_k = \frac{1}{2}(a_k + b_k)$ as the next estimate to x_*, giving the error $e_k = c_k - x_*$. The *worst* possible error would occur if x_* was close to a_k or b_k, half the bracket's width away from c_k, and hence $|e_k| = |c_k - x_*| \leq \frac{1}{2}(b_k - a_k) = 2^{-k-1}(b_0 - a_0)$.

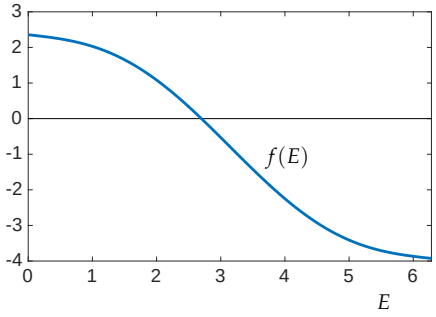
Theorem 4.1. Given a bracket $[a, b] \subset \mathbb{R}$ for which $f(a)f(b) < 0$, there exists $x_* \in [a, b]$ such that $f(x_*) = 0$ and the bisection point c_k satisfies

$$|c_k - x_*| \leq \frac{b-a}{2^{k+1}}.$$

We say this iteration *converges linearly* (the log of the error is bounded by a straight line when plotted against iteration count – see the next example) with rate $\rho = 1/2$. Practically, this means that the error is cut in half at each iteration, *independent of the behavior of* f. Reduction of the initial bracket width by ten orders of magnitude would require roughly $\log_2 10^{10} \approx 33$ iterations. If f is fast to evaluate, this convergence will be pretty quick; moreover, since the algorithm only relies on our ability to compute the sign of $f(x)$ accurately, the algorithm is robust to strange behavior in f (such as local minima).

Example 4.1. Kepler's equation is among the most historically important nonlinear equations, playing a pivotal role in two-body celestial mechanics. Given orbital parameters $e \in [0, 1)$ (the *eccentricity* of the elliptical orbit) and $M \in [0, 2\pi]$ (the *mean anomaly*), find $E \in [0, 2\pi]$ (the *eccentric anomaly*) such that

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



Cast this as the root-finding problem $f(E) = 0$, where

$$f(E) = M - E + e \sin E.$$

In this example we set $e = 0.8$ and $M = 3\pi/4$, yielding the function shown in the margin. Judging from this plot, the desired root E_* falls in the interval $[2, 3]$. Using the initial bracket $[a, b] = [2, 3]$, the bisection method converges as steadily as expected, cutting the error in half at every step. Figure 4.2 shows the convergence to the exact root $E_* = 2.69889638445749738544\ldots$.

4.1.2 *Regula Falsi*

A simple adjustment to bisection can often yield much quicker convergence. The name of the resulting algorithm, *regula falsi* (literally 'false rule') hints at the technique. As with bisection, begin with an interval $[a_0, b_0] \subset \mathbb{R}$ such that $f(a_0)f(b_0) < 0$. The goal is to be

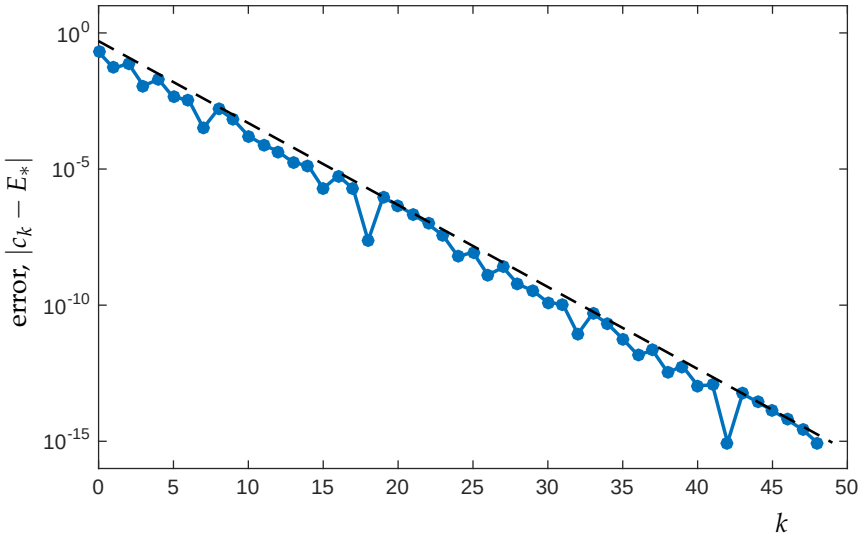


Figure 4.1: The convergence of the bisection method for solving Kepler's equation (Example 4.1) with initial bracket $[a, b] = [2, 3]$. Use the midpoint c_k of the kth bracket as an estimate of the root E_*. The dashed line shows the error bound $|c_k - E_*| \leq (b - a)/2^{k+1}$.

more sophisticated about the choice of the root estimate $c_k \in (a_k, b_k)$. Instead of simply choosing the middle point of the bracket as in bisection, we approximate f with the line $p_k \in \mathcal{P}_1$ that interpolates $(a_k, f(a_k))$ and $(b_k, f(b_k))$, so that $p_k(a_k) = f(a_k)$ and $p(b_k) = f(b_k)$. This unique polynomial is given (in the Newton form) by

$$p_k(x) = f(a_k) + \frac{f(b_k) - f(a_k)}{b_k - a_k}(x - a_k).$$

Now approximate the zero of f in $[a_k, b_k]$ by the zero of the linear model p_k:

$$c_k = \frac{a_k f(b_k) - b_k f(a_k)}{f(b_k) - f(a_k)}.$$

The algorithm then takes the following form:

For $k = 0, 1, 2, \ldots$

1. Compute $f(c_k)$ for $c_k = \dfrac{a_k f(b_k) - b_k f(a_k)}{f(b_k) - f(a_k)}$.
2. If $f(c_k) = 0$, exit; otherwise, repeat with
$$[a_{k+1}, b_{k+1}] := \begin{cases} [a_k, c_k], & \text{if } f(a_k) f(c_k) < 0; \\ [c_k, b_k], & \text{if } f(c_k) f(b_k) < 0. \end{cases}$$
3. Stop when $f(c_k)$ is sufficiently small, or the maximum number of iterations is exceeded.

Note that only Step 3 differs significantly from the bisection method. The former algorithm forces the bracket width $b_k - a_k$ to zero as it homes in on the root. In contrast, there is no mechanism in the *regula falsi* algorithm to drive the bracket width to zero: it will still always converge (in exact arithmetic) even though the bracket length does

not typically decrease to zero. Analysis of *regula falsi* is more complicated than the trivial bisection analysis; we give a convergence proof only for a special case. We will use the opportunity to introduce the notion of *convex functions*, a fundamental idea in optimization theory, especially for problems in higher dimensions.

Definition 4.1. Let $[a, b]$ be a finite interval of $\mathbb{R}$. Then a function $f \in C[a, b]$ is *convex* provided that for all $x, y \in [a, b]$ and $\lambda \in [0, 1]$,

$$f\big(\lambda x + (1 - \lambda)y\big) \leq \lambda f(x) + (1 - \lambda) f(y).$$

At first sight convexity might seem to be an abstract concept. However, an easy sufficient condition will be more familiar.

Lemma 4.1. If $f \in C^2[a, b]$ and $f''(x) \geq 0$ for all $x \in [a, b]$, then f is convex.

Proof. (See Rockafeller, *Convex Analysis*, Theorem 4.4.)

First note that f' is monotonically increasing on $[a, b]$. To see this, use the Fundamental Theorem of Calculus to write, for any $\xi \in [a, b]$,

$$f'(\xi) - f'(a) = \int_a^\xi f''(s)\, \mathrm{d}s.$$

The integrand on the right is nonnegative, so as ξ increases, the value of the integral cannot decrease. Since $f'(a)$ is a constant,

$$f'(\xi) = f'(a) + \int_a^\xi f''(s)\, \mathrm{d}s$$

is a monotonically increasing function of $\xi \in [a, b]$.

Now fix $\lambda \in [0, 1]$ and $z := \lambda x + (1 - \lambda)y$. Again use the Fundamental Theorem of Calculus to write

Such a z is a *convex combination* of x and y.

$$f(z) - f(x) = \int_x^z f'(s)\, \mathrm{d}s.$$

Now use monotonicity of f' to get the upper bound

$$\begin{aligned} f(z) &= f(x) + \int_x^z f'(s)\, \mathrm{d}s \\ &\leq f(x) + f'(z)(z - x). \end{aligned} \tag{4.1}$$

Another upper bound follows similarly. Write

$$f(y) - f(z) = \int_y^z f'(s)\, \mathrm{d}s$$

and again use monotonicity of the derivative (with $z \leq y$) to obtain

$$\begin{aligned} f(z) &= f(y) - \int_z^y f'(s)\, \mathrm{d}s \\ &\leq f(y) - f'(z)(y - z). \end{aligned} \tag{4.2}$$

Now preumultiply (4.1) by λ and (4.2) by $1-\lambda$ and add to obtain:

$$\begin{aligned} f(z) &= \lambda f(z) + (1-\lambda) f(z) \\ &\leq \lambda f(x) + (1-\lambda) f(y) + \lambda f'(z)(z-x) - (1-\lambda) f'(z)(y-z). \end{aligned} \tag{4.3}$$

Notice that

$$\lambda(z-x) - (1-\lambda)(y-z) = -\lambda x - (1-\lambda)y + z = 0,$$

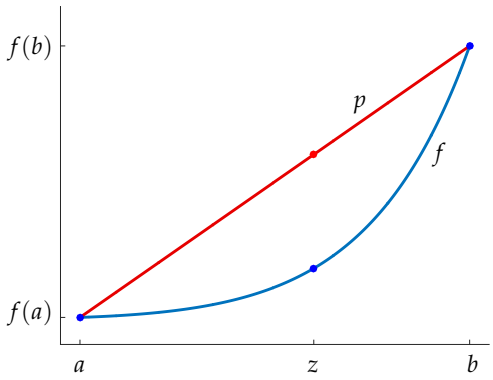
and hence (4.3) reduces to

$$f(z) \leq \lambda f(x) + (1-\lambda) f(y) + \lambda f'(z)(z-x).$$

Thus $f(x) \geq 0$ for all $x \in [a,b]$ proves that f is convex. ∎

Convexity implies that the linear interpolant will always be located above the function, as sketched on the right. The linear interpolant to f at $x = a$ and $x = b$ is

$$p(x) = f(a) + \frac{f(b)-f(a)}{b-a}(x-a).$$



Any $z \in [a,b]$ can be written as $z = \lambda a + (1-\lambda)b$, so

$$\begin{aligned} p(z) &= f(a) + \frac{f(b)-f(a)}{b-a}\big(\lambda a + (1-\lambda)b - a\big) \\ &= f(a) + \frac{f(b)-f(a)}{b-a}(1-\lambda)(b-a) \\ &= \lambda f(a) + (1-\lambda) f(b) \\ &\geq f\big(\lambda a + (1-\lambda)b\big) \\ &= f(z), \end{aligned}$$

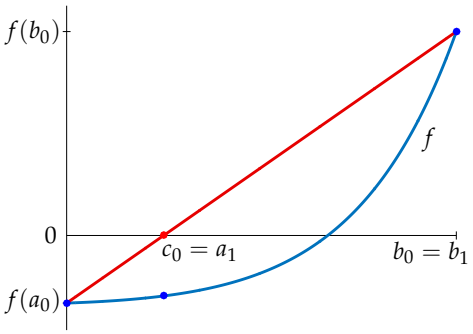
where the inequality follows from convexity of f.

This observation plays a key role in the next proof, which guarantees convergence of *regula falsi* for convex functions.

Theorem 4.2. Suppose f is a convex function on $[a_0, b_0]$ with $a_0 < b_0$ and $f(a_0) < 0 < f(b_0)$. Then *regula falsi* converges: either $f(c_k) = 0$ for some $k \geq 0$, or $c_k \to x_* \in [a_0, b_0]$ with $f(x_*) = 0$.

Proof. (See Stoer & Bulirsch, *Introduction to Numerical Analysis*, 2nd ed., §5.9.)

The argument preceding the proof ensures that the linear interpolant p to f at a_0 and b_0 satisfies $p(x) \geq f(x)$ for all $x \in [a_0, b_0]$. Since c_0 is selected so that $p_0(c_0) = 0$, it follows that $f(c_0) \leq 0$, so the new bracket will be $[a_1, b_1] = [c_0, b_0]$.



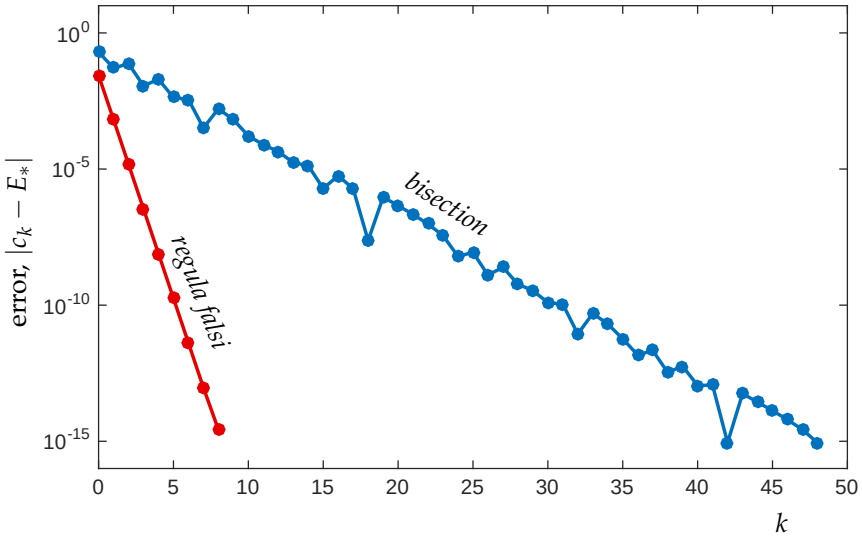


Figure 4.2: The convergence of the bisection and *regula falsi* methods for solving Kepler's equation (Example 4.1) with initial bracket $[a, b] = [2, 3]$. The root c_k in *regula falsi* converges to E_* much more rapidly than the midpoint c_k of the bisection brackets.

If $f(c_0) = 0$, the method has exactly found a root, and stops; otherwise,

$$f(a_1) = f(c_0) < 0 < f(b_0) = f(b_1).$$

Now use the convexity of f on the subinterval $[a_1, b_1]$ to iterate this argument, first showing that $[a_2, b_2] = [c_1, b_1]$, and then, in general,

$$[a_{k+1}, b_{k+1}] = [c_k, b_k].$$

Notice that $c_k > a_k = c_{k-1}$. If the algorithm never finds an exact root, it forms a sequence of estimates $\{c_k\}$ that is monotonically increasing and bounded above by b_0. The monotone convergence theorem in real analysis ensures that any bounded monotone sequence of real numbers must converge to a limit. Thus, $\lim_{k\to\infty} c_k = \gamma$ with $f(\gamma) \leq 0$, and this γ must be a fixed point of the *regula falsi* iteration, i.e.,

For a proof, see Rudin, *Principles of Mathematical Analysis*, Theorem 3.14.

$$\gamma = \frac{\gamma f(b_0) - b_0 f(\gamma)}{f(b_0) - f(\gamma)}.$$

Rearrange this expression to get $(\gamma - b_0)f(\gamma) = 0$. Since $f(b_0) > 0$, conclude that $\gamma \neq b_0$, and hence $f(\gamma) = 0$. Thus, *regula falsi* converges for convex functions. ∎

Example 4.2. Now apply the *regula falsi* method to the same version of Kepler's equation we solved with bisection in Example 4.1. Figure **??** compares the convergence of *regula falsi* to bisection, both with the same initial bracket $[2, 3]$. About 9 steps of *regula falsi* delivers the same accuracy obtained by more than 40 steps of bisection. For nice problems like this one, *regula falsi* is much more efficient.

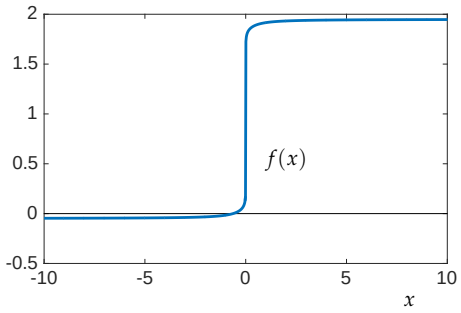
Is *regula falsi* always superior to bisection? If f has a zero, one can alway rig a bracket around it so that the zero x_* is exactly at its

midpoint, $x_* = \frac{1}{2}(a_0 + b_0)$, giving convergence of bisection in a single iteration. For most such functions, the first *regula falsi* iterate is different, and not a root of our function. Thus bisection can beat *regula falsi*, but such examples seem contrived, depending essentially on a lucky bracket. Can one construct more compelling examples?

Example 4.3. Lest Example 4.2 suggest that *regula falsi* is always superior to bisection, we now consider a function for which *regula falsi* converges very slowly. Sketch out a few sample functions. You will soon see how to design an f such that the root c_k of the linear approximation converges slowly toward x_* as k increases. The function should be relatively flat and small in magnitude in a large region near the root. One such example is

$$f(x) = \text{sign}(\tan^{-1}(x)) \left| \frac{2}{\pi} \tan^{-1}(x) \right|^{1/20} + \frac{19}{20},$$

which has a single root at $x_* \approx -0.6312881\ldots$. This function is illustrated in the margin. Figure 4.3 compares convergence of bisection and *regula falsi* for this f with the initial bracket $[-10, 10]$. The small value of f at the left end of the bracket ensures that $[a_1, b_1] = [c_0, b]$ will be almost as large as the initial bracket $[a, b]$.



4.1.3 *Accuracy*

Here we have assumed that we calculate $f(x)$ to perfect accuracy, an unrealistic expectation on a computer. If we attempt to compute x_* to very high accuracy, we will eventually experience errors due to inaccuracies in our function $f(x)$. For example, $f(x)$ may come from approximating the solution to a differential equation, were there

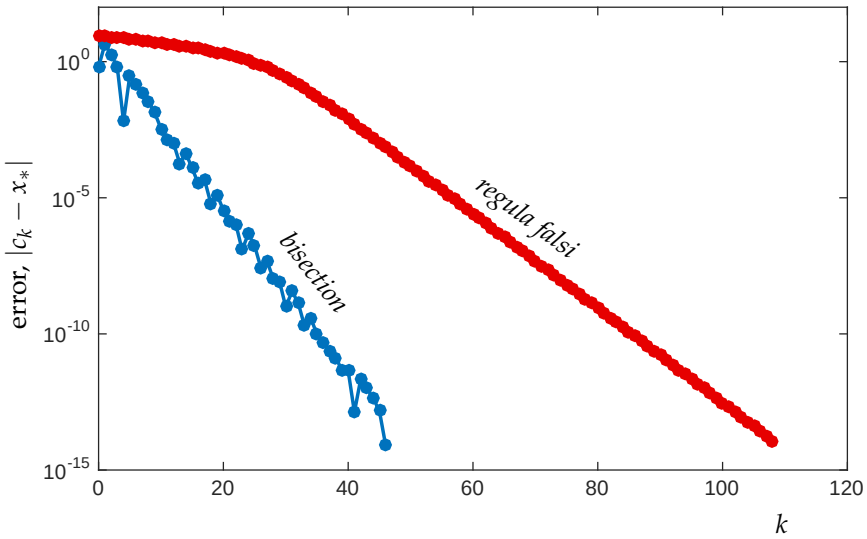
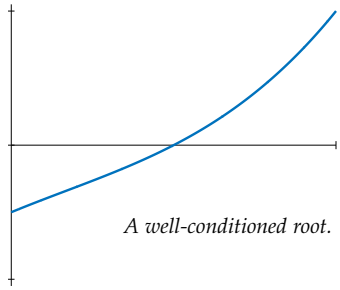


Figure 4.3: The convergence of the bisection and *regula falsi* methods for the function in Example 4.3 with initial bracket $[a, b] = [-10, 10]$. The root c_k in *regula falsi* converges very slowly to the right as k increases, due to the small value of f on the left end of the bracket. In contrast, bisection merrily cuts the error in half at each step, affected only by the sign of f, not its magnitude.

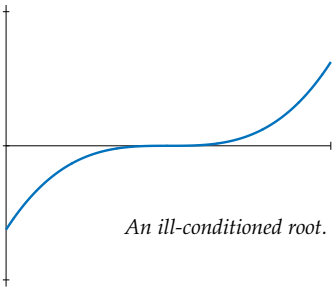
is some approximation error we must be concerned about; more generally, the accuracy of f will be limited by the computer's floating point arithmetic. One must also be cautious of subtracting one like quantity from another (as in construction of c_k in both algorithms), which can give rise to *catastrophic cancellation*.

4.1.4 *Conditioning*

When $|f'(x_0)| \gg 0$, the desired root is easy to pick out. In cases where $f'(x_0) \approx 0$, the root will be *ill-conditioned*, and it can be difficult to locate. This is the case, for example, when x_0 is a multiple root of f. (You may find it strange that the more copies of a root you have, the more difficult it can be to compute it!) In such cases bisection has the advantage that it only depends on the *sign* of f.



A well-conditioned root.



An ill-conditioned root.

4.1.5 *Deflation*

What is one to do if multiple distinct roots are required? One approach is to choose a new initial bracket that omits all known roots. Another technique, though numerically fragile, is to work with $\widehat{f}(x) := f(x)/(x - x_0)$, where x_0 is the previously computed root.

LECTURE 29: *Newton's Method*

We have studied two bracketing methods for finding zeros of a function, bisection and *regula falsi*. These methods have certain virtues (most importantly, they always converge), but some exploratory evaluations of f might be necessary to determine an initial bracket. Moreover, while these methods appear to converge fairly quickly, when f is expensive to compute (e.g., requiring solution of a differential equation) or many systems must be solved (e.g., to evaluate $\sqrt{A}$ as a zero of $f(x) = x^2 - A$ for many values of A), every objective function evaluation counts. The next few sections describe algorithms that can converge more quickly that bracketing methods — provided we have a sufficiently good initial estimate of the root.

4.2 *Newton's method*

We begin with Newton's method, the most celebrated root-finding algorithm. The algorithm's motivation resembles *regula falsi*: model f with a line, and estimate the root of f by the root of that line. In *regula falsi*, this line interpolated the function values at either end of the root bracket. Newton's method is based purely on local information at the current solution estimate, x_k. Whereas the bracketing methods only required that f be continuous, Newton's method needs $f \in C^1(\mathbb{R})$; to analyze the algorithm, we will further require $f \in C^2(\mathbb{R})$. This latter condition means that f can be expanded in a Taylor series centered at the approximate root x_k:

$$f(x_*) = f(x_k) + f'(x_k)(x_* - x_k) + \tfrac{1}{2} f''(\xi)(x_* - x_k)^2, \tag{4.4}$$

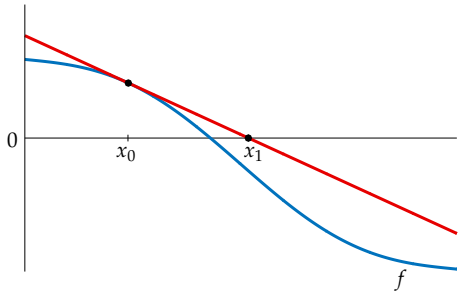
where x_* is the exact solution, $f(x_*) = 0$, and ξ is between x_k and x_*. Ignore the error term in this series, and you have a linear model for f; i.e., $f'(x_k)$ is the slope of the line secant to f at the point x_k. Specifically,

$$0 = f(x_*) \approx f(x_k) + f'(x_k)(x_* - x_k),$$

which implies

$$x_* \approx x_k - \frac{f(x_k)}{f'(x_k)}.$$

We get an iterative method by replacing x_* in this formula with x_{k+1}.



> *Newton's method* updates the approximate root x_k via
>
> $$x_{k+1} := x_k - \frac{f(x_k)}{f'(x_k)}. \tag{4.5}$$

Newton's method is famous because it often converges very quickly, but that excellent convergence comes at no small cost. For a bad starting guess x_0, it can *diverge* entirely. When it converges, the root it finds can, in some circumstances, depend sensitively on the initial guess: this is a famous source of beautiful fractals. However, for a good x_0, the convergence is usually lightning quick. Let $e_k = x_k - x_*$ be the error at the kth step. Subtract x_* from both sides of the iteration (4.5) to obtain a recurrence for the error,

$$e_{k+1} = e_k - \frac{f(x_k)}{f'(x_k)}. \tag{4.6}$$

The Taylor expansion of $f(x_*)$ about the point x_k given in (4.4) gives

$$0 = f(x_k) - f'(x_k)e_k + \tfrac{1}{2}f''(\xi)e_k^2.$$

Solving this equation for $f(x_k)$ and substituting that formula into the expression (4.6) for e_{k+1} gives

$$\begin{aligned} e_{k+1} &= e_k - \frac{f'(x_k)e_k + \frac{1}{2}f''(\xi)e_k^2}{f'(x_k)} \\ &= -\frac{f''(\xi)}{2f'(x_k)}e_k^2. \end{aligned}$$

When x_k converges to x_*, $\xi \in [x_*, x_k]$ also converges to x_*. Supposing that x_* is a simple root, so that $f'(x_*) \neq 0$, the above analysis suggests that when x_k is near x_*,

$$|e_{k+1}| \leq C|e_k|^2$$

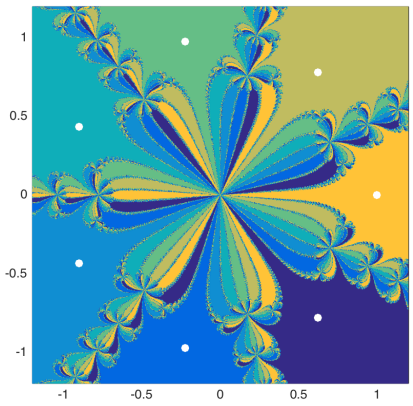
for constant

$$C = \frac{|f''(x_*)|}{2|f'(x_*)|}$$

independent of k. Thus we say that if $f'(\star) \neq 0$, then Newton's method *converges quadratically*, roughly meaning each iteration of Newton's method *doubles the number of correct digits* at each iteration. Compare this to bisection, where

$$|e_{k+1}| \leq \tfrac{1}{2}|e_k|,$$

meaning that the error was halved at each step. Significantly, Newton's method will often exhibit a transient period of linear convergence as x_k is gradually improved; once x_k gets close enough to x_*, the behavior transitions to quadratic convergence and full machine precision is attained in just a couple more iterations.



Newton's method for finding zeros of $f(x) = x^7 - 1$ in the complex plane. This function has seven zeros the complex plane, the principal roots of unity (shown as white dots in the plot). The color encodes the convergence of Newton's method: for each point $x_0 \in \mathbb{C}$, iterate Newton's method until convergence. The color corresponds to which of the seven roots that x_0 converged. In some regions, small changes to x_0 get attracted to vastly different roots.

Example 4.4. One way to compute $\sqrt{2}$ is to find the zero of

$$f(x) = x^2 - 2.$$

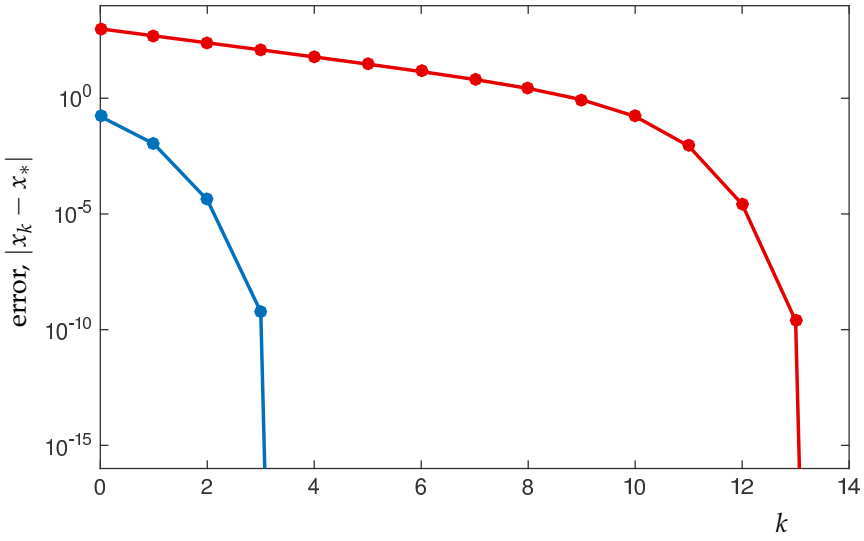


Figure 4.4: The convergence of Newton's method for $f(x) = x^2 - 2$. If x_0 is sufficiently close to the zero $x_* = \sqrt{2}$, the method converges very rapidly. For a poor x_0, the convergence is slow until x_k is sufficiently close. (The situation could be worse: in many cases, bad initial guesses lead to the divergence of Newton's method.)

Figure 4.5 shows convergence in just a few steps for the realistic starting guess $x_0 = 1.25$. The plot also shows the convergence behavior for the (entirely ridiculous) starting guess $x_0 = 1000$, to illustrate a linear phase of convergence as the iterate gradually approaches the region of quadratic convergence. Once x_k is sufficiently close to x_*, convergence proceeds very quickly.

Table 4.1 shows the iterates for $x_0 = 1000$, computed exact arithmetic in Mathematica, and displayed here to more than eighty digits. This is a bit excessive: in the floating point arithmetic we have used

k	x_k
0	1000.0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
1	500.0010000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
2	250.0024999960000079999840000319999360001279997440005119989760020479959040081919836160 3
3	125.0052499580004679945840630552651285659801482359562239344169580047744668579946389648 4
4	62.5106246430170331488869135840332046452975994432574456663116460063101739147830976134 1
5	31.2713096020621945559642235877170054837456580184233208653636523657827808040615382736 4
6	15.6676329948683664003075552710028165206510015971032445945258154376740347992183401224 8
7	7.8976423478563580671905136093423623811696836517416702511646103416077762821736496011 1
8	4.0754412405194989208879857338706713335299114996130926715933398019154830807536096186 2
9	2.2830928243925538398630669035817794614433923363437778160605553848163720075955537623 6
10	1.5795487524060153652754700172749893512746398177638901618897579136393958626586032325 1
11	1.4228665795786682509120968385630981830931092942876392816289093467384703623818499269 3
12	1.4142398735915306231936461644112003518252948934786012671639574689639269004077455837 5
13	1.4142135626178485126558900035917439663220762854896890824239894439161543633562536005 6
14	1.4142135623730950488228680777571711822141811472942311663725480437703133244040615571 6
15	1.4142135623730950488016887242096980785698304670594999486043964007946076509385830519 0
16	1.4142135623730950488016887242096980785696718753769480731766797379907324784621070477 4
exact	1.4142135623730950488016887242096980785696718753769480731766797379907324784621070388503875 3 . . .

Table 4.1: Convergence of Newton's method for $f(x) = x^2 - 2$ for a poor initial guess. The first 9 iterations each roughly cut he error in half; later iterations then start to double the correct digits (blue, underlined).

all semester, we can only expect to get 15 or 16 digits of accuracy in the best case. It is worth looking at all these digits to get a better appreciation of the quadratic convergence. Once we are in the quadratic regime, notice the characteristic doubling of the number of correct digits (in blue and underlined) at each iteration.

LECTURE 30: *Convergence of Newton's Method via Direct Iteration*

4.3 Direct iteration

We have already performed a simple analysis of Newton's method to gain an appreciation for the quadratic convergence rate. For a broader perspective, we shall now put Newton's method into a more general framework, so that the accompanying analysis would allow us to understand simpler iterations like the 'constant slope method:'

$$x_{k+1} = x_k - \alpha f(x_k)$$

for some constant α (which could approximate $1/f'(x_*)$, for example). We begin by formalizing the notion of the rate of convergence.

Definition 4.2. A root-finding algorithm is *pth-order convergent* if

$$|e_{k+1}| \leq C|e_k|^p$$

for some $p \geq 1$ and positive constant C.

- If $p = 1$, then $C < 1$ is necessary for convergence, and C is called the *linear convergence rate*.
- If $p > 1$, the constant C is called the *asymptotic error constant*.

In the last section, our analysis showed that, if $f'(x_*) \neq 0$ and x_0 is sufficiently close to x_*, then Newton's method is second-order convergent with asymptotic error constant

$$C = \frac{|f''(x_*)|}{2|f'(x_*)|}.$$

Earlier we saw that bisection is linearly convergent for $f \in C[a_0, b_0]$ with rate $C = 1/2$.

One can analyze Newton's method and its variants through the following general framework. Consider *direct iterations* of the form

$$x_{k+1} = \Phi(x_k),$$

for some iteration function Φ. This framework will include Newton's method, since we can take

$$\Phi(x) = x - \frac{f(x)}{f'(x)}.$$

If the starting guess is an exact root, $x_0 = x_*$, the method should be smart enough to return $x_1 = x_*$. Thus the root x_* is a *fixed point* of Φ:

$$x_* = \Phi(x_*).$$

For further details on this standard approach, see G. W. Stewart, *Afternotes on Numerical Analysis*, §§2–4; J. Stoer & R. Bulirsch, *Introduction to Numerical Analysis*, 2nd ed., §5.2; L. W. Johnson and R. D. Riess, *Numerical Analysis*, second ed., §4.3.

We seek an expression for the error $e_{k+1} = x_{k+1} - x_*$ in terms of e_k and properties of Φ. Assume, for example, that $\Phi(x) \in C^2(\mathbb{R})$, so that we can write the Taylor series for Φ expanded about x_*:

$$\begin{aligned} x_{k+1} = \Phi(x_k) &= \Phi(x_*) + (x_k - x_*)\Phi'(x_*) + \tfrac{1}{2}(x_k - x_*)^2\Phi''(\xi) \\ &= x_* + (x_k - x_*)\Phi'(x_*) + \tfrac{1}{2}(x_k - x_*)^2\Phi''(\xi) \end{aligned}$$

for some ξ between x_k and x_*. From this we obtain an expression for the errors:

$$e_{k+1} = e_k\Phi'(x_*) + \tfrac{1}{2}e_k^2\Phi''(\xi).$$

Convergence analysis is reduced to the study of $\Phi'(x_*)$, $\Phi''(x_*)$, etc. To analyze methods that converge faster than quadratic, one would simply add extra terms in the Taylor series.

Example 4.5. For Newton's method

$$\Phi(x) = x - \frac{f(x)}{f'(x)},$$

so the quotient rule gives

$$\Phi'(x) = 1 - \frac{f'(x)^2 - f(x)f''(x)}{f'(x)^2} = \frac{f(x)f''(x)}{f'(x)^2}.$$

Provided x_* is a simple root so that $f'(x_*) \neq 0$ (and supposing $f \in C^2(\mathbb{R})$), we have $\Phi'(x_*) = 0$, and thus

$$e_{k+1} = \tfrac{1}{2}e_k^2\Phi''(\xi),$$

and hence we again see quadratic convergence provided x_k is sufficiently close to x_*. The asymptotic error constant is thus

$$C = \frac{|\Phi''(x_*)|}{2} = \frac{|f''(x_*)|}{2|f'(x_*)|},$$

as expected.

What happens when $f'(x_*) = 0$? The direct iteration framework makes it straightforward to analyze this situation. If x_* is a multiple root, we might worry that Newton's method might have trouble converging, since we are dividing $f(x_k)$ by $f'(x_k)$, and both quantities are nearing zero as $x_k \to x_*$. To study convergence, we investigate

$$\lim_{x \to x_*} \Phi'(x) = \lim_{x \to x_*} \frac{f(x)f''(x)}{f'(x)^2}.$$

This limit has the indeterminate form $0/0$. Assuming sufficient differentiability, we can invoke l'Hôpital's rule:

$$\lim_{x \to x_*} \frac{f(x)f''(x)}{f'(x)^2} = \lim_{x \to x_*} \frac{f'(x)f''(x) + f(x)f'''(x)}{2f'(x)f''(x)},$$

but this is also of the indeterminate form $0/0$ when $f'(x_*) = 0$. Again using l'Hôpital's rule and now assuming $f''(x_*) \neq 0$,

$$\lim_{x\to x_*} \frac{f(x)f''(x)}{f'(x)^2} = \lim_{x\to x_*} \frac{f''(x)^2 + 2f'(x)f'''(x) + f(x)f^{(iv)}(x)}{2(f'(x)f'''(x) + f''(x)^2)} = \lim_{x\to x_*} \frac{f''(x)^2}{2f''(x)^2} = \frac{1}{2}.$$

Thus, Newton's method converges locally to a double root according to

$$e_{k+1} = \tfrac{1}{2}e_k + \mathcal{O}(e_k^2).$$

This is *linear* convergence *at the same rate as bisection*! If x_* has multiplicity exceeding two, then $f''(x_*) = 0$ and further analysis is required. One would find that the rate remains linear, and gets even slower. The slow convergence of Newton's method for multiple roots is exacerbated by the chronic ill-conditioning of such roots. Let us summarize what might seem to be a paradoxical situation: the more 'copies' of root there are present, the more difficult that root is to find!

LECTURE 31: *The Secant Method: Prototypical Quasi-Newton Method*

Newton's method is fast if one has a good initial guess x_0. Even then, it can be inconvenient and expensive to compute the derivatives $f'(x_k)$ at each iteration. The final root finding algorithm we consider is the *secant method*, a kind of *quasi-Newton method* based on an approximation of f'. It can be thought of as a hybrid between Newton's method and *regula falsi*.

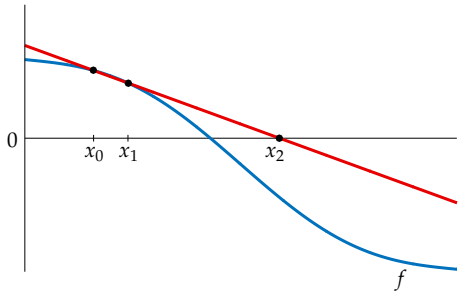
4.4 *The Secant Method*

Throughout this semester, we have seen how derivatives can be approximated using finite differences, for example,

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

for some small h. (Recall that too-small h will give a bogus answer due to rounding errors, so some caution is needed; see Section 3.5.) What if we replace $f'(x_k)$ in Newton's method with this sort of approximation? The natural algorithm that emerges is the *secant method*,

$$x_{k+1} = x_k - f(x_k)\,\frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})} = \frac{x_{k-1}f(x_k) - x_k f(x_{k-1})}{f(x_k) - f(x_{k-1})}.$$



Note the similarity between this formula and the *regula falsi* iteration:

$$c_k = \frac{a_k f(b_k) - b_k f(a_k)}{f(b_k) - f(a_k)}.$$

Both methods approximate f by a line that joins two points on the graph of f, but the secant method requires no initial bracket for the root. Instead, the user simply provides *two* starting points x_0 and x_1 with no stipulation about the signs of $f(x_0)$ and $f(x_1)$. As a consequence, there is no guarantee that the method will converge: as in Newton's method, a poor initial guess can lead to divergence.

Do we recover the convergence behavior of Newton's method? Not quite – but it is still better than the linear convergence exhibited by the bisection and regula falsi methods (provided it does converge). The convergence theory brings in a new technique we have not see before, where the error $e_k = x_k - x_*$ is presumed to reduce according to a generic convergence behavior, $e_{k+1} \approx M e_k^r$, and this *ansatz* is used to derive values of M and r. Begin by writing the linear interpolant to f at x_k and x_{k-1} in the Newton form

See Dahlquist and Björck, *Numerical Methods*, Section 6.4.3.

$$p(x) = f(x_k) + \frac{x - x_k}{x_{k-1} - x_k}\big(f(x_{k-1}) - f(x_k)\big). \tag{4.7}$$

Once again we can put the interpolation error formula (Theorem 1.3) to good use. Assuming that $f \in C^2(\mathbb{R})$, for any $x \in \mathbb{R}$ one can write

$$f(x) - p(x) = \frac{f''(\xi)}{2}(x - x_k)(x - x_{k-1}),$$

where ξ falls within the extremes of x, x_k, and x_{k-1}. Since $f(x_*) = 0$, we can thus write

$$0 = p(x_*) + \frac{f''(\xi)}{2}(x_* - x_k)(x_* - x_{k-1}).$$

Defining $e_j := x_j - x_*$ as usual, this last equation is

$$0 = p(x_*) + \frac{f''(\xi)}{2} e_k e_{k-1}.$$

Substituting formula (4.7) for p gives

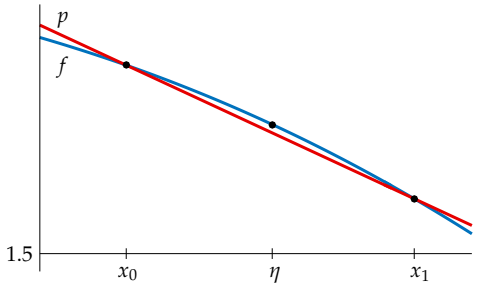
$$0 = f(x_k) + \frac{x_* - x_k}{x_{k-1} - x_k}\big(f(x_{k-1}) - f(x_k)\big) + \frac{f''(\xi)}{2} e_k e_{k-1}. \tag{4.8}$$

Now recall that, by design, the secant method picks x_{k+1} as the zero of p, i.e.,

$$\begin{aligned} 0 &= p(x_{k+1}) \\ &= f(x_k) + \frac{x_{k+1} - x_k}{x_{k-1} - x_k}\big(f(x_{k-1}) - f(x_k)\big). \end{aligned} \tag{4.9}$$

Subtract (4.8) from (4.9) to obtain

$$\begin{aligned} 0 &= \frac{x_{k+1} - x_*}{x_{k-1} - x_k}\big(f(x_{k-1}) - f(x_k)\big) - \frac{f''(\xi)}{2} e_k e_{k-1} \\ &= e_{k+1}\frac{f(x_{k-1}) - f(x_k)}{x_{k-1} - x_k} - \frac{f''(\xi)}{2} e_k e_{k-1}. \end{aligned} \tag{4.10}$$



We can clean up this last formula by realizing that

$$\frac{f(x_{k-1}) - f(x_k)}{x_{k-1} - x_k}$$

is the slope of the linear interpolant p. It is also a difference quotient for f, and so, by the Mean Value Theorem, there exists $\eta \in [x_k, x_{k-1}]$ where the slope of f matches that of the interpolant:

$$f'(\eta) = \frac{f(x_{k-1}) - f(x_k)}{x_{k-1} - x_k}.$$

Substituting this formula into (4.10) gives

$$0 = f'(\eta)e_{k+1} - \frac{f''(\xi)}{2} e_k e_{k-1},$$

which can be solved for

$$e_{k+1} = \frac{f''(\xi)}{2f'(\eta)} e_k e_{k-1}. \tag{4.11}$$

Contrast this formula with the error recurrence for Newton's method in (4.7):

$$e_{k+1} = -\frac{f''(\xi)}{2f'(x_k)} e_k^2.$$

As x_k and x_{k-1} converge to x_*, the values for ξ and η must also converge toward x_*, justifying the approximation

$$e_{k+1} \approx C e_k e_{k-1} \tag{4.12}$$

for asymptotic error constant

$$C = \frac{f''(x_*)}{2f'(x_*)},$$

presuming, as usual, that x_* is a simple root, so $f'(x_*) \neq 0$.

It remains to convert the approximation (4.12) into some *order* of convergence. As x_k converges, we expect $|e_{k-1}|$ to be larger than e_k, so $e_k e_{k-1}$ in the secant method is probably not as small as the e_k^2 term that appeared in the analogous formula for Newton's method. Can we quantify how much smaller? Suppose the error obeys the approximation

$$e_{j+1} \approx M e_j^r \tag{4.13}$$

for some constants M and r. Then

$$e_k \approx M e_{k-1}^r$$

implies that

$$e_{k-1} \approx M^{-1/r} e_k^{1/r},$$

and so the error approximation (4.12) suggests

$$e_{k+1} \approx C e_k e_{k-1} \approx C M^{-1/r} e_k^{1+1/r}. \tag{4.14}$$

This equation must agree with the the error form (4.13) with $j = k$:

$$e_{k+1} \approx M e_k^r. \tag{4.15}$$

Equating the approximations (4.14) and (4.15) gives

$$C M^{-1/r} e_k^{1+1/r} = M e_k^r.$$

Matching the *constants* then gives

$$M = C^{r/(r+1)},$$

while matching the *rates* gives $1 + 1/r = r$, i.e.,

$$r^2 - r - 1 = 0.$$

Solve this quadratic to get

$$r = \frac{1 \pm \sqrt{5}}{2}.$$

The negative sign choice gives $r = (1 - \sqrt{5})/2 = -0.6180\ldots$, which does not correspond to a convergent process. (If $|e_k| < 1$, then $|e_k^{-0.6180\ldots}| > 1$.) Thus, if the process *converges* according to $e_{k+1} \leq M e_k^r$, the r must correspond to the positive root,

$$r = \frac{1 + \sqrt{5}}{2} := \varphi = 1.6180\ldots,$$

the *golden ratio*. It follows that

$$|e_{k+1}| \leq M |e_k|^\varphi$$

for a constant $M > 0$. Note that $\varphi < 2$, so, in the region of asymptotic convergence (x_k close to x_*), one step of the secant method will make a bit less progress to the root than one step of Newton's method.

Though you may regret that the secant method does not recover the quadratic convergence of Newton's method, take solace in the fact that the secant method requires only one function evaluation $f(x_k)$ at each iteration, as opposed to Newton's method, which requires $f(x_k)$ and $f'(x_k)$. Typically the derivative is more expensive to compute than the function itself. Assuming that evaluating $f(x_k)$ and $f'(x_k)$ requires the same amount of effort, then we can compute *two steps* of the secant method for roughly the same cost as a *one step* of Newton's method. These two steps of the secant method combine to give an improved convergence rate:

Of course, for the secant method one stores the $f(x_{k-1})$ value computed during the previous iteration.

$$|e_{k+2}| \leq M |e_{k+1}|^\varphi \leq M \left| M |e_k|^\varphi \right|^\varphi \leq M^{1+\varphi} |e_k|^{\varphi^2},$$

where $\varphi^2 = \frac{1}{2}(3 + \sqrt{5}) \approx 2.62 > 2$. Hence, in terms of computing time, the secant method can actually be more efficient than Newton's method.

This discussion is drawn from §3.3 of Kincaid and Cheney, *Numerical Analysis*, 3rd ed.

Figure 4.5 compares the convergence of the secant method to Newton's method for the function $f(x) = x^2 - 2$, which we can use to compute $x_* = \sqrt{2}$ as in Figure 4.5. This example starts with the (bad) initial guess $x_0 = 10$. To ensure that the secant method is not hampered by a bad value of x_1, this experiments uses the same x_1 value computed using Newton's method. After these two initial steps, both methods steadily converge, but Newton's method takes fewer iterations, in agreement with the theory derived in this and the last lecture. Table 4.2 shows the iterates x_k and magnitude of the errors $|e_k|$ for both methods.

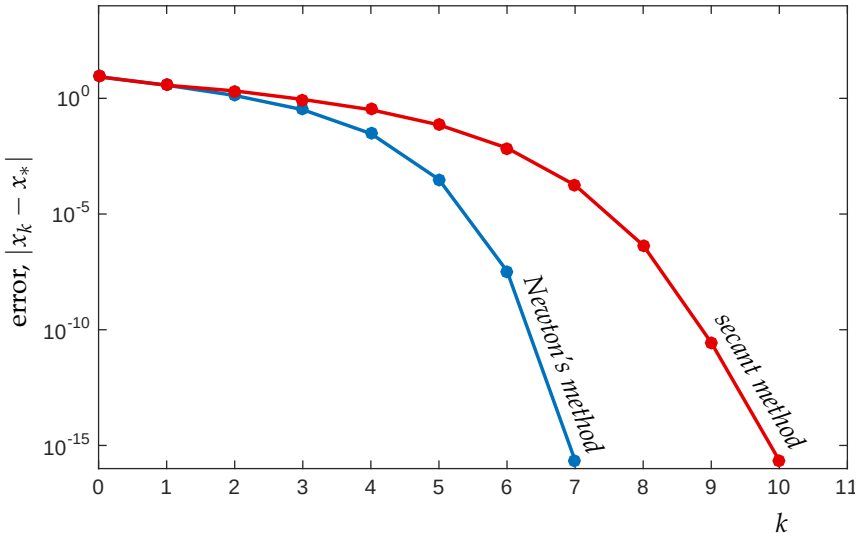


Figure 4.5: The convergence of Newton's method and the secant method for $f(x) = x^2 - 2$. Both methods start with $x_0 = 1$, and the secant method uses the same x_1 that was generated by the first step of Newton's method.

k	x_k (*Newton*)	$\lvert e_k \rvert$ (*Newton*)	x_k (*secant*)	$\lvert e_k \rvert$ (*secant*)
0	10.000000000000	8.585786437627	10.000000000000	8.585786437627
1	5.100000000000	3.685786437627	5.100000000000	3.685786437627
2	2.746078431373	1.331864868999	3.509933774834	2.095720212461
3	1.737194874380	0.322981312007	2.311360664564	0.897147102191
4	1.444238094866	0.030024532493	1.737194874380	0.322981312007
5	1.414525655149	0.000312092776	1.485785199551	0.071571637178
6	1.414213596802	0.000000034429	1.421385900007	0.007172337634
7	1.414213562373	0.000000000000	1.414390138133	0.000176575760
8			1.414214008974	0.000000446601
9			1.414213562401	0.000000000028
10			1.414213562373	0.000000000000

Table 4.2: Comparison of iterates from Newton's method and the secant method for finding a zero of $f(x) = x^2 - 2$.

5
Ordinary Differential Equations

LECTURE 32: *Introduction to Numerical Integration*

THE FINAL SEGMENT of the course addresses techniques for approximating the solution of an ordinary differential equation of the general form

$$x'(t) = f(t, x(t)).$$

For the most part, we will consider initial value problems, where the solution is determined by an *initial condition*

$$x(t_0) = x_0.$$

A wide variety of methods have been proposed to solve such equations, often derived from the techniques of interpolation, approximation, and quadrature studied earlier in the course.

Differential equations play the dominant role in applied mathematics. Their (approximate) solution is required in nearly every corner of physics, chemistry, biology, engineering, finance, and beyond. For many practical problems involving nonlinearities, one cannot write down a closed-form solution to a differential equation in terms of familiar functions such as polynomials, trigonometric functions, and exponentials. Thus the numerical solution of differential equations is an enormous field. In this course we shall only be able to focus on ordinary differential equations (ODEs). Partial differential equations (PDEs) are even more challenging, requiring several additional semester-long courses to cover the basic methods.

The numerical solution of differential equations began in earnest with Leonhard Euler and his contemporaries in the mid 1700s, with especially important contributions following between 1880 and 1905. The primary motivating application was celestial mechanics, where the approximate integration of Newton's differential equations of motion was needed to predict the orbits of planets and comets. Indeed celestial mechanics (more generally, Hamiltonian systems) con-

tinues to motivate the field, leading to recent innovations in so-called *geometric* or *symplectic* integrators.

5.1 Existence theory for ODEs

Before computing numerical solutions to ODE initial value problems, we should address the variety of problems that arise, and the theoretical conditions under which one can guarantee that a solution exists.

5.1.1 Scalar equations

A standard scalar initial value problem takes the form

Given:	$x'(t) = f(t, x(t))$, with $x(t_0) = x_0$
Determine:	$x(t)$ for all $t \geq t_0$.

That is, we are given a formula for the derivative of some unknown function $x(t)$, together with a single value of the function at some *initial time*, t_0. The goal is to use this information to determine $x(t)$ at all points t beyond the initial time.

Differential equations are an inherently graphical subject, so we should examine a few sample problems and plots of their solutions.

Example 5.1. First consider the simple, essential model problem

$$x'(t) = \lambda x(t),$$

with exact solution $x(t) = e^{\lambda t} c$, where the constant c is determined by the initial data (t_0, x_0). In the common case that $t_0 = 0$,

$$x_0 = x(0) = e^{\lambda \cdot 0} c = c,$$

so the exact solution is

$$x(t) = e^{\lambda t} x_0.$$

If $\lambda > 0$, the solution grows exponentially with t; $\lambda < 0$ yields exponential decay. Because this linear equation is easy to solve, it provides a good test case for numerical algorithms. Moreover, it is the prototypical linear ODE; from it, we gain insight into the local behavior of nonlinear ODEs.

Applications typically give equations whose whose solutions cannot be expressed as simply as the solution of this linear model problem. Among the tools that improve our understanding of more difficult problems is the *direction field* of the function $f(t, x)$, a key

If $\lambda = \alpha + i\beta \in \mathbb{C}$ (with $\alpha, \beta \in \mathbb{R}$), then

$$e^{t\lambda} = e^{t(\alpha + i\beta)} = e^{t\alpha} e^{it\beta}.$$

Since $it\beta$ is purely imaginary, $|e^{it\beta}| = 1$, so

$$|e^{t\lambda}| = e^{t\alpha}.$$

Thus $|e^{t\lambda}| \to 0$ as $t \to \infty$ if $\operatorname{Re} \lambda < 0$, while $|e^{t\lambda}| \to \infty$ as $t \to \infty$ if $\operatorname{Re} \lambda > 0$.

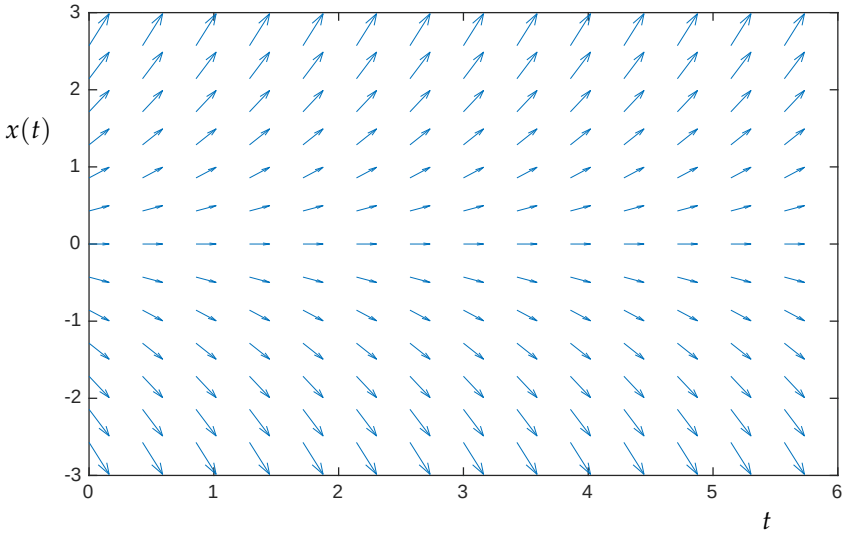


Figure 5.1: Force field for the equation $x'(t) = x(t)$, for which $f(t, x) = x$.

technique from the sub-discipline of the *qualitative analysis* of ODEs. Here is the critical idea: The function $f(t, x(t))$ reveals the *slope* of the solution $x(t)$ going through any point in the $(t, x(t))$ plane. Hence one can get a good impression about the behavior of a differential equation by plotting these slopes throughout some interesting region of the $(t, x(t))$ plane.

For an elementary introduction to the qualitative analysis of ODEs, see Hubbard and West, *Differential Equations: A Dynamical Systems Approach, Part I*, Springer-Verlag, 1991.

To plot the direction field, let the horizontal axis represent t, and the vertical axis represent x. Then divide the (t, x) plane with regular grid points, $\{(t_j, x_k)\}$. Centered at each grid point, draw a line segment whose slope is $f(t_j, x_k)$. To get a rough impression of the solution of the differential equation $x'(t) = f(t, x)$ with $x(t_0) = x_0$, begin at the point (t_0, x_0), and follow the direction of the slope lines.

Figure 5.1 shows the direction field for $x'(t) = x(t)$, giving $f(t, x) = x$. Since f does not depend directly on t, the differential equation is *autonomous*. In the plot of the direction field, for a fixed value of x, the arrows point in the same direction and have the same magnitude for all t.

One only needs a few simple MATLAB commands to produce a direction field like the one seen in Figure 5.1, thanks to the build-in `quiver` routine.

```
f = inline('x','t','x');                        % x' = f(t,x) = x
x = linspace(-3,3,15); t = linspace(0,6,15);    % grid of points at which to plot the slope
[T,X] = meshgrid(t,x);                          % turn grid vectors into matrices
figure(1), clf
quiver(T,X,ones(size(T)),f(T,X)), hold on       % produce a "quiver" plot
axis([min(t) max(t) min(x) max(x)])             % adjust the axes
```

Figure 5.2 repeats Figure 5.1, but now showing solution trajectories for $x(0) = 0.1$ and $x(0) = -0.01$. Notice how these solutions follow

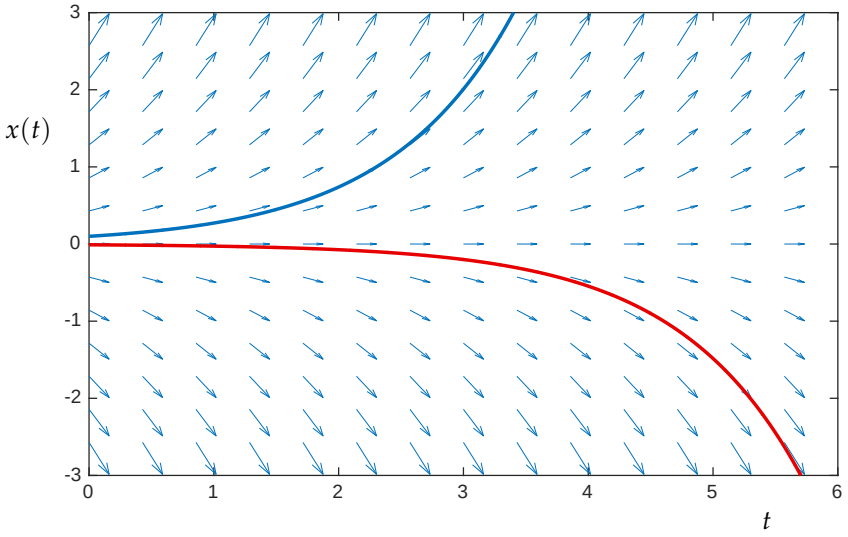


Figure 5.2: Force field for the equation $x'(t) = x(t)$, now showing solutions for $x(0) = 0.1$ (in blue) and $x(0) = -0.01$ (in red).

the arrows in the direction field.

Example 5.2. Next consider an equation that, for most $x(0)$, lacks an elementary solution that can be expressed in closed form,

$$x'(t) = \sin(tx(t)).$$

The direction field for $\sin(xt)$ is shown below. Though we don't have access to the exact solution, it is a simple matter to compute accurate approximations. Several solutions (for $x(0) = 3$, $x(0) = 0$, and $x(0) = -2$) are superimposed on the direction field. These were computed using a one-step method of the kind we will discuss momentarily. (Those areas where up and down arrows appear to cross

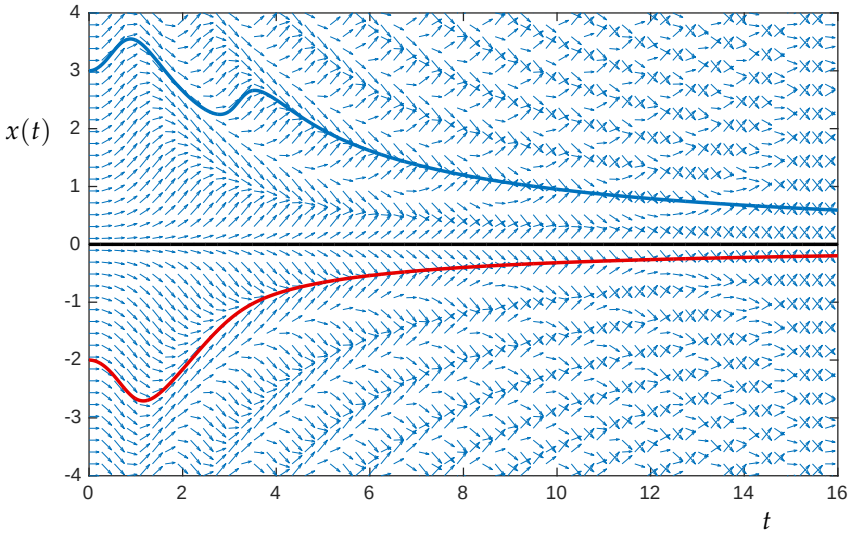


Figure 5.3: Force field for the equation $x'(t) = \sin(tx(t))$, showing solutions for $x(0) = 3$ (blue), $x(0) = 0$ (black) and $x(0) = -2$ (red).

are asymptotes of the solution: between the up and down arrow is a point where the slope $f(t, x)$ is zero.)

5.1.2 *Systems of equations*

In most applications we do not have a simple scalar equation, but rather a system of equations describing the coupled dynamics of several variables. Such situations give rise to vector-valued functions $\mathbf{x}(t) \in \mathbb{R}^n$. In particular, the initial value problem becomes

Given:	$\mathbf{x}'(t) = \mathbf{f}(t, \mathbf{x}(t))$, with $\mathbf{x}(t_0) = \mathbf{x}_0$
Determine:	$\mathbf{x}(t)$ for all $t \geq t_0$.

All the techniques for solving scalar initial value problems described in this course can be applied to systems of this type.

5.1.3 *Higher-order ODEs*

Newton's Second Law, $\mathbf{F}(t) = m\mathbf{a}(t)$, leads to many important second-order differential equations. Noting the acceleration $\mathbf{a}(t)$ is the second derivative of the position $\mathbf{x}(t)$, we arrive at

$$\mathbf{x}''(t) = m^{-1}\mathbf{F}(t).$$

Thus, we are often interested in systems of higher-order ODEs.

To keep the notation simple, consider the scalar second-order problem

Given:	$x''(t) = f(t, x(t), x'(t))$, with $x(t_0) = x_0$, $x'(t_0) = y_0$
Determine:	$x(t)$ for all $t \geq t_0$.

Note, in particular, that the initial conditions $x(t_0)$ and $x'(t_0)$ must both be supplied.

In some cases, one instead knows $x(t)$ at two distinct points, $x(t_0) = x_0$ and $x(t_{\text{final}}) = x_{\text{final}}$, leading to an ODE *boundary value problem*.

This second order equation (and higher-order ODE's as well) can always be written as a first order system of equations. Define $x_1(t) = x(t)$, and let $x_2(t) = x'(t)$. Then

$$\begin{aligned} x_1'(t) &= x'(t) = x_2(t) \\ x_2'(t) &= x''(t) = f(t, x(t), x'(t)) = f(t, x_1(t), x_2(t)). \end{aligned}$$

Writing this in vector form, $\mathbf{x}(t) = [x_1(t)\ x_2(t)]^{\mathrm{T}}$, and the differential equation becomes

Fonts matter: $x(t)$ denotes a scalar quantity, while $\mathbf{x}(t)$ is a vector.

$$\mathbf{x}'(t) = \begin{bmatrix} x_1'(t) \\ x_2'(t) \end{bmatrix} = \begin{bmatrix} x_2(t) \\ f(t, x_1(t), x_2(t)) \end{bmatrix} = \mathbf{f}(t, \mathbf{x}(t)).$$

The initial value is given by

$$\mathbf{x}_0 = \begin{bmatrix} x_1(t_0) \\ x_2(t_0) \end{bmatrix} = \begin{bmatrix} x(t_0) \\ x'(t_0) \end{bmatrix}.$$

The most famous second-order differential equation,

$$x''(t) = -x(t),$$

has the solution $x(t) = \alpha\cos(t) + \beta\sin(t)$, for constants α and β depending upon the initial values. Write the second-order equation as the system

$$\mathbf{x}'(t) = \begin{bmatrix} x_1'(t) \\ x_2'(t) \end{bmatrix} = \begin{bmatrix} x_2(t) \\ -x_1(t) \end{bmatrix}.$$

Combining Newton's inverse-square description of gravitational force with his Second Law leads to the system of second order ODEs

$$\mathbf{x}''(t) = \frac{-\mathbf{x}(t)}{\|\mathbf{x}(t)\|_2^3},$$

where $\mathbf{x} \in \mathbb{R}^3$ is a vector in Euclidean space, and the 2-norm denotes the usual (Euclidean) length of a vector,

$$\|\mathbf{x}(t)\|_2 = (x_1(t)^2 + x_2(t)^2 + x_3(t)^2)^{1/2}.$$

Since $\mathbf{x}(t) \in \mathbb{R}^3$, this second order equation reduces to a system of *six* first order equations.

5.1.4 *Picard's Theorem: Existence and Uniqueness of Solutions*

Before constructing numerical solutions to these differential equations, it is important to understand when solutions exist at all. Picard's theorem establishes existence and uniqueness.

Theorem 5.1 (Picard's Theorem)**.**
Let $f(t, x)$ be a continuous function on the rectangle

$$D = \{(t, x) : t \in [t_0, t_{\text{final}}], x \in [x_0 - c, x_0 + c]\}$$

for some fixed $c > 0$. Furthermore, suppose $|f(t, x_0)| \leq K$ for all $t \in [t_0, t_{\text{final}}]$, and suppose there exists some *Lipschitz constant* $L > 0$ such that

$$|f(t, u) - f(t, v)| \leq L\,|u - v|$$

For a proof, see Süli and Mayers, Section 12.1.

for all $u, v \in [x_0 - c, x_0 + c]$ and all $t \in [t_0, t_{\text{final}}]$. Finally, suppose that

$$c \geq \frac{K}{L}\Big(\mathrm{e}^{L(t_{\text{final}} - t_0)} - 1\Big).$$

(That is, the box D must be sufficiently large to compensate for large values of K and L.) Then there *exists* a *unique* $x \in C^1[t_0, t_{\text{final}}]$ such that $x(t_0) = x_0$, $x'(t) = f(t, x)$ for all $t \in [t_0, t_{\text{final}}]$, and $|x(t) - x_0| \leq c$ for all $t \in [t_0, t_{\text{final}}]$.

In simpler words, these hypotheses ensure the existence of a unique C^1 solution to the initial value problem, and this solution stays within the rectangle D for all $t \in [t_0, t_{\text{final}}]$, i.e., as t increases the solution will 'exit' from the right-side of the rectangle, not the top or bottom.

5.2 *One-step methods*

We are prepared to discuss some numerical methods to approximate the solution to all these ODEs. To simplify the notation, we present our methods in the context of the scalar equation

$$x'(t) = f(t, x(t))$$

with the initial condition $x(t_0) = x_0$. All the algorithms generalize trivially to systems: simply replace scalars with vectors.

When computing approximate solutions to the initial value problem, we will not obtain the solution for every value of $t > t_0$, but only on a discrete grid. In particular, we will generate approximate solutions at some regular grid of time steps

$$t_k = t_0 + kh$$

for some constant *step-size* h. (The methods we consider in this subsection allow h to change with each step size, so one actually has $t_k = t_{k-1} + h_k$. For simplicity of notation, we will assume for now that the step-size is fixed.)

The approximation to x at the time t_k is denoted by x_k, so hopefully

$$x_k \approx x(t_k).$$

Of course, the initial data is exact:

$$x_0 = x(t_0).$$

The field of *asymptotic analysis* delivers approximations in terms of elementary functions that can be highly accurate; these are typically derived in a non-numerical fashion, and often have the virtue of accurately identifying leading order behavior of complicated solutions. For a beautiful introduction to this important area of applied mathematics, see Carl M. Bender and Seven A. Orszag, *Advanced Mathematical Methods for Scientists and Engineers*; McGraw-Hill, 1978; Springer, 1999.

5.2.1 *Euler's method*

We need some approximation that will advance from the exact point on the solution curve, (t_0, x_0) to time t_1. From basic calculus we

know that

$$x'(t) = \lim_{h \to 0} \frac{x(t+h) - x(t)}{h}.$$

This definition of the derivative inspires our first method. Apply it at time t_0 with a small but finite time step $h > 0$ to obtain

$$x'(t_0) \approx \frac{x(t_0+h) - x(t_0)}{h}.$$

Since $x'(t_0) = f(t_0, x(t_0)) = f(t_0, x_0)$, we know the quantity on the left hand side of this approximation. The only quantity we don't know is $x(t_0 + h) = x(t_1)$. Rearrange the above to put $x(t_1)$ on the left hand side:

$$x(t_1) \approx x(t_0) + hx'(t_0) = x_0 + hf(t_0, x_0).$$

This approximation is precisely the one suggested by the direction field discussion in Section 5.1.1. There, to progress from the starting point (t_0, x_0), we followed the line of slope $f(t_0, x_0)$ some distance, which in the present context is our step size, h. To progress from the new point, (t_1, x_1), we follow a new slope, $f(t_1, x_1)$, giving the iteration

$$x_2 = x_1 + hf(t_1, x_1).$$

There is an important distinction here. Ideally, we would have derived our value of $x_2 \approx x(t_2)$ from the formula

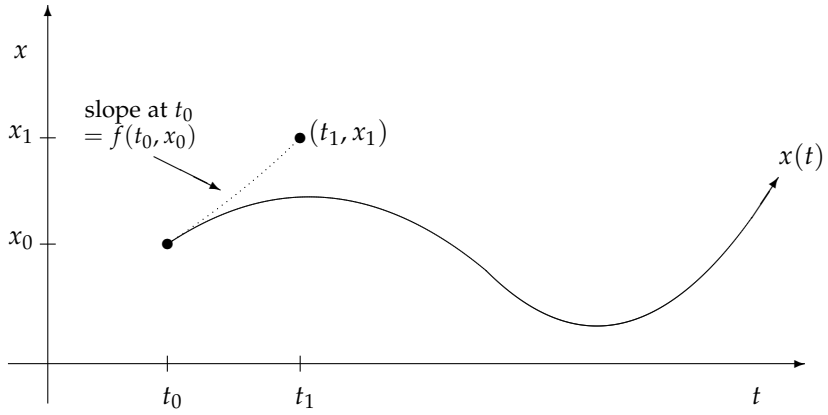
$$x(t_2) \approx x(t_1) + hf(t_1, x(t_1)).$$

However, an error was made in the computation of $x_1 \approx x(t_1)$; we do not know the exact value $x(t_1)$. Thus, we settle for computing x_2 from x_1, a quantity we do know. This might seem like a minor distinction, but in general the difference between the approximation x_k and the true solution $x(t_k)$ is vital. At each step, a *local error* is made due to the approximation of the derivative by a line. These local errors accumulate, giving a *global error*. Is a small local error enough to ensure small global error? This question is the subject of the next two lectures.

Given the approximation x_2, repeat the same procedure to obtain x_3, and so on. Formally,

Euler's Method: $$x_{k+1} = x_k + hf(t_k, x_k).$$

The first step of Euler's method is illustrated in the following schematic.



Example 5.3. Consider the performance of Euler's method on the two examples in Section 5.1.1. First, we examine the equation, $x'(t) = x(t)$, with initial condition $x(0) = 1$. We apply two step sizes: $h = 0.5$ and $h = 0.1$. Naturally, we expect that decreasing h will improve the local accuracy. But with $h = 0.1$, we require five times as many approximations as with $h = 0.5$. How do the errors made at these many steps accumulate? The plot below shows that both approximations underestimate the true solution, but as we expect, the smaller step size yields the better approximation – but requires more work to get to a fixed destination time.

Example 5.4. Next, consider the second example, $x'(t) = \sin(tx(t))$, this time with $x(0) = 5$. Since we do not know the exact solution, we can only compare approximate answers, here obtained with $h = 0.5$ and $h = 0.1$. For $t > 4$, the solutions completely differ from one another! Again, the smaller step size is the more accurate solution. In the plot below, the direction field is shown together with the approximate solutions. Note that $f(t, x) = \sin(tx)$ varies with x, so when the $h = 0.5$ solution diverges from the $h = 0.1$ solution, very different values of f are used to generate iterates. The $h = 0.5$ solution 'jumps' over the correct asymptote, and provides a very misleading answer.

Example 5.5. For a final example of Euler's method, consider the equation

$$x'(t) = 1 + x(t)^2$$

with $x(0) = 0$. This equation looks innocuous enough; indeed, you might notice that the exact solution is $x(t) = \tan(t)$. The true solution blows up *in finite time*, $x(t) \to \infty$ as $t \to \pi/2$. (Such blow-up behavior is common in ODEs and PDEs where the formula for the derivative of x involves higher powers of x.) It is reasonable to seek an approximate solution to the differential equation for $t \in [0, \pi/2)$, but beyond $t = \pi/2$, the equation does not have a solution, and any answer produced by our numerical method is, essentially, garbage.

This example is given in Kincaid and Cheney, page 525.

For any finite x, $f(t,x) = 1 + x^2$ will always be finite. Thus Euler's method,

$$\begin{aligned} x_{k+1} &= x_k + hf(t_k, x_k) \\ &= h + x_k(1 + hx_k) \end{aligned}$$

will always produce some finite quantity; it will never give the infinite answer at $t = \pi/2$. Still, as we see in the plots below, Euler's method captures the qualitative behavior well, with the iterates growing very large soon after $t = \pi/2$. (Notice that the vertical axis is logarithmic, so by $t = 2$, the approximation with time step $h = 0.05$ exceeds 10^{10}.)

LECTURE 33: *Local Analysis of One-Step Integrators*

What can be said of the error between the computed solution x_k at time $t_k = t_0 + kh$ and the exact solution $x(t_k)$? In this lecture and the next, we analyze this error, as a function of k, h, and properties of the ODE, for an important class of algorithms that generalize the forward Euler method.

5.2.2 *Runge–Kutta Methods*

To obtain increased accuracy in Euler's method,

$$x_{k+1} = x_k + hf(t_k, x_k),$$

one might naturally reduce the step-size, h. Since Euler's method derives from a first-order approximation to the derivative, we might expect the error to decay linearly in h. Before making this rigorous, first consider some better approaches: we are rarely satisfied with order-h accuracy! By improving upon Euler's method, we hope to obtain an improved solution while still maintaining a large time-step.

First consider a modification that might not look like such a big improvement: simply replace $f(t_k, x_k)$ by $f(t_{k+1}, x_{k+1})$ to obtain

$$x_{k+1} = x_k + hf(t_{k+1}, x_{k+1}),$$

called the *backward Euler method*. Because x_{k+1} depends on the value $f(t_{k+1}, x_{k+1})$, this scheme is called an *implicit method*; to compute x_{k+1}, one needs to solve a (generally nonlinear) system of equations, rather more involved than the simple update required for the forward Euler method.

At *each step*, one must find a zero of the function

$$G(x_{k+1}) = x_{k+1} - x_k - hf(t_{k+1}, x_{k+1})$$

using, for example Newton's method or the secant method. If h is small and f is not too wild, we might hope that we could get an initial guess $x_{k+1} \approx x_k$, or $x_{k+1} \approx x_k + hf(t_k, x_k)$. Note that this nonlinear iteration could require multiple evaluations of f to advance the backward Euler method by one time step.

One can improve on both Euler methods by averaging the updates they make to x_k:

$$x_{k+1} = x_k + \tfrac{1}{2}h\Big(f(t_k, x_k) + f(t_{k+1}, x_{k+1})\Big). \tag{5.1}$$

This method is the *trapezoid method*, for it can be derived by integrating the equation $x'(t) = f(t, x(t))$,

$$\int_{t_k}^{t_{k+1}} x'(t)\,\mathrm{d}t \;=\; \int_{t_k}^{t_{k+1}} f(t, x)\,\mathrm{d}t,$$

and approximating the integral on the right using the trapezoid rule. The fundamental theorem of calculus gives the exact formula for the integral on the left, $x(t_{k+1}) - x(t_k)$. Together, this gives

$$x(t_{k+1}) - x(t_k) \approx \frac{t_{k+1} - t_k}{2}\Big(f\big(t_k, x(t_k)\big) + f\big(t_{k+1}, x(t_{k+1})\big)\Big). \tag{5.2}$$

Replacing the inaccessible exact values $x(t_k)$ and $x(t_{k+1})$ with their approximations x_k and x_{k+1}, and using the time-step $h = t_{k+1} - t_k$, equation (5.2) suggests

$$x_{k+1} - x_k = \frac{h}{2}\big(f(t_k, x_k) + f(t_{k+1}, x_{k+1})\big).$$

Rearranging this equation gives the trapezoid method (5.1) for x_{k+1}.

Like the backward Euler method, the trapezoid rule is implicit, due to the $f(t_{k+1}, x_{k+1})$ term. To obtain a similar *explicit* method, replace x_{k+1} by its approximation from the explicit Euler method:

$$f(t_k + h, x_{k+1}) \approx f(t_k + h, x_k + hf(t_k, x_k)).$$

The result is called *Heun's method* or the *improved Euler method*:

$$x_{k+1} = x_k + \tfrac{1}{2}h\Big(f(t_k, x_k) + f(t_k + h, x_k + hf(t_k, x_k))\Big).$$

Note that this method can be implemented using only two evaluations of the function $f(t, x)$.

The *modified Euler method* takes a similar approach to Heun's method:

$$x_{k+1} = x_k + hf\big(t_k + \tfrac{1}{2}h, x_k + \tfrac{1}{2}hf(t_k, x_k)\big),$$

which also requires two f evaluations per step.

Additional function evaluations can deliver increasingly accurate explicit one-step methods, an important family of which are known as *Runge–Kutta methods*. In fact, the forward Euler and Heun methods are examples of one- and two-stage Runge–Kutta methods. The *four-stage Runge–Kutta method* is among the most famous one-step methods:

$$x_{k+1} = x_k + \tfrac{1}{6}h\Big(k_1 + 2k_2 + 2k_3 + k_4\Big),$$

One often encounters this method implemented as a subroutine called `RK4` in old FORTRAN codes.

where

$$\begin{aligned} k_1 &= f(t_k, x_k) \\ k_2 &= f(t_k + \tfrac{1}{2}h, x_k + \tfrac{1}{2}hk_1) \\ k_3 &= f(t_k + \tfrac{1}{2}h, x_k + \tfrac{1}{2}hk_2) \\ k_4 &= f(t_k + h, x_k + hk_3). \end{aligned}$$

We must address an important consideration: these more sophisticated methods might potentially give better approximations of the solution $x(t)$, but they require more evaluations of the function f per step than the forward Euler method. Many interesting applications give functions f that are expensive to evaluate. One must make a trade-off: methods with greater accuracy allow for larger time-step h, but require more function evaluations per time step. To understand the interplay between accuracy and computational expense, we require a more nuanced understanding of the convergence behavior of these various methods.

5.2.3 *Truncation Error*

All explicit one-step methods can be written in the general form

$$x_{k+1} = x_k + h\Phi(t_k, x_k; h).$$

Such methods incur two types of error:

1. The error due to the fact that even if the method was exact at t_k, the updated value x_{k+1} at t_{k+1} will not be exact. This is called *truncation error*, or *local error*.

2. In practice, the value x_k is not exact. How is this discrepancy, the fault of previous steps, magnified by the current step? This accumulated error is called *global error*.

Let us make these notions of error more precise. At every given time t_k, $k = 1, 2, \ldots$, we have some approximation x_k to the value $x(t_k)$. Denote the *global error* by

$$e_k := x(t_k) - x_k.$$

We seek to understand this error as a function of the step size h.

To analyze the global error e_k, first consider the approximations made at each iteration. In the last lecture, we saw that Euler's method made an error by approximating the derivative $x'(t_k)$ by a finite difference,

$$\frac{x(t_{k+1}) - x(t_k)}{h} \approx x'(t_k) = f(t_k, x(t_k)).$$

This type of error is made at every step. Generalize this error for all explicit one-step methods.

Definition 5.1. The *truncation error* of an explicit one-step ODE integrator is defined as

$$T_k = \frac{x(t_{k+1}) - x(t_k)}{h} - \Phi(t_k, x(t_k); h).$$

If $T_k \to 0$ as $h \to 0$, the method is *consistent*. If $T_k = \mathcal{O}(h^p)$, the method *has order-p truncation error.*

The key to understanding truncation error is to note that T_k is essentially just a rearranged version of the general one-step method, *except that the exact solutions $x(t_k)$ and $x(t_{k+1})$ have replaced the approximations x_k and x_{k+1}*. Thus, the truncation error can be regarded as a measure of the error the method would make in a single step if supplied with perfect data, $x(t_k)$.

Example 5.6. It is simple to compute T_k for the explicit Euler method:

$$\begin{aligned} T_k &= \frac{x(t_{k+1}) - x(t_k)}{h} - \Phi(t_k, x(t_k); h) \\ &= \frac{x(t_{k+1}) - x(t_k)}{h} - f(t_k, x(t_k)) \\ &= \frac{x(t_{k+1}) - x(t_k)}{h} - x'(t_k). \end{aligned}$$

This last substitution, $f(t_k, x(t_k)) = x'(t_k)$, is valid because f is evaluated at the exact solution $x(t_k)$. (Recall that in general, $f(t_k, x_k) \neq x'(t_k)$.) Assuming that $x(t) \in C^2[t_k, t_{k+1}]$, we can expand $x(t)$ in a Taylor series about $t = t_k$ to obtain

$$x(t_{k+1}) = x(t_k) + hx'(t_k) + \tfrac{1}{2}h^2 x''(\xi)$$

for some $\xi \in [t_k, t_{k+1}]$. Rearrange this to obtain a formula for $x'(t_k)$, and substitute it into the formula for T_k, yielding

$$\begin{aligned} T_k &= \frac{x(t_{k+1}) - x(t_k)}{h} - x'(t_k) \\ &= \frac{x(t_{k+1}) - x(t_k)}{h} - \frac{x(t_{k+1}) - x(t_k)}{h} + \tfrac{1}{2}hx''(\xi) \\ &= \tfrac{1}{2}hx''(\xi). \end{aligned}$$

Thus, the forward Euler method has truncation error $T_k = \mathcal{O}(h)$, so $T_k \to 0$ as $h \to 0$.

Similarly, one can find that Heun's method and the modified Euler's method both have $\mathcal{O}(h^2)$ truncation error, while the error for the four-stage Runge–Kutta method is $\mathcal{O}(h^4)$. Extrapolating from this data, one might expect that a method requiring m evaluations of f can deliver $\mathcal{O}(h^m)$ truncation error. Unfortunately, this is not true beyond $m = 4$, hence the fame of the four-stage Runge–Kutta method. All Runge–Kutta methods with $\mathcal{O}(h^5)$ truncation error require *at least six* evaluations of f.

As we will discuss later, the same function evaluations for higher order methods can be strategically combined to give two methods with different orders of accuracy. Comparing the estimates from two methods of different orders, one can estimate the error in the integration. Such estimates then allow one to adjust the time-step h on the fly during an integration to control the error.

Next we must address a fundamental question: Does $T_k \to 0$ as $h \to 0$ ensure global convergence, $e_k \to 0$, for each $k = 1, 2, \ldots$?

LECTURE 34: *Global Analysis of One-Step Integrators*

5.3 Global error analysis for explicit one-step methods

The last lecture addressed the truncation error, T_k, of a one-step method. Consistency (i.e., $T_k \to 0$ as $h \to 0$) is an obvious necessary condition for the *global error*

$$e_k = x(t_k) - x_k$$

to converge as $h \to 0$. In this lecture, we wish to understand this key question:

Is consistency sufficient for convergence of the global error as $h \to 0$?

As before, consider the general one step method

$$x_{k+1} = x_k + h\Phi(t_k, x_k; h)$$

where the choice of $\Phi(t_k, x_k; h)$ defines the specific algorithm. We can rearrange the formula for truncation error,

$$T_k = \frac{x(t_{k+1}) - x(t_k)}{h} - \Phi(t_k, x(t_k); h),$$

to obtain an expression for $x(t_{k+1})$,

$$x(t_{k+1}) = x(t_k) + h\Phi(t_k, x(t_k); h) + hT_k.$$

This formula is comparable to the one-step method itself,

$$x_{k+1} = x_k + h\Phi(t_k, x_k; h).$$

Combining these expressions gives a formula for the global error,

$$\begin{aligned} e_{k+1} &= x(t_{k+1}) - x_{k+1} \\ &= x(t_k) - x_k + h\Big(\Phi(t_k, x(t_k); h) - \Phi(t_k, x_k; h)\Big) + hT_k \\ &= e_k + h\Big(\Phi(t_k, x(t_k); h) - \Phi(t_k, x_k; h)\Big) + hT_k. \end{aligned}$$

Recall the example $x'(t) = 1 + x^2$ from Section 5.1. That equation blew up in finite time, while the iterates of Euler's method were always finite. This is disappointing: for some equations, we can essentially have infinite global error! Thus, to get a useful error bound, we must make an assumption that the ODE is well behaved. Suppose we are integrating our equation from t_0 to some fixed t_{final}. Then assume there exists a constant L_Φ, *depending on the equation, the time interval, and the particular method* (but not h), such that

$$|\Phi(t, u; h) - \Phi(t, v; h)| \leq L_\Phi |u - v|$$

for all $t \in [t_0, t_{\text{final}}]$ and all $u, v \in \mathbb{R}$. This assumption is closely related to the *Lipschitz condition* that plays an essential role in the theorem of existence of solutions given in Section 5.1. For 'nice' ODEs and reasonable methods Φ, this condition is not difficult to satisfy.

For example, for the forward Euler method, $L_\Phi = L$, where L is the usual Lipshitz constant for the ordinary differential equation.

This assumption is precisely what we need to bound the difference between $\Phi(t_k, x(t_k); h)$ and $\Phi(t_k, x_k; h)$ that appears in the formula for e_k. In particular, we now have

$$\begin{aligned}
|e_{k+1}| &= \Big| e_k + h\Big(\Phi(t_k, x(t_k); h) - \Phi(t_k, x_k; h)\Big) + hT_k \Big| \\
&\le |e_k| + h\Big|\Phi(t_k, x(t_k); h) - \Phi(t_k, x_k; h)\Big| + h|T_k| \\
&\le |e_k| + hL_\Phi |x(t_k) - x_k| + h|T_k| \\
&= |e_k| + hL_\Phi |e_k| + h|T_k| \\
&= |e_k|(1 + hL_\Phi) + h|T_k|.
\end{aligned}$$

Suppose we are interested in all iterates from x_0 up to x_n for some n. Then let T denote the magnitude of the maximum truncation error over all those iterates:

$$T := \max_{0 \le k \le n} |T_k|.$$

We now build up an expression for e_n iteratively:

$$\begin{aligned}
|e_0| &= |x(t+0) - x_0| = 0 \\
|e_1| &\le h|T_0| \le hT \\
|e_2| &\le |e_1|(1 + hL_\Phi) + h|T_1| \le hT(1 + hL_\Phi) + hT \\
|e_3| &\le |e_2|(1 + hL_\Phi) + h|T_2| \le hT(1 + hL_\Phi)^2 + hT(1 + hL_\Phi) + hT \\
&\vdots \\
|e_n| &\le hT \sum_{k=0}^{n-1} (1 + hL_\Phi)^k.
\end{aligned}$$

Notice that this bound for $|e_n|$ is a finite geometric series, and thus we have the convenient formula

$$\begin{aligned}
|e_n| &\le hT\Big(\frac{(1 + hL_\Phi)^n - 1}{(1 + hL_\Phi) - 1}\Big) \\
&= \frac{T}{L_\Phi}\big((1 + hL_\Phi)^n - 1\big) \\
&< \frac{T}{L_\Phi}\big(\mathrm{e}^{nhL_\Phi} - 1\big). \qquad (5.3)
\end{aligned}$$

Recall that

$$\mathrm{e}^{\alpha x} = 1 + \alpha x + \tfrac{1}{2}(\alpha x)^2 + \tfrac{1}{3!}(\alpha x)^3 + \cdots$$

and so, since $hL_\Phi > 0$,

$$1 + hL_\Phi < \mathrm{e}^{hL_\Phi}.$$

(This result and proof are given as Theorem 12.2 in the text by Süli and Mayers.)

There are two key lessons to be learned from this bound on $|e_n|$.

- Fix h. The bound suggests that $|e_n|$ can grow *exponentially* as n increases. As you continue your numerical integration indefinitely, the error can be quite severe, even for well-behaved differential equations.

- Focus attention on some fixed target time t_{final}, and consider time steps
$$h := \frac{t_{\text{final}} - t_0}{n},$$
so that $x_n \approx x(t_{\text{final}})$. As $n \to \infty$, note that $h \to 0$, and in this case $nh = t_{\text{final}} - t_0$ is fixed. Thus in the bound
$$|e_n| < \frac{T}{L_\Phi}\left(\mathrm{e}^{nhL_\Phi} - 1\right)$$
Since L_Φ is independent of the step size h, the term $\mathrm{e}^{nhL_\Phi} - 1$ is a constant, and $|e_n| = \mathcal{O}(T)$. Hence if the truncation error converges, $T \to 0$ as $h \to 0$, then the global error at t_{final} will also converge. Moreover, if $T_k = \mathcal{O}(h^p)$, then the global error at t_{final} will also be $\mathcal{O}(h^p)$. Remember this beautiful, vital fact: *the global error reduces at the same rate as the truncation error for one-step methods*!

The plots in Figure 5.4 confirm these observations. Again for the model problem $x'(t) = x(t)$ with $(t_0, x_0) = (0, 1)$, the figure shows the error for Euler's method ($T_k = \mathcal{O}(h)$), Heun's method ($T_k = \mathcal{O}(h^2)$), and the four-stage Runge–Kutta method ($T_k = \mathcal{O}(h^4)$) for $t \in [0, 10]$. Note the logarithmic scale of the vertical axes in these plots. As n increases, the error *grows exponentially* in all these cases. However, as h is reduced, the error gets smaller at all fixed times. The extent of this error reduction is what one would expect from the local truncation errors. All of the plots start with $h = 0.1$ (black dots). The other curves show the result of repeatedly cutting h in half, giving $h = 0.05$ (blue), $h = 0.025$ (red), $h = 0.0125$ (cyan), and $h = 0.00625$ (magenta).

- If T_k is proportional to h (forward Euler), then cutting h in half should scale T_k by $1/2$.

- If T_k is proportional to h^2 (Heun's method), then cutting h in half should scale T_k by $1/4$.

- If T_k is proportional to h^4 (four-stage Runge–Kutta), then cutting h in half should scale T_k by $1/16$.

5.4 *Adaptive Time-Step Selection*

One-step methods make it simple to change the time-step h at each iteration. For complicated nonlinear problems, it is quite natural that

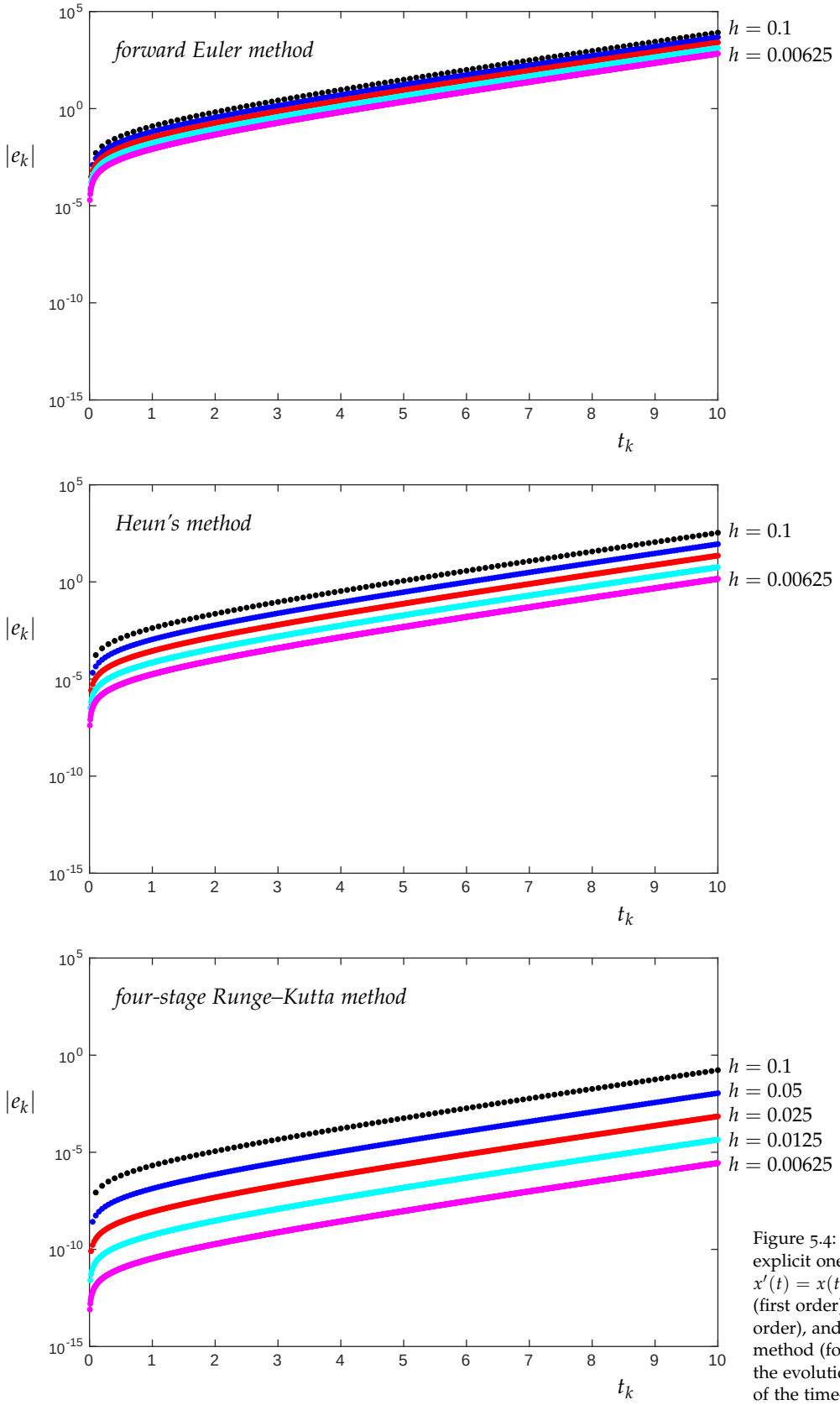


Figure 5.4: Global errors for three explicit one-step methods applied to $x'(t) = x(t)$: the forward Euler method (first order), Heun's method (second order), and the four-stage Runge–Kutta method (fourth order). Each plot shows the evolution of the error for five values of the time-step h. As k increases, the error grows *exponentially*. At a fixed t, the error reduces as h is reduced, at the rate given by the truncation error.

some regions (especially when x' is large) will merit a small time-step h, yet other regions, where there is less change in the solution, can easily be handled with a large value of h.

In the 1960s, Erwin Fehlberg suggested a beautiful way in which the step-size could be automatically adjusted at each step. There exist Runge–Kutta methods of order 4 and order 5 that can both be generated with the same *six* evaluations of f. (Recall that any fifth-order Runge–Kutta method requires at least six function evaluations.) First, we define the necessary f evaluations for this method:

$$
\begin{aligned}
k_1 &= f(t_k, x_k) \\
k_2 &= f(t_k + \tfrac{1}{4}h, x_k + \tfrac{1}{4}hk_1) \\
k_3 &= f(t_k + \tfrac{3}{8}h, x_k + \tfrac{3}{32}hk_1 + \tfrac{9}{32}hk_2) \\
k_4 &= f(t_k + \tfrac{12}{13}h, x_k + \tfrac{1932}{2197}hk_1 - \tfrac{7200}{2197}hk_2 + \tfrac{7296}{2197}hk_3) \\
k_5 &= f(t_k + h, x_k + \tfrac{439}{216}hk_1 - 8hk_2 + \tfrac{3680}{513}hk_3 - \tfrac{845}{4104}hk_4) \\
k_6 &= f(t_k + \tfrac{1}{2}h, x_k - \tfrac{8}{27}hk_1 + 2hk_2 - \tfrac{3544}{2565}hk_3 + \tfrac{1859}{4104}hk_4 - \tfrac{11}{40}hk_5).
\end{aligned}
$$

The following method has $\mathcal{O}(h^5)$ truncation error:

$$
x_{k+1} \;=\; x_k + h\Big(\tfrac{16}{135}k_1 + \tfrac{6656}{12825}k_3 + \tfrac{28561}{56430}k_4 - \tfrac{9}{50}k_5 + \tfrac{2}{55}k_6\Big).
$$

The f evaluations used to compute these k_j values can be combined in a different manner to obtain the following approximation, which only has $\mathcal{O}(h^4)$ truncation error:

$$
\widehat{x}_{k+1} \;=\; x_k + h\Big(\tfrac{25}{216}k_1 + \tfrac{1408}{2565}k_3 + \tfrac{2197}{4104}k_4 - \tfrac{1}{5}k_5\Big).
$$

Why would one be interested in an $\mathcal{O}(h^4)$ method when an $\mathcal{O}(h^5)$ approximation is available? By inspecting $x_{k+1} - \widehat{x}_{k+1}$, we can see how much the extra order of accuracy changes the solution. A significant difference signals that the step size h may be too large; software will react by reducing the step size before proceeding. This technology is implemented in MATLAB's `ode45` routine. (The `ode23` routine is similar, but based on a pair of second and third order methods.)

Another popular fifth-order method, designed by Cash and Karp (1990), uses six carefully chosen function evaluations that can be combined to also provide $\mathcal{O}(h)$, $\mathcal{O}(h^2)$, $\mathcal{O}(h^3)$, and $\mathcal{O}(h^4)$ approximations.

LECTURE 35: *Linear Multistep Methods: Truncation Error*

5.5 *Linear multistep methods: general form, local error*

One-step methods construct an approximate solution $x_{k+1} \approx x(t_{k+1})$ using only one previous approximation, x_k. This approach enjoys the virtue that the step size h can be changed at every iteration, if desired, thus providing a mechanism for error control. This flexibility comes at a price: For each order of accuracy in the truncation error, each step must perform at least one new evaluation of the derivative function $f(t, x)$. This might not sound particularly onerous, but in many practical problems, f evaluations are terribly time-consuming. A classic example is the N-body problem, which arises in models ranging from molecular dynamics to galactic evolution. Such models give rise to N coupled second order nonlinear differential equations, where the function f measures the forces between the N different particles. An evaluation of f requires $\mathcal{O}(N^2)$ arithmetic operations to compute, costly indeed when N is in the millions. Every f evaluation counts.

A landmark improvement to this N^2 approach, the *fast multipole method*, was developed by Leslie Greengard and Vladimir Rokhlin in the late 1980s.

One could potentially improve this situation by re-using f evaluations from previous steps of the ODE integrator, rather than always requiring f to be evaluated at multiple new (t, x) values at each step (as is the case, for example, with higher order Runge–Kutta methods). Consider the method

$$x_{k+1} = x_k + h\big(\tfrac{3}{2} f(t_k, x_k) - \tfrac{1}{2} f(t_{k-1}, x_{k-1})\big),$$

where h is the step size, $h = t_{k+1} - t_k = t_k - t_{k-1}$. Here x_{k+1} is determined from the two previous values, x_k and x_{k-1}. Unlike Runge–Kutta methods, f is not evaluated at points between t_k and t_{k+1}. Rather, *each step requires only one new f evaluation*, since $f(t_{k-1}, x_{k-1})$ would have been computed already at the previous step. Hence this method has roughly the same computational requirements as Euler's method, though soon we will see that its truncation error is $\mathcal{O}(h^2)$. The Heun and midpoint rules attained this same truncation error, but required *two* new f evaluations at each step.

Several drawbacks to this new scheme are evident: the step size is fixed throughout the iteration, and values for both x_0 and x_1 are needed before starting the method. The former concern can be addressed in practice through interpolation techniques. To handle the latter concern, initial data can be generated using a one-step method with small step size h. In some applications, including some problems in celestial mechanics, an asymptotic series expansion of the solution, accurate near $t \approx t_0$, can provide suitable initial data.

Step-size adjustment is possible, e.g., interpolating (t_{k-1}, x_{k-1}) and (t_k, x_k) to get intermediate values, but step-size adjustment still requires more care than with one-step methods.

5.5.1 General linear multistep methods

This section considers a general class of integrators known as *linear multistep methods*.

These notes on linear multistep methods draw heavily from the excellent presentation in Süli and Mayers, *Numerical Analysis: An Introduction*, Cambridge University Press, 2003.

Definition 5.2. general *m-step linear multistep method* has the form

$$\sum_{j=0}^{m} \alpha_j x_{k+j} = h \sum_{j=0}^{m} \beta_j f(t_{k+j}, x_{k+j}),$$

with $\alpha_m \neq 0$. If $\beta_m \neq 0$, then the formula for x_{k+m} involves x_{k+m} on the right hand side, so the method is *implicit*; otherwise, the method is *explicit*. A final convention requires $|\alpha_0| + |\beta_0| \neq 0$, for if $\alpha_0 = \beta_0 = 0$, then we actually have an $m-1$ step method masquerading as a m-step method. As f is only evaluated at (t_j, x_j), we adopt the abbreviation

$$f_j = f(t_j, x_j).$$

By this definition most Runge–Kutta methods, though one-step methods, *are not* multistep methods. Euler's method is an example of a one-step method that also fits this multistep template. Here are a few examples of linear multistep methods:

Euler's method:	$x_{k+1} - x_k = h f_k$	$\alpha_0 = -1, \alpha_1 = 1; \beta_0 = 1, \beta_1 = 0.$
Trapezoid rule:	$x_{k+1} - x_k = \frac{h}{2}(f_k + f_{k+1})$	$\alpha_0 = -1, \alpha_1 = 1; \beta_0 = \frac{1}{2}, \beta_1 = \frac{1}{2}.$
Adams–Bashforth:	$x_{k+2} - x_{k+1} = \frac{h}{2}(3f_{k+1} - f_k)$	$\alpha_0 = 0, \alpha_1 = -1, \alpha_2 = 1;$ $\beta_0 = -\frac{1}{2}, \beta_1 = \frac{3}{2}, \beta_2 = 0.$

The 'Adams–Bashforth' method presented above is the 2-step example of a broader class of *Adams–Bashforth* formulas. The 4-step Adams–Bashforth method takes the form

$$x_{k+4} = x_{k+3} + \frac{h}{24}\Big(55 f_{k+3} - 59 f_{k+2} + 37 f_{k+1} - 9 f_k\Big)$$

for which

$$\begin{aligned} &\alpha_0 = 0, \quad \alpha_1 = 0, \quad \alpha_2 = 0, \quad \alpha_3 = -1, \quad \alpha_4 = 1; \\ &\beta_0 = -\tfrac{9}{24}, \quad \beta_1 = \tfrac{37}{24}, \quad \beta_2 = -\tfrac{59}{24}, \quad \beta_3 = \tfrac{55}{24}, \quad \beta_4 = 0. \end{aligned}$$

The *Adams–Moulton* methods are a parallel class of *implicit* formulas. The 3-step version of this method is

$$x_{k+3} = x_{k+2} + \frac{h}{24}\Big(9 f_{k+3} + 19 f_{k+2} - 5 f_{k+1} + f_k\Big),$$

giving

$$\begin{aligned} &\alpha_0 = 0, \quad \alpha_1 = 0, \quad \alpha_2 = -1, \quad \alpha_4 = 1; \\ &\beta_0 = \tfrac{1}{24}, \quad \beta_1 = -\tfrac{5}{24}, \quad \beta_2 = \tfrac{19}{24}, \quad \beta_3 = \tfrac{9}{24}. \end{aligned}$$

5.5.2 *Truncation error for linear multistep methods*

Recall that the truncation error of one-step methods of the form $x_{k+1} = x_k + h\Phi(t_k, x_k; h)$ was given by

$$T_k = \frac{x(t_{k+1}) - x(t_k)}{h} - \Phi(t_k, x_k; h).$$

With general linear multistep methods is associated an analogous formula, based on substituting the exact solution $x(t_k)$ for the approximation x_k, and rearranging terms.

Definition 5.3. The truncation error for the linear multistep method

$$\sum_{j=0}^{m} \alpha_j x_{k+j} = h \sum_{j=0}^{m} \beta_j f(t_{k+j}, x_{k+j})$$

is given by the formula

$$T_k = \frac{\sum_{j=0}^{m} \left[\alpha_j \, x(t_{k+j}) - h \, \beta_j \, f(t_{k+j}, x(t_{k+j}))\right]}{h \sum_{j=0}^{m} \beta_j}.$$

The $\sum_{j=0}^{m} \beta_j$ term in the denominator is a normalization term; were it absent, multiplying the entire multistep formula by a constant would alter the truncation error, but the not the iterates x_k.

To get a simple form of the truncation error, we turn to Taylor series:

$$\begin{aligned}
x(t_{k+1}) = \quad x(t_k + h) &= x(t_k) + hx'(t_k) + \tfrac{h^2}{2!}x''(t_k) + \tfrac{h^3}{3!}x'''(t_k) + \tfrac{h^4}{4!}x^{(4)}(t_k) + \cdots \\
x(t_{k+2}) = \quad x(t_k + 2h) &= x(t_k) + 2hx'(t_k) + \tfrac{2^2h^2}{2!}x''(t_k) + \tfrac{2^3h^3}{3!}x'''(t_k) + \tfrac{2^4h^4}{4!}x^{(4)}(t_k) + \cdots \\
x(t_{k+3}) = \quad x(t_k + 3h) &= x(t_k) + 3hx'(t_k) + \tfrac{3^2h^2}{2!}x''(t_k) + \tfrac{3^3h^3}{3!}x'''(t_k) + \tfrac{3^4h^4}{4!}x^{(4)}(t_k) + \cdots \\
&\vdots \\
x(t_{k+m}) = x(t_k + mh) &= x(t_k) + mhx'(t_k) + \tfrac{m^2h^2}{2!}x''(t_k) + \tfrac{m^3h^3}{3!}x'''(t_k) + \tfrac{m^4h^4}{4!}x^{(4)}(t_k) + \cdots
\end{aligned}$$

and also

$$\begin{aligned}
f(t_{k+1}, x(t_{k+1})) = x'(t_k + h) &= x'(t_k) + hx''(t_k) + \tfrac{h^2}{2!}x'''(t_k) + \tfrac{h^3}{3!}x^{(4)}(t_k) + \cdots \\
f(t_{k+2}, x(t_{k+2})) = x'(t_k + 2h) &= x'(t_k) + 2hx''(t_k) + \tfrac{2^2h^2}{2!}x'''(t_k) + \tfrac{2^3h^3}{3!}x^{(4)}(t_k) + \cdots \\
f(t_{k+3}, x(t_{k+3})) = x'(t_k + 3h) &= x'(t_k) + 3hx''(t_k) + \tfrac{3^2h^2}{2!}x'''(t_k) + \tfrac{3^3h^3}{3!}x^{(4)}(t_k) + \cdots \\
&\vdots \\
f(t_{k+m}, x(t_{k+m})) = x'(t_k + mh) &= x'(t_k) + mhx''(t_k) + \tfrac{m^2h^2}{2!}x'''(t_k) + \tfrac{m^3h^3}{3!}x^{(4)}(t_k) + \cdots.
\end{aligned}$$

Substituting these expansions into the expression for T_k (eventually)

yields a convenient formula:

$$\Big(\sum_{j=0}^{m}\beta_j\Big)T_k = \frac{\sum_{j=0}^{m}\Big[\alpha_j\,x(t_{k+j}) - h\,\beta_j\,f(t_{k+j},x(t_{k+j}))\Big]}{h}$$

$$= h^{-1}\Big[\sum_{j=0}^{m}\alpha_j\Big]x(t_k) + \sum_{\ell=0}^{\infty}h^{\ell}\Big[\sum_{j=0}^{m}\Big(\alpha_j\frac{1}{(\ell+1)!}j^{\ell+1} - \beta_j\frac{1}{\ell!}j^{\ell}\Big)x^{(\ell+1)}(t_k)\Big]$$

$$= \frac{1}{h}\Big[\sum_{j=0}^{m}\alpha_j\Big]x(t_k) + \Big[\sum_{j=0}^{m}j\alpha_j - \sum_{j=0}^{m}\beta_j\Big]x'(t_k)$$

$$+ h\Big[\sum_{j=0}^{m}\frac{j^2}{2}\alpha_j - \sum_{j=0}^{m}j\beta_j\Big]x''(t_k)$$

$$+ h^2\Big[\sum_{j=0}^{m}\frac{j^3}{6}\alpha_j - \sum_{j=0}^{m}\frac{j^2}{2}\beta_j\Big]x'''(t_k)$$

$$+ h^3\Big[\sum_{j=0}^{m}\frac{j^4}{24}\alpha_j - \sum_{j=0}^{m}\frac{j^3}{6}\beta_j\Big]x^{(4)}(t_k) + \cdots.$$

In particular, the coefficient of the h^ℓ term is simply

$$\sum_{j=0}^{m}\frac{j^{\ell+1}}{(\ell+1)!}\alpha_j - \sum_{j=0}^{m}\frac{j^{\ell}}{\ell!}\beta_j$$

for all nonnegative integers ℓ.

Definition 5.4. A linear multistep method is *consistent* if $T_k \to 0$ as $h \to 0$.

The formula for T_k. gives an easy condition to check the consistency of a method.

Theorem 5.2. An m-step linear multistep method of the form

$$\sum_{j=0}^{m}\alpha_j x_{k+j} = h\sum_{j=0}^{m}\beta_j f_{k+j}$$

is consistent if and only if

$$\sum_{j=0}^{m}\alpha_j = 0 \qquad \text{and} \qquad \sum_{j=0}^{m}j\alpha_j = \sum_{j=0}^{m}\beta_j.$$

If one of the conditions in Theorem 5.2 are violated, then the formula for the truncation error contains either a term that grows like $1/h$ or remains constant as $h \to 0$. The Taylor analysis of the truncation error yields even more information, though: inspecting the coefficients multiplying h, h^2, etc. reveals easy conditions for determining the overall truncation error of a linear multistep method.

Definition 5.5. A linear multistep method is *order-p accurate* if $T_k = \mathcal{O}(h^p)$ as $h \to 0$.

Theorem 5.3. An m-step linear multistep method is order-p accurate if and only if it is consistent and

$$\sum_{j=0}^{m} \frac{j^{\ell+1}}{(\ell+1)!}\alpha_j = \sum_{j=0}^{m} \frac{j^{\ell}}{\ell!}\beta_j$$

for all $\ell = 1, \ldots, p-1$.

The next few examples show this theorem in action, applying it to some of the linear multistep methods discussed earlier.

Example 5.7 (Forward Euler method)**.**

$$\alpha_0 = -1, \alpha_1 = 1; \beta_0 = 1, \beta_1 = 0.$$

Clearly $\alpha_0 + \alpha_1 = -1 + 1 = 0$ and $(0\alpha_0 + 1\alpha_1) - (\beta_0 + \beta_1) = 0$. Thus the method is consistent.

When analyzed as a one-step method, the forward Euler method had truncation error $T_k = \mathcal{O}(h)$. The same result should hold when we analyze the algorithm as a linear multistep method. Indeed,

$$\left(\tfrac{1}{2}0^2\alpha_0 + \tfrac{1}{2}1^2\alpha_1\right) - \left(0\beta_0 + 1\beta_1\right) = \tfrac{1}{2} \neq 0.$$

Thus, $T_k = \mathcal{O}(h)$.

Example 5.8 (Trapezoid method)**.**

$$\alpha_0 = -1, \alpha_1 = 1; \beta_0 = \frac{1}{2}, \beta_1 = \frac{1}{2}.$$

Again, consistency is easy to verify: $\alpha_0 + \alpha_1 = -1 + 1 = 0$ and $(0\alpha_0 + 1\alpha_1) - (\beta_0 + \beta_1) = 1 - 1 = 0$. Furthermore,

$$\left(\tfrac{1}{2}0^2\alpha_0 + \tfrac{1}{2}1^2\alpha_1\right) - (0\beta_0 + 1\beta_1) = \tfrac{1}{2} - \tfrac{1}{2} = 0,$$

so $T_k = \mathcal{O}(h^2)$, but

$$\left(\tfrac{1}{6}0^3\alpha_0 + \tfrac{1}{6}1^3\alpha_1\right) - \left(\tfrac{1}{2}0^2\beta_0 + \tfrac{1}{2}1^2\beta_1\right) = \tfrac{1}{6} - \tfrac{1}{4} \neq 0,$$

so the trapezoid rule is not third order accurate: $T_k = \mathcal{O}(h^2)$.

Example 5.9 (2-step Adams–Bashforth)**.**

$$\begin{aligned} &\alpha_0 = 0, && \alpha_1 = -1, && \alpha_2 = 1; \\ &\beta_0 = -\tfrac{1}{2}, && \beta_1 = \tfrac{3}{2}, && \beta_2 = 0. \end{aligned}$$

Does this explicit 2-step method deliver $\mathcal{O}(h^2)$ accuracy, like the (implicit) Trapezoid method? Consistency follows easily:

$$\alpha_0 + \alpha_1 + \alpha_2 = 0 - 1 + 1 = 0$$

and

$$(0\alpha_0 + 1\alpha_1 + 2\alpha_2) - (\beta_0 + \beta_1) = 1 - 1 = 0.$$

The second order condition is also satisfied,

$$\left(\tfrac{1}{2}0^2\alpha_0 + \tfrac{1}{2}1^2\alpha_1 + \tfrac{1}{2}2^2\alpha_2\right) - \left(0\beta_0 + 1\beta_1\right) = \tfrac{3}{2} - \tfrac{3}{2} = 0,$$

but not the third order,

$$\left(\tfrac{1}{6}0^3\alpha_0 + \tfrac{1}{6}1^3\alpha_1 + \tfrac{1}{6}2^3\alpha_2\right) - \left(\tfrac{1}{2}0^2\beta_0 + \tfrac{1}{2}1^2\beta_1\right) = \tfrac{7}{6} - \tfrac{3}{4} \neq 0.$$

Thus $T_k = \mathcal{O}(h^2)$: the method is second order.

Example 5.10 (4-step Adams–Bashforth)**.**

$$\begin{array}{lllll} \alpha_0 = 0, & \alpha_1 = 0, & \alpha_2 = 0, & \alpha_3 = -1, & \alpha_4 = 1; \\ \beta_0 = -\frac{9}{24}, & \beta_1 = \frac{37}{24}, & \beta_2 = -\frac{59}{24}, & \beta_3 = \frac{55}{24}, & \beta_4 = 0. \end{array}$$

Consistency holds, since $\sum \alpha_j = -1 + 1 = 0$ and

$$\sum_{j=0}^{4} j\alpha_j - \sum_{j=0}^{4} \beta_j = \left(3(-1) + 4(1)\right) - \left(\tfrac{9}{24} + \tfrac{37}{24} - \tfrac{47}{24} + \tfrac{55}{24}\right) = 1 - 1 = 0.$$

The coefficients of h, h^2, and h^3 in the expansion for T_k all vanish:

$$\sum_{j=0}^{4} \tfrac{1}{2}j^2\alpha_j - \textstyle\sum_{j=0}^{4} j\beta_j = \left(\tfrac{3^2}{2}(-1) + \tfrac{4^2}{2}(1)\right) - \left(0(-\tfrac{9}{24}) + 1(\tfrac{37}{24}) + 2(-\tfrac{59}{24}) + 3(\tfrac{55}{24})\right) = \tfrac{7}{2} - \tfrac{84}{24} = 0;$$

$$\sum_{j=0}^{4} \tfrac{1}{6}j^3\alpha_j - \textstyle\sum_{j=0}^{4} \tfrac{1}{2}j^2\beta_j = \left(\tfrac{3^3}{6}(-1) + \tfrac{4^3}{6}(1)\right) - \left(\tfrac{1^2}{2}(\tfrac{37}{24}) + \tfrac{2^2}{2}(-\tfrac{59}{24}) + \tfrac{3^2}{2}(\tfrac{55}{24})\right) = \tfrac{37}{6} - \tfrac{148}{24} = 0;$$

$$\sum_{j=0}^{4} \tfrac{1}{24}j^4\alpha_j - \textstyle\sum_{j=0}^{4} \tfrac{1}{6}j^3\beta_j = \left(\tfrac{3^4}{24}(-1) + \tfrac{4^4}{24}(1)\right) - \left(\tfrac{1^3}{6}(\tfrac{37}{24}) + \tfrac{2^3}{6}(-\tfrac{59}{24}) + \tfrac{3^3}{6}(\tfrac{55}{24})\right) = \tfrac{175}{24} - \tfrac{1050}{144} = 0.$$

However, the $\mathcal{O}(h^4)$ term is not eliminated:

$$\sum_{j=0}^{4} \tfrac{1}{120}j^5\alpha_j - \textstyle\sum_{j=0}^{4} \tfrac{1}{24}j^4\beta_j = \left(\tfrac{3^5}{120}(-1) + \tfrac{4^5}{120}(1)\right) - \left(\tfrac{1^4}{24}(\tfrac{37}{24}) + \tfrac{2^4}{24}(-\tfrac{59}{24}) + \tfrac{3^4}{24}(\tfrac{55}{24})\right) = \tfrac{1267}{120} - \tfrac{887}{144} \neq 0.$$

Thus $T_k = \mathcal{O}(h^4)$: the method is fourth order.

A similar computation establishes fourth-order accuracy for the 4-step (implicit) Adams–Moulton formula

$$x_{k+3} = x_{k+2} + \tfrac{1}{24}h\Big(9f_{k+3} + 19f_{k+2} - 5f_{k+1} + f_k\Big).$$

In the last lecture, we proved that *consistency implies convergence* for one-step methods. Essentially, provided the differential equation is sufficiently well-behaved (in the sense of Picard's Theorem), then the numerical solution produced by a consistent one-step method on the fixed interval $[t_0, t_{\text{final}}]$ will converge to the true solution as $h \to 0$. Of course, this is a key property that we hope is shared by multistep methods.

Whether this is true for general linear multistep methods is the subject of the next lecture. For now, we merely present some computational evidence that, for certain methods, the global error at t_{final} behaves in the same manner as the truncation error.

Consider the model problem $x'(t) = x(t)$ for $t \in [0, 1]$ with $x(0) = x_0 = 1$, which has the exact solution is $x(t) = \mathrm{e}^t$. We shall approximate this solution using Euler's method, the second-order Adams–Bashforth formula, and the fourth-order Adams–Bashforth formula. The latter two methods require data not only at $t = 0$, but also at several additional values, $t = h$, $t = 2h$, and $t = 3h$. For this simple experiment, we can use the value of the exact solution, $x_1 = x(t_1)$, $x_2 = x(t_2)$, and $x_3 = x(t_3)$.

We close by offering evidence that there is more to the analysis of linear multistep methods than truncation error. Here are two explicit methods that are both second order:

$$x_{k+2} - \tfrac{3}{2}x_{k+1} + \tfrac{1}{2}x_k = h(\tfrac{5}{4}f_{k+1} - \tfrac{3}{4}f_k)$$

$$x_{k+2} - 3x_{k+1} + 2x_k = h(\tfrac{1}{2}f_{k+1} - \tfrac{3}{2}f_k).$$

We apply these methods to the model problem $x'(t) = x(t)$ with $x(0) = 1$, with exact initial data $x_0 = 1$ and $x_1 = e^h$. The results of these two methods are shown below. The first method tracks the exact solution $x(t) = e^t$ very nicely. The second method, however, shows a disturbing property: while it matches up quite well for the initial steps, it soon starts to fall far from the solution. Why does this second-order method do so poorly for such a simple problem? Does this reveal a general problem with linear multistep methods? If not, how do we identify such ill-mannered methods?

LECTURE 36: *Linear Multistep Mehods: Zero Stability*

5.6 Linear multistep methods: zero stability

Does consistency imply convergence for linear multistep methods? This is always the case for one-step methods, as proved in section 5.3, but the example at the end of the last lecture suggests the issue is less straightforward for multistep methods. By understanding the subtleties, we will come to appreciate one of the most significant themes in numerical analysis: *stability* of discretizations.

We are interested in the behavior of linear multistep methods as $h \to 0$. In this limit, the right hand side of the formula for the generic multistep method,

$$\sum_{j=0}^{m} \alpha_j x_{k+j} = h \sum_{j=0}^{m} \beta_j f(t_{k+j}, x_{k+j}),$$

makes a negligible contribution. This motivates our consideration of the trivial model problem $x'(t) = 0$ with $x(0) = 0$. Does the linear multistep method recover the exact solution, $x(t) = 0$?

When $x'(t) = 0$, clearly we have $f_{k+j} = 0$ for all j. The condition $\alpha_m \neq 0$ allows us to write

$$x_m = -\frac{(\alpha_0 x_0 + \alpha_1 x_1 + \cdots + \alpha_{m-1} x_{m-1})}{\alpha_m}$$

Hence if the method is started with exact data

$$x_0 = x_1 = \cdots = x_{m-1} = 0,$$

then

$$x_m = -\frac{(\alpha_0 \cdot 0 + \alpha_1 \cdot 0 + \cdots + \alpha_{m-1} \cdot 0)}{\alpha_m} = 0,$$

and this pattern will continue: $x_{m+1} = 0$, $x_{m+2} = 0, \ldots$. Any linear multistep method with exact starting data produces the exact solution for this special problem, regardless of the time-step.

Of course, for more complicated problems it is unusual to have exact starting values $x_1, x_2, \ldots x_{m-1}$; typically, these values are only approximate, obtained from some high-order one-step ODE solver or from an asymptotic expansion of the solution that is accurate in a neighborhood of t_0. To discover how multistep methods behave, we must first understand how these errors in the initial data pollute future iterations of the linear multistep method.

Definition 5.6. Suppose the initial value problem $x'(t) = f(t, x)$, $x(t_0) = x_0$ satisfies the requirements of Picard's Theorem over the

interval $[t_0, t_{\text{final}}]$. For an m-step linear multistep method, consider two sequences of starting values for a fixed time-step h

$$\{x_0, x_1, \ldots, x_{m-1}\} \quad \text{and} \quad \{\widehat{x}_0, \widehat{x}_1, \ldots, \widehat{x}_{m-1}\},$$

that generate the approximate solutions $\{x_j\}_{j=0}^n$ and $\{\widehat{x}_j\}_{j=0}^n$, where $t_n = t_{\text{final}}$. The multistep method is *zero-stable* for this initial value problem if for sufficiently small h there exists some constant M (independent of h) such that

$$|x_k - \widehat{x}_k| \leq M \max_{0 \leq j \leq m-1} |x_j - \widehat{x}_j|$$

for all k with $t_0 \leq t_k \leq t_{\text{final}}$. More plainly, a method is zero-stable for a particular problem if errors in the starting values are not magnified in an unbounded fashion.

Proving zero-stability directly from this definition would be a chore. Fortunately, there is an easy way to check zero stability all at once *for all sufficiently 'nice' differential equations*. To begin with, consider a particular example.

Example 5.11 (A novel second order method)**.** The truncation error formulas from the last lecture can be used to derive a variety of linear multistep methods that satisfy a given order of truncation error. You can use those conditions to verify that the explicit two-step method

$$x_{k+2} = 2x_k - x_{k+1} + h(\tfrac{1}{2} f_k + \tfrac{5}{2} f_{k+1}) \tag{5.4}$$

is second order accurate. Now we will test the zero-stability of this algorithm on the trivial model problem, $x'(t) = 0$ with $x(0) = 0$. Since $f(t, x) = 0$ in this case, the method reduces to

$$x_{k+2} = 2x_k - x_{k+1}.$$

As seen above, this method produces the exact solution if given exact initial data, $x_0 = x_1 = 0$. But what if $x_0 = 0$ but $x_1 = \varepsilon$ for some small $\varepsilon > 0$? This method produces the iterates

$$\begin{aligned}
x_2 &= 2x_0 - x_1 = 2 \cdot 0 - \varepsilon = -\varepsilon \\
x_3 &= 2x_1 - x_2 = 2(\varepsilon) - (-\varepsilon) = 3\varepsilon \\
x_4 &= 2x_2 - x_3 = 2(-\varepsilon) - 3\varepsilon = -5\varepsilon \\
x_5 &= 2x_3 - x_4 = 2(3\varepsilon) - (-5\varepsilon) = 11\varepsilon \\
x_6 &= 2x_4 - x_5 = 2(-5\varepsilon) - (11\varepsilon) = -21\varepsilon \\
x_7 &= 2x_5 - x_6 = 2(11\varepsilon) - (-21\varepsilon) = 43\varepsilon \\
x_8 &= 2x_6 - x_7 = 2(-21\varepsilon) - (43\varepsilon) = 85\varepsilon.
\end{aligned}$$

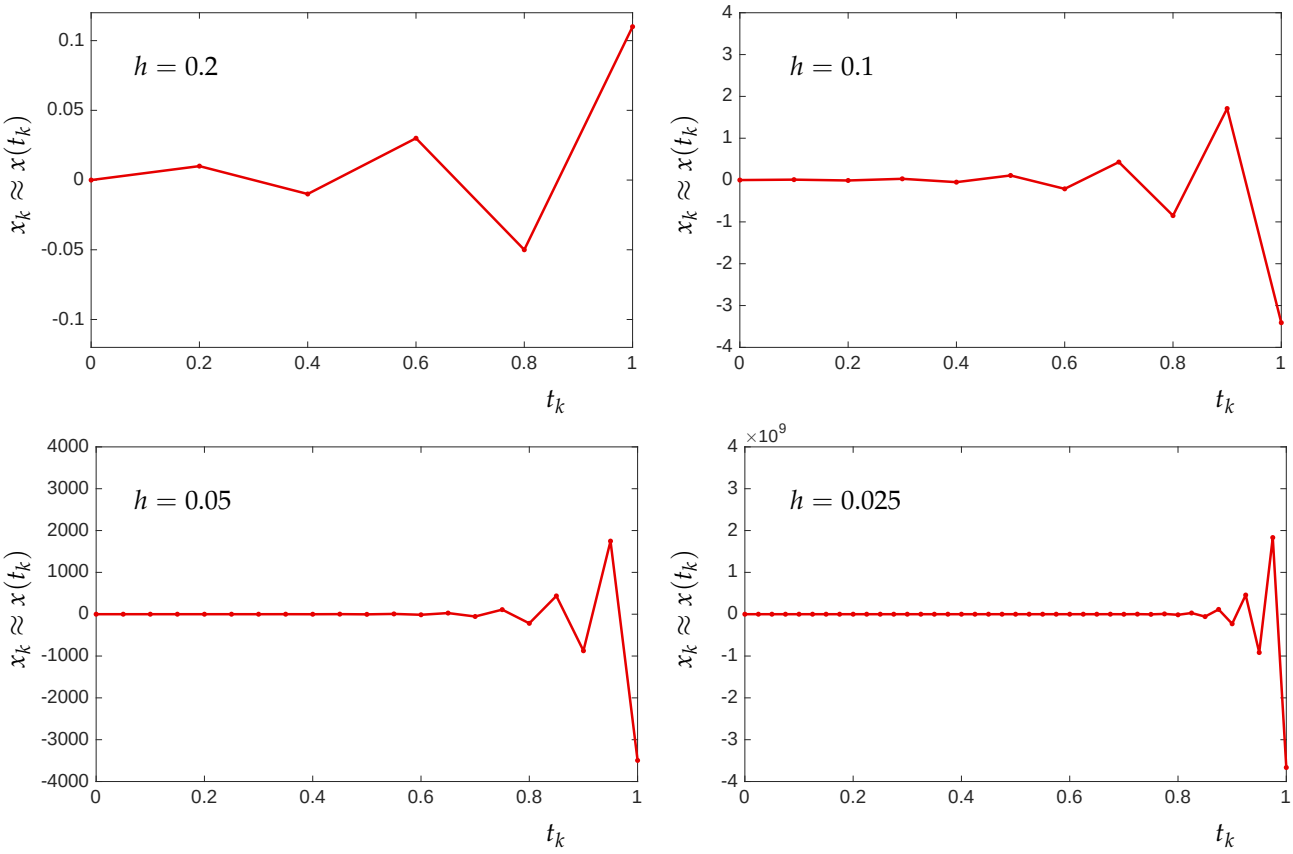


Figure 5.5: Approximate solutions to $x'(t) = 0$ using the novel second-order method (5.4) with initial data $x_0 = 0$ and $x_1 = 0.01$. As h gets smaller, the method produces approximations to the true solution $x(t) = 0$ of exponentially degrading quality! (Note how the scale of the vertical axis grows from plot to plot.)

In just seven steps, the error has been multiplied 85-fold. The error is roughly doubling at each step, and before long the approximate 'solution' is complete garbage. Figure 5.5 shows this instability, plotting x_k for four different values of h and $\varepsilon = 0.01$.

This examples illustrates another quirk. When applied to this particular model problem, the linear multistep method reduces to $\sum_{j=0}^{m} \alpha_j x_{k+j} = 0$, and thus never incorporates the time-step, h. Hence *the error at some fixed time* $t_{\text{final}} = hk$ *gets worse as* h *gets smaller and* k *grows accordingly*! Figure 5.6 puts all four of solutions from Figure 5.5 together in one plot, dramatically illustrating how the solutions degrade *as* h *gets smaller*!

Though this method has second-order local (truncation) error, it blows up if fed incorrect initial data for x_1. Decreasing h can magnify this effect, even if, for example, the error in x_1 is proportional to h. We can draw a larger lesson from this simple problem: For linear multistep methods, consistency (i.e., $T_k \to 0$ as $h \to 0$) is *not sufficient* to ensure convergence.

Let us analyze our unfortunate method a little more carefully. Setting the starting values x_0 and x_1 aside for the moment, we want to

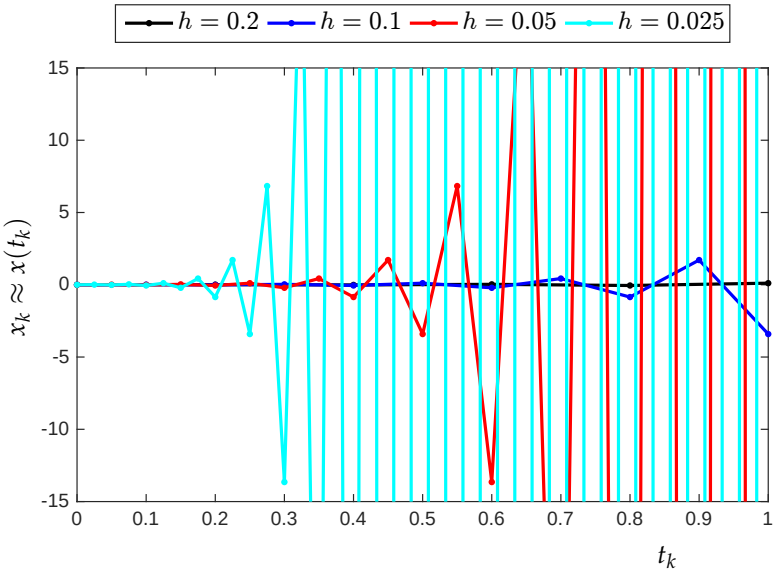


Figure 5.6: All four solutions from Figure 5.5 plotted together, to illustrate how the approximate solutions from the method (5.4) to $x'(t) = 0$ with $x_0 = 0$ and $x_1 = \varepsilon$ degrade as $h \to 0$.

find all sequences $\{x_j\}_{j=0}^{\infty}$ that satisfy the linear, constant-coefficient recurrence relation

$$x_{k+2} = 2x_k - x_{k+1}.$$

Since the x_k values grew exponentially in the example above, assume that this recurrence has a solution of the form $x_k = \gamma^k$ for all $k = 0, 1, \ldots$, where γ is some number that we will try to determine. Plug this *ansatz* for x_k into the recurrence relation to see if you can make it work as a solution:

$$\gamma^{k+2} = 2\gamma^k - \gamma^{k+1}.$$

Divide this equation through by γ^k to obtain the quadratic equation

$$\gamma^2 = 2 - \gamma.$$

If γ solves this quadratic, then the putative solution $x_k = \gamma^k$ indeed satisfies the difference equation. Since

$$\gamma^2 + \gamma - 2 = (\gamma + 2)(\gamma - 1)$$

the roots of this quadratic are simply $\gamma = -2$ and $\gamma = 1$. Thus we expect solutions of the form $x_k = (-2)^k$ and the less interesting $x_k = 1^k = 1$.

If $x_k = \gamma_1^k$ and $x_k = \gamma_2^k$ are both solutions of the recurrence, then $x_k = A\gamma_1^k + B\gamma_2^k$ is also a solution, for any real numbers A and B. To see this, note that

$$\gamma_1^2 + \gamma_1 - 2 = \gamma_2^2 + \gamma_2 - 2 = 0,$$

and so

$$A\gamma_1^k\left(\gamma_1^2 + \gamma_1 - 2\right) = B\gamma_2^k\left(\gamma_2^2 + \gamma_2 - 2\right) = 0.$$

Rearranging this equation,

$$A\gamma_1^{k+2} + B\gamma_1^{k+2} = 2(A\gamma_1^k + B\gamma_1^k) - (A\gamma_1^{k+1} + B\gamma_1^{k+1}),$$

which implies that $x_k = A\gamma_1^k + B\gamma_2^k$ is a solution to the recurrence.

In fact, this is the general form of a solution to our recurrence. For any starting values x_0 and x_1, one can determine the associated constants A and B. For example, with $\gamma_1 = -2$ and $\gamma_2 = 1$, the initial conditions $x_0 = 0$ and $x_1 = \varepsilon$ require that

$$\begin{aligned} A + B &= 0 \\ -2A + B &= \varepsilon, \end{aligned}$$

which implies

$$A = -\varepsilon/3, \qquad B = \varepsilon/3.$$

Indeed, the solution

$$x_k = \frac{\varepsilon}{3} - \frac{\varepsilon}{3}(-2)^k \tag{5.5}$$

generates the iterates $x_0 = 0$, $x_1 = \varepsilon$, $x_2 = -\varepsilon$, $x_3 = 3\varepsilon$, $x_4 = -5\varepsilon$, ... computed previously. Notice that (5.5) reveals *exponential* growth with k: this growth overwhelms algebraic improvements in the estimate x_1 that might occur as we reduce h. For example, if $\varepsilon = x_1 - x(t_0 + h) = ch^p$ for some constant c and $p \geq 1$, then $x_k = ch^p(1 - (-2)^k)/3$ still grows exponentially in k.

5.6.1 The Root Condition

The real trouble with the previous method was that the formula for x_k involves the term $(-2)^k$. Since $|-2| > 1$, this component of x_k grows exponentially in k. This term is simply an artifact of the finite difference equation, and has nothing to do with the underlying differential equation. As k increases, this $(-2)^k$ term swamps the other term in the solution. It is called a *parasitic solution*.

Let us review how we determined the general form of the solution. We assumed a solution of the form $x_k = \gamma^k$, then plugged this solution into the recurrence $x_{k+2} = 2x_k - x_{k+1}$. The possible values for γ were roots of the equation $\gamma^2 = 2 - \gamma$.

The process we just applied to one specific linear multistep method can readily be extended to the general linear multistep method

$$\sum_{j=0}^{m} \alpha_j x_{k+j} = h \sum_{j=0}^{m} \beta_j f(t_{k+j}, x_{k+j}). \tag{5.6}$$

by applying the method again to the trivial equation $x'(t) = 0$. For this special equation, the method (5.6) reduces to

$$\sum_{j=0}^{m} \alpha_j x_{k+j} = 0.$$

Substituting $x_k = \gamma^k$ yields

$$\sum_{j=0}^{m} \alpha_j \gamma^{k+j} = 0.$$

Canceling γ^k,

$$\sum_{j=0}^{m} \alpha_j \gamma^j = 0.$$

Definition 5.7. The *characteristic polynomial* of an m-step linear multistep method is the degree-m polynomial

$$\rho(z) = \sum_{j=0}^{m} \alpha_j z^j.$$

For $x_k = \gamma^k$ to be a solution to the above recurrence, γ must be a root of the characteristic polynomial, $\rho(\gamma) = 0$. Since the characteristic polynomial has degree m, it will have m roots. If these roots are distinct, call them $\gamma_1, \gamma_2, \ldots, \gamma_m$, the general form of the solution of

$$\sum_{j=0}^{m} \alpha_j x_{k+j} = 0$$

is

$$x_k = c_1 \gamma_1^k + c_2 \gamma_2^k + \cdots c_m \gamma_m^k.$$

for constants $c_1, \ldots, c_m$ that are determined from the starting values $x_0, \ldots, x_m$.

If some root, say γ_1 is repeated p times, then instead of contributing the term $c_1\gamma_1^k$ to the general solution, it will contribute a term of the form $c_{1,1}\gamma_1^k + c_{1,2}k\gamma_1^k + \cdots + c_{1,p}k^{p-1}\gamma_1^k$.

To avoid parasitic solutions to a linear multistep method, all the roots of the characteristic polynomial should be located within the unit disk in the complex plane, i.e., $|\gamma_j| \leq 1$ for all $j = 1 \ldots, m$. Thus, for the simple differential equation $x'(t) = 0$, we have found a way to describe zero stability: Initial errors will not be magnified if the characteristic polynomial has all its roots in the unit disk; any roots on the unit disk should be simple (i.e., not multiple).

Following on from the previous marginal note, we note that if some root, say γ_1, is a repeated root on the unit circle, $|\gamma_1| = 1$, then the general solution will have terms like $k\gamma_1^k$, so $|k\gamma_1^k| = k|\gamma_1|^k = k$. While this term will not grow exponentially in k, it does grow algebraically, and errors will still grow enough as $h \to 0$ to violate zero stability.

This analysis might seem too trivial: after all, the problem $x'(t) = 0$ is not particularly interesting. What is remarkable is that, in a sense, if the linear multistep method is zero stable for $x'(t) = 0$, the one can prove it is zero stable *for all well-behaved differential equations*! This result was discovered in the late 1950s by Germund Dahlquist.

Theorem 5.4. A linear multistep method is zero-stable for any 'well-behaved' initial value problem provided it satisfies the *root condition*:

- all roots of $\rho(\gamma) = 0$ lie in the unit disk, i.e., $|\gamma| \leq 1$;
- any roots on the unit circle ($|\gamma| = 1$) are simple (i.e., not multiple).

An excellent discussion of this theoretical material is given in Süli and Mayers, *Numerical Analysis: An Introduction*, Cambridge University Press, 2003; these notes follow, in part, their exposition. See also E. Hairer, S. P. Nørsett, and G. Wanner, *Solving Ordinary Differential Equations I: Nonstiff Problems*, 2nd ed., Springer-Verlag, 1993.

One can see now where the term *zero-stability* comes from: it is necessary and sufficient for the stability definition to hold for the differential equation $x'(t) = 0$. In recognition of the discoverer of this key result, zero-stability is sometimes called *Dahlquist stability*. (Another synonymous term is *root stability*.) In addition to making this beautiful characterization, Dahlquist also answered the question about the conditions necessary for a multistep method to be convergent.

Theorem 5.5 (Dahlquist Equivalence Theorem)**.**
Suppose an m-step linear multistep method applied to a 'well-behaved' initial value problem on $[t_0, t_{\text{final}}]$ with consistent starting values,

$$x_k \to x(t_k) \quad \text{for } t_k = t_0 + hk,\ k = 0, \ldots, m-1$$

as $h \to 0$. This method is convergent, i.e.,

$$x_{\lceil (t-t_0)/h \rceil} \to x(t) \quad \text{for all } t \in [t_0, t_{\text{final}}].$$

as $h \to 0$ if and only if the method is *consistent* and *zero-stable*.

If the exact solution is sufficiently smooth, $x(t) \in C^{p+1}[t_0, t_{\text{final}}]$ and the multistep method is order-p accurate ($T_k = \mathcal{O}(h^p)$), then

$$x(t_k) - x_k = \mathcal{O}(h^p)$$

for all $t_k \in [t_0, t_{\text{final}}]$.

Dahlquist also characterized the maximal order of convergence for a zero-stable m-step multistep method.

Theorem 5.6 (First Dahlquist Stability Barrier)**.**
A zero-stable m-step linear multistep method has truncation error no better than

- $\mathcal{O}(h^{m+1})$ if m is odd
- $\mathcal{O}(h^m)$ if m is even.

Example 5.12 (A method on the brink of stability)**.** We close this lecture with an example of a method that, we might figuratively say, is 'on the brink of stability.' That is, the method is zero-stable, but it stretches that definition to its limit. Consider the method

$$x_{k+2} = x_k + 2hf_{k+1}, \tag{5.7}$$

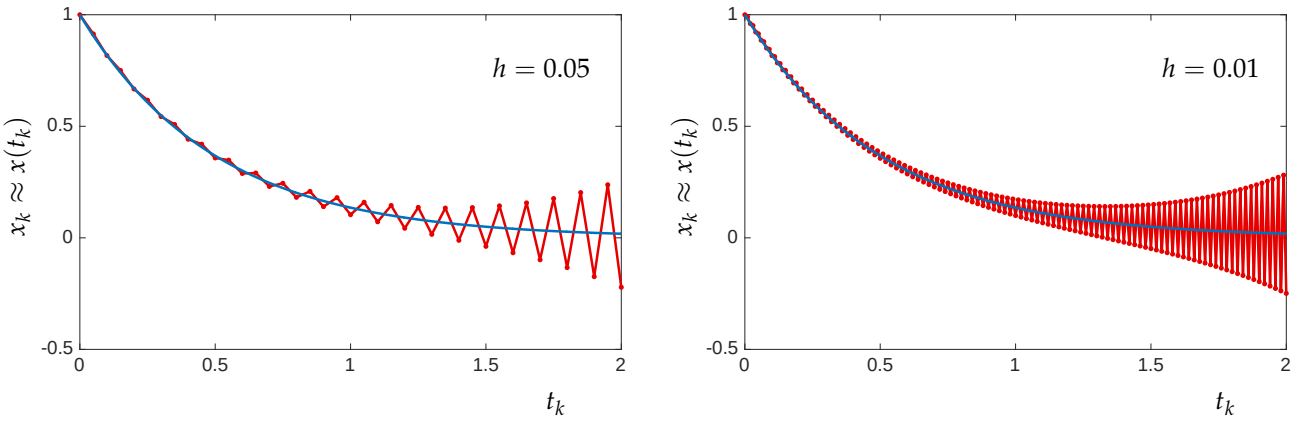


Figure 5.7: The second-order zero-stable method (5.7) applied to $x'(t) = -2x(t)$, started with initial condition $x_0 = 1$ and a 1% error in x_1, i.e., $x_1 = 1.01e^{-2h}$ (shown in red). The method is 'on the brink of stability': the systematic error in x_1 prevents the method from converging, but the deviation from the true solution $x(t) = e^{-2t}$ (in blue) is bounded. This would not be the case if the method violated the root condition.

which has $\mathcal{O}(h^2)$ truncation error. The characteristic polynomial is $z^2 - 1 = (z+1)(z-1)$, which has the two roots $\gamma_1 = -1$ and $\gamma_2 = 1$. These are distinct roots on the unit circle, so the method is zero-stable.

Apply this method to the model problem $x'(t) = \lambda x$ with $x(0) = 1$. Substituting $f(t_k, x_k) = \lambda x_k$ into the method gives

$$x_{k+2} = x_k + 2\lambda h x_{k+1}.$$

For a fixed λ and h, this is just another recurrence relation like we have considered above. It has solutions of the form γ^k, where γ is a root of the polynomial

$$\gamma^2 - 2\lambda h\gamma - 1 = 0.$$

In fact, those roots are simply

$$\gamma = \lambda h \pm \sqrt{\lambda^2 h^2 + 1}.$$

Since $\sqrt{\lambda^2 h^2 + 1} \geq 1$ for any $h > 0$ and $\lambda \neq 0$, at least one of the roots γ will always be greater than one in modulus, thus leading to a solution x_k that grows exponentially with k. Of course, the exact solution to this equation is $x(t) = e^{\lambda t}$, so if $\lambda < 0$, then we have $x(t) \to 0$ as $t \to \infty$. The numerical approximation will generally diverge, giving the qualitatively opposite behavior!

How is this possible for a zero-stable method? The key is that here, unlike our previous zero-unstable method, the exponential growth rate depends upon the time-step h. Zero stability only requires that on a fixed finite time interval $t \in [t_0, t_{\text{final}}]$, the amount by which errors in the initial data are magnified be *bounded*.

Figure 5.7 shosw what this means. Set $\lambda = -2$ and $[t_0, t_{\text{final}}] = [0, 2]$. Start the method with $x_0 = 1$ and $x_1 = 1.01\ e^{-2h}$. That is, the second data point has an initial error of 1%. The plot on the left

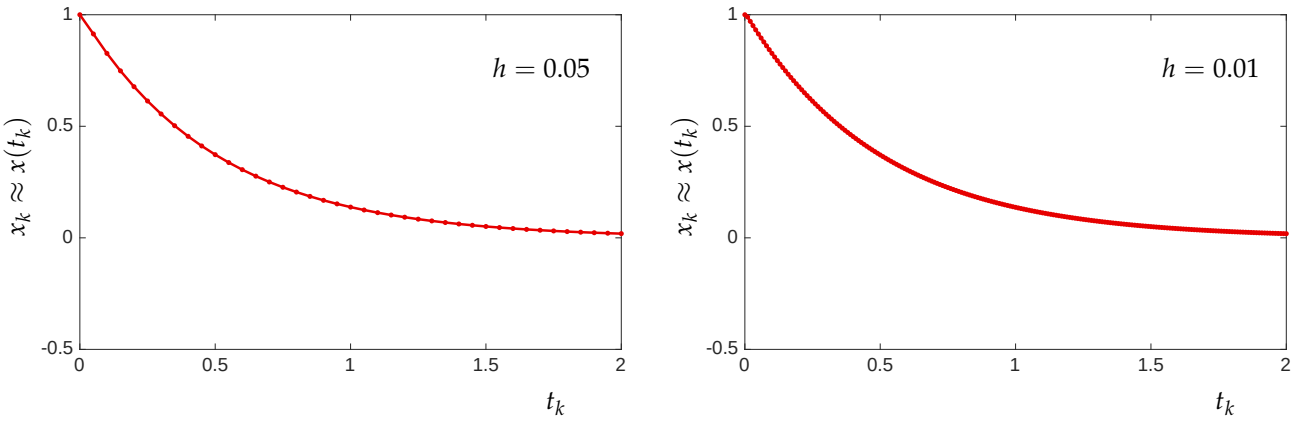


Figure 5.8: Second-order Adams–Bashforth method applied to $x'(t) = -2x(t)$, started with initial condition $x_0 = 1$ and a 10% error in x_1, i.e., $x_1 = 1.01e^{-2h}$. Despite the systematic error, the method still behaves like the true solution, $x(t) = e^{-2t}$.

shows the solution for $h = 0.05$, while the plot on the right uses $h = 0.01$. In both cases, the solution oscillates wildly across the true solution, and the amplitude of these oscillations grows with t. As we reduce the step-size, the solution remains equally bad. (If the method were not zero-stable, we would expect the error to magnify as h shrinks.)

The solution does not blow up, but nor does it converge as $h \to 0$. So does this example contradict the Dahlquist Equivalence Theorem? No! The hypotheses for that theorem require consistent starting values. In this case, that means $x_1 \to x(t_0 + h)$ as $h \to 0$. (We assume that $x_0 = x(t_0)$ is exact.) In the example shown above, we have kept fixed x_1 to have a 1% error as $h \to 0$, so it is not consistent.

Not all linear multistep methods behave as badly as this one in the presence of imprecise starting data. Recall the second-order Adams–Bashforth method from the previous lecture.

$$x_{k+2} - x_{k+1} = \frac{h}{2}(3f_{k+1} - f_k).$$

This method is zero stable, as $\rho(z) = z^2 - z = z(z-1)$. Figure 5.8 repeats the exercise of Figure 5.7, with the same errors in x_1, but with the second-order Adams–Bashforth method. Though the initial value error will throw off the solution slightly, we recover the correct qualitative behavior.

Judging from the different manner in which our two second-order methods handle this simple problem, it appears that there is still more to understand about linear multistep methods. This is the subject of the next lecture.

LECTURE 37: *Linear Multistep Methods: Absolute Stability, Part I*
LECTURE 38: *Linear Multistep Methods: Absolute Stability, Part II*

5.7 Linear multistep methods: absolute stability

At this point, it may well seem that we have a complete theory for linear multistep methods. With an understanding of truncation error and zero stability, the convergence of any method can be easily understood. However, one further wrinkle remains. (Perhaps you expected this: thus far the β_j coefficients have played no role in our stability analysis!) Up to this point, our convergence theory addresses the case where $h \to 0$. Methods differ significantly in how small h must be before one observes this convergent regime. For h too large, exponential errors that resemble those seen for zero-unstable methods can emerge for rather benign-looking problems—and for some ODEs and methods, the restriction imposed on h to avoid such behavior can be severe. To understand this problem, we need to consider how the numerical method behaves on a less trivial canonical model problem.

For an elaboration of many details described here, see Chapter 12 of Süli and Mayers.

Now consider the model problem $x'(t) = \lambda x(t)$, $x(0) = x_0$ for some fixed $\lambda \in \mathbb{C}$, which has the exact solution $x(t) = \mathrm{e}^{t\lambda} x_0$. In those cases where the real part of λ is negative (i.e., λ is in the open left half of the complex plane), we have $|x(t)| \to 0$ as $t \to \infty$. For a fixed step size $h > 0$, will a linear multistep method mimic this behavior? The explicit Euler method applied to this equation takes the form

$$\begin{aligned} x_{k+1} &= x_k + h f_k \\ &= x_k + h\lambda x_k \\ &= (1 + h\lambda) x_k. \end{aligned}$$

Hence, this recursion has the general solution

$$x_k = (1 + h\lambda)^k x_0.$$

Under what conditions will $x_k \to 0$? Clearly we need $|1 + h\lambda| < 1$; this condition is more easily interpreted by writing $|1 + h\lambda| = |-1 - h\lambda|$, where that latter expression is simply the distance of $h\lambda$ from -1 in the complex plane. Hence $|1 + h\lambda| < 1$ provided $h\lambda$ is located strictly in the interior of the disk of radius 1 in the complex plane, centered at -1. This is the *stability region* for the explicit Euler method, shown in the plot on the next page.

Now consider the backward (implicit) Euler method for this same model problem:

$$\begin{aligned} x_{k+1} &= x_k + h f_{k+1} \\ &= x_k + h\lambda x_{k+1}. \end{aligned}$$

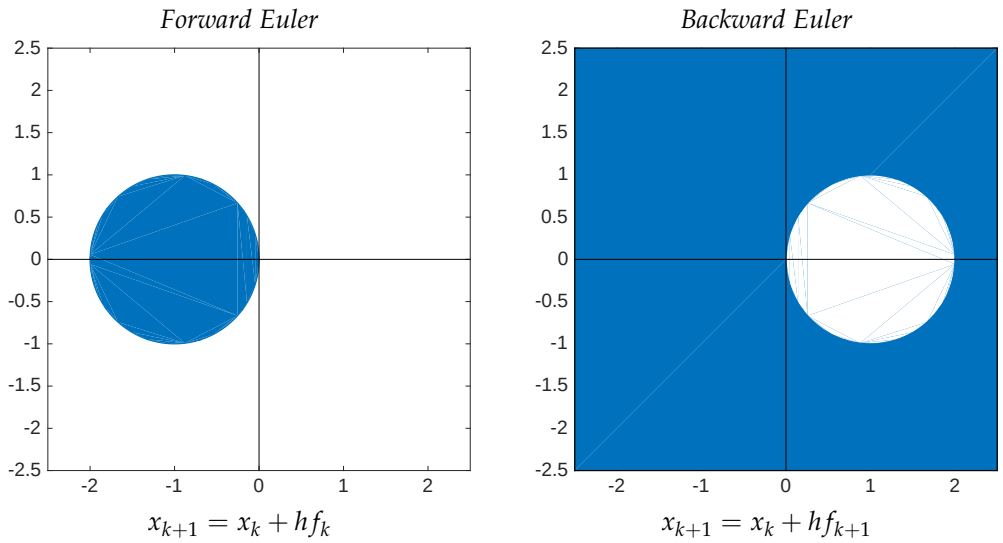


Figure 5.9: Stability regions for the forward and backward Euler method. If $h\lambda$ is contained within the blue region, then the approximate solution $\{x_k\}$ to $x'(t) = \lambda x(t)$ will converge, $|x_k| \to 0$, as $k \to \infty$.

Solve this equation for x_{k+1} to obtain

$$x_{k+1} = \frac{1}{1 - h\lambda} x_k,$$

from which it follows that

$$x_k = (1 - h\lambda)^{-k} x_0.$$

Thus $x_k \to 0$ provided $|1 - h\lambda| > 1$, i.e., $h\lambda$ must be *more than* a distance of 1 away from 1 in the complex plane. As illustrated in the plot on the next page, the backward Euler method has a much larger stability region than the explicit Euler method. In fact, the entire left half of the complex plane is contained in the stability region for the implicit method. Since $h > 0$, for any value of λ with negative real part, the backward Euler method will produce decaying solutions that qualitatively mimic the exact solution.

If $h\lambda$ falls within the stability region for a method, we say that the method is *absolutely stable* for that value of $h\lambda$. Figure 5.9 shows the stability regions for the forward and backward Euler methods. The blue region shows values of λh in the complex plane for which the method is absolutely stable. (For the backward Euler method, this regions extends throughout the complex plane, beyond the range of the plot.)

A general linear multistep method

$$\sum_{j=0}^{m} \alpha_j x_{k+j} = h \sum_{j=0}^{m} \beta_j f_{k+j}$$

applied to $x'(t) = \lambda x$, $x(0) = x_0$ reduces to

$$\sum_{j=0}^{m} \alpha_j x_{k+j} = h\lambda \sum_{j=0}^{m} \beta_j x_{k+j},$$

which can be rearranged as

$$\sum_{j=0}^{m} (\alpha_j - h\lambda\beta_j) x_{k+j}.$$

This expression closely resembles the formula we analyzed when assessing the zero stability of linear multistep methods, except that now we have the $h\lambda\beta_j$ terms. The new equation is also a linear constant-coefficient recurrence relation, so just as before we can assume that it has solutions of the form $x_k = \gamma^k$ for constant γ. The values of $\gamma \in \mathbb{C}$ for which such x_k will be solutions to the recurrence are the roots of the *stability polynomial*

$$\sum_{j=0}^{m} (\alpha_j - h\lambda\beta_j) z^j,$$

which can be written as

$$\rho(z) - h\lambda\sigma(z) = 0,$$

where ρ is the characteristic polynomial,

$$\rho(z) = \sum_{j=0}^{m} \alpha_j z^j$$

and

$$\sigma(z) = \sum_{j=0}^{m} \beta_j z^j.$$

Thus for a fixed $h\lambda$, there will be m solutions of the form γ_j^k for the m roots $\gamma_1, \ldots, \gamma_m$ of the stability polynomial. If these roots are all distinct, then for any initial data $x_0, \ldots, x_{m-1}$ we can find constants $c_1, \ldots, c_m$ such that

$$x_k = \sum_{j=1}^{m} c_j \gamma_j^k.$$

For a given value $h\lambda$, we have $x_k \to 0$ provided that $|\gamma_j| < 1$ for all $j = 1, \ldots, m$. If that condition is met, we say that the linear multistep method is *absolutely stable* for that value of $h\lambda$.

In the next section, we will describe how linear systems of differential equations, $\mathbf{x}'(t) = \mathbf{A}\mathbf{x}(t)$, can give rise, through an eigenvalue decomposition of $\mathbf{A}$, to the scalar problem $x'(t) = \lambda x(t)$ with complex values of the eigenvalue λ (even if $\mathbf{A}$ is real). This explains our interest in values of $h\lambda \in \mathbb{C}$.

We can now add a condition to our growing list of requirements to look for when assessing the quality of a linear multistep method. We seek linear multistep methods with the following properties:

- high order truncation error;
- zero stability;
- absolute stability region that contains as much of the left half of the complex plane as possible.

Those methods for which the stability region contains the entire left half plane are distinguished, as they will produce, *for any value of* h, exponentially decaying numerical solutions to linear problems that have exponentially decaying true solutions, i.e., when $\operatorname{Re}\lambda < 0$.

Definition 5.8. A linear multistep method is *A-stable* provided that its stability region contains the entire left half of the complex plane.

Figure 5.10 shows the stability regions for two Adams–Bashforth and Adams–Moulton methods. Notice two trends in these plots: (1) the implicit Adams–Moulton methods have a larger stability region than the explicit Adams–Bashforth methods of the same order; (2) as the order of the method increases, the stability region gets smaller.

Figure 5.11 shows the stability regions for a class of implicit integrators called *backward difference methods*. The 1-step backward difference method is simply the trapezoid method described earlier. All four of these methods have contain the entire negative axis within their stability region, which will make these methods very effective for important systems of differential equations we will discuss in the next lecture.

How does one draw plots of the sort shown here? We take the second order Adams–Bashforth method

$$x_{k+2} - x_{k+1} = h(\tfrac{3}{2}f_{k+1} - \tfrac{1}{2}f_k)$$

as an example. Apply this rule to $x'(t) = f(t, x(t)) = \lambda x(t)$ to obtain

$$x_{k+2} - x_{k+1} = \lambda h(\tfrac{3}{2}x_{k+1} - \tfrac{1}{2}x_k),$$

with which we associate the stability polynomial

$$z^2 - (1 + \tfrac{3}{2}\lambda h)z + \tfrac{1}{2}\lambda h = 0.$$

Any point $\lambda h \in \mathbb{C}$ on the boundary of the stability region must be one for which the stability polynomial has a root z with $|z| = 1$. We can rearrange the stability polynomial to give

$$\lambda h = \frac{z^2 - z}{\frac{3}{2}z - 1}.$$

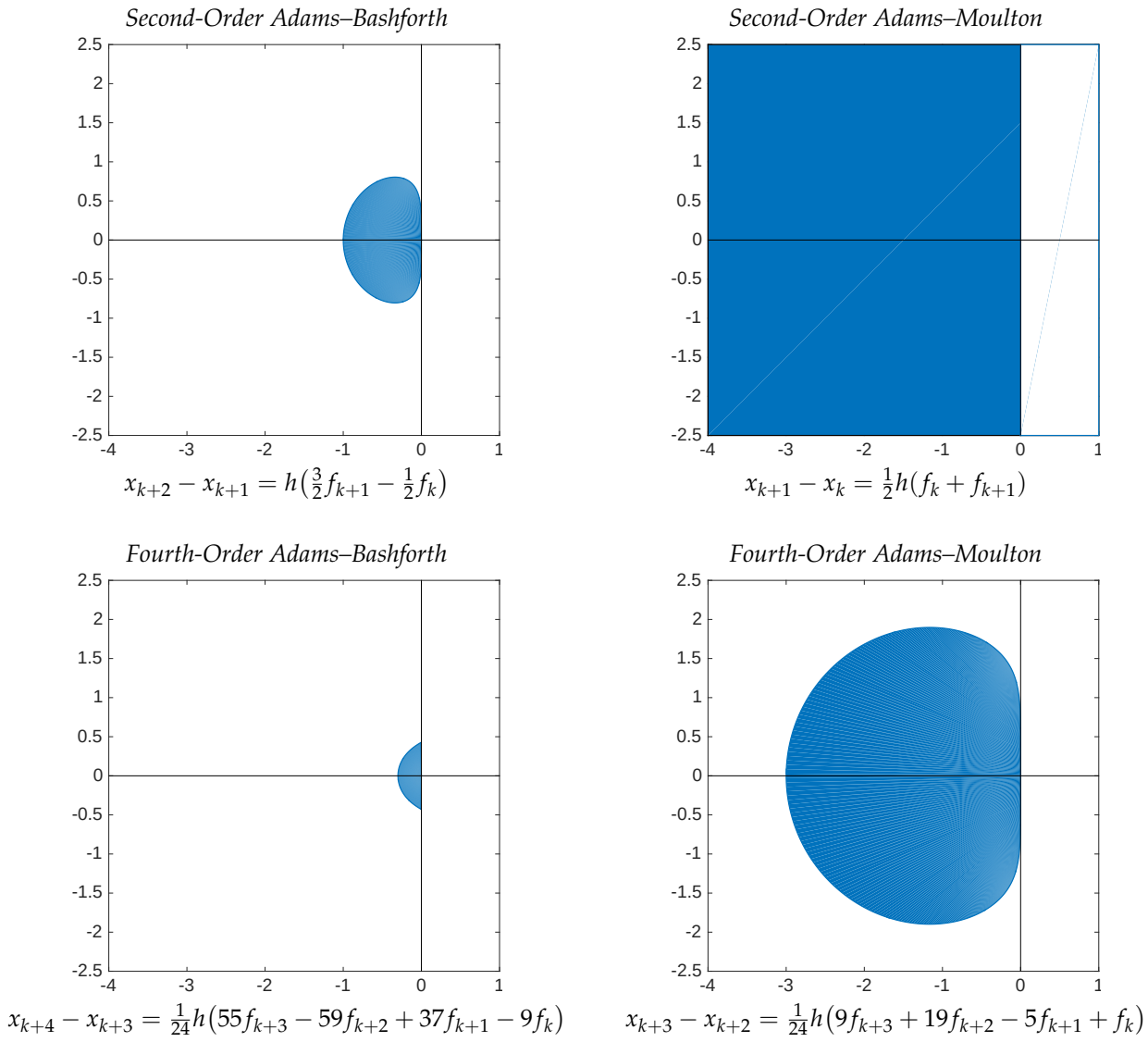


Figure 5.10: Stability regions for the second-order and fourth-order Adams–Bashforth (explicit) and Adams–Moulton (implicit) methods. If $h\lambda$ is contained within the blue region, then the approximate solution $\{x_k\}$ to $x'(t) = \lambda x(t)$ will converge, $|x_k| \to 0$, as $k \to \infty$.

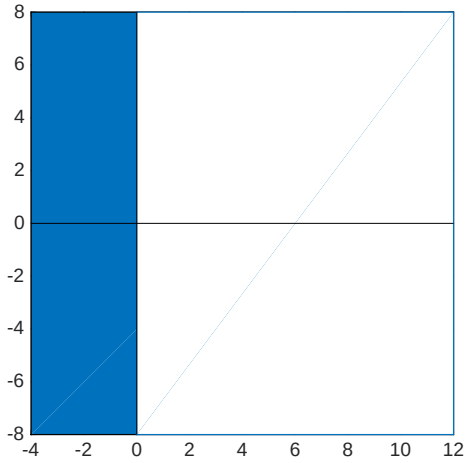
For general methods, this expression takes the form

$$\lambda h = \frac{\sum_{j=0}^{m} \alpha_j z^j}{\sum_{j=0}^{m} \beta_j z^j}, \tag{5.8}$$

To determine the boundary of the stability region, we sample this formula for all $z \in \mathbb{C}$ with $|z| = 1$, i.e., we trace out the image for $z = e^{i\theta}$, $\theta \in [0, 2\pi)$. This curve will divide the complex plane into stable and unstable regions, which can be distinguished by testing the roots of the stability polynomial for λh within each of those regions.

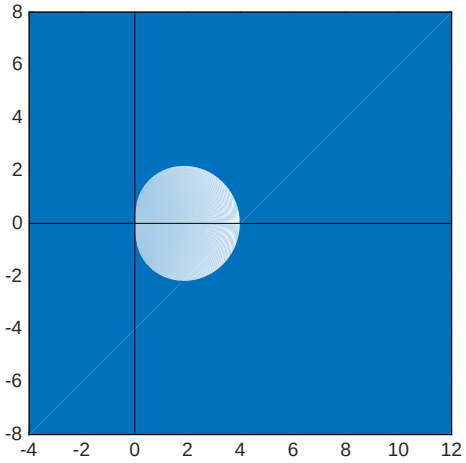
We illustrate this process for the fourth order Adams–Bashforth scheme. The curve described in the last paragraph is shown in Figure 5.12; it divides the complex plane into regions where the stability

1-Step Backward Difference Method (Trapezoid)



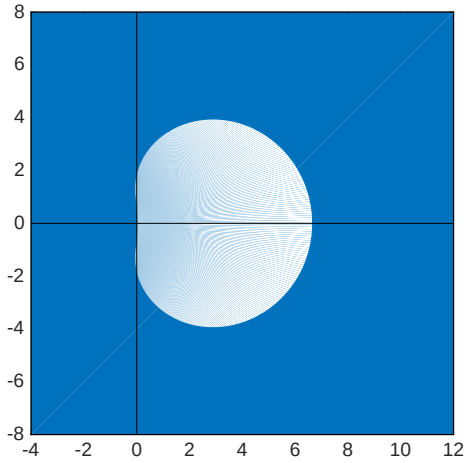
$$x_{k+1} - x_k = \tfrac{1}{2}h(f_k + f_{k+1})$$

2-Step Backward Difference Method



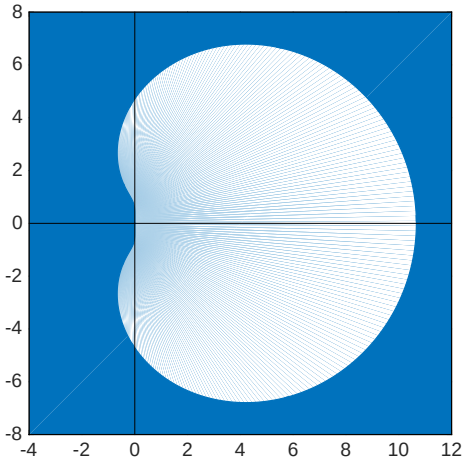
$$3x_{k+2} - 4x_{k+1} + x_k = 2hf_{k+2}$$

3-Step Backward Difference Method



$$11x_{k+3} - 18x_{k+2} + 9x_{k+1} - 2x_k = 6hf_{k+3}$$

4-Step Backward Difference Method



$$25x_{k+4} - 48x_{k+3} + 36x_{k+2} - 16x_{k+1} + 3x_k = 12hf_{k+4}$$

Figure 5.11: Stability regions for four (implicit) backward difference methods. If $h\lambda$ is contained within the blue region, then the approximate solution $\{x_k\}$ to $x'(t) = \lambda x(t)$ will converge, $|x_k| \to 0$, as $k \to \infty$.

polynomial has an equal numbers of roots larger than 1 in magnitude. As denoted by the numbers on the plot: outside the curve there is one root larger than one; within the rightmost lobes of this curve, two roots are larger than one; within the leftmost region, no roots are larger than one in magnitude. The latter is the stable region, which is colored blue in the bottom-left plot in Figure 5.10.

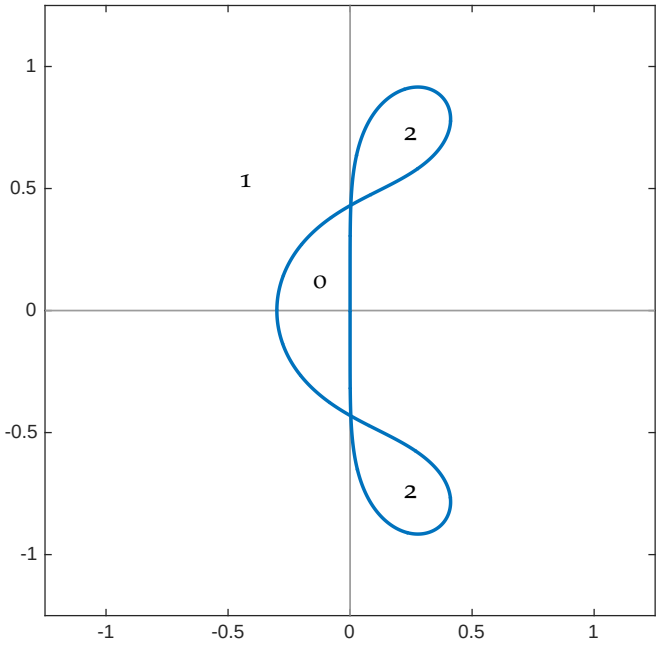


Figure 5.12: The curve traced out by $\sum_{j=0}^{m} a_j e^{ji\theta} / \sum_{j=0}^{m} b_j e^{ji\theta}$ for $\theta \in [0, 2\pi)$. The numbers reveal the number of roots of the stability polynomial that have magnitude larger than one. The stability region is the region bounded by this curve for which all the roots of the stability polynomial are less than one in magnitude.

LECTURE 39: *Systems of ODEs, Stiff differential equations*

5.8 Systems of linear differential equations

Thus far we have mainly considered scalar ODEs. Both one-step and linear multistep methods readily generalize to *systems* of ODEs, where the scalar $x(t) \in \mathbb{R}$ is replaced by a vector $\mathbf{x}(t) \in \mathbb{R}^n$. In these notes, we shall focus upon *linear systems* of ODEs.

Of course, many applications give rise to *nonlinear* ODEs; understanding the linear case is essential to understanding the behavior of nonlinear systems in the vicinity of a steady-state.

Consider the linear system of differential equations

$$\mathbf{x}'(t) = \mathbf{A}\mathbf{x}(t), \quad \mathbf{x}(0) = \mathbf{x}_0,$$

for $\mathbf{A} \in \mathbb{C}^{n\times n}$ and $\mathbf{x}(t) \in \mathbb{C}^n$. We wish to see how the scalar linear stability theory discussed in the last lecture applies to such systems. Suppose that the matrix $\mathbf{A}$ is diagonalizable, so that it can be written $\mathbf{A} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^{-1}$ for the diagonal matrix $\mathbf{\Lambda} = \operatorname{diag}(\lambda_1, \ldots, \lambda_n)$. Premultiplying the differential equation by $\mathbf{V}^{-1}$ yields

$$\mathbf{V}^{-1}\mathbf{x}'(t) = \mathbf{\Lambda}\mathbf{V}^{-1}\mathbf{x}(t), \quad \mathbf{V}^{-1}\mathbf{x}(0) = \mathbf{V}^{-1}\mathbf{x}_0. \tag{5.9}$$

Now let $\mathbf{y}(t) = \mathbf{V}^{-1}\mathbf{x}(t)$, which represents $\mathbf{x}(t)$ in a transformed coordinate system in which the eigenvectors form the new coordinate axes. In these new coordinates, the matrix equation decouples into a system of n independent scalar linear equations: the system (5.9) can be written as

$$\mathbf{y}'(t) = \mathbf{\Lambda}\mathbf{y}(t), \quad \mathbf{y}(0) = \mathbf{V}^{-1}\mathbf{x}_0.$$

This system is equivalent to the n scalar problems

$$\begin{aligned} y_1'(t) &= \lambda_1 y_1(t), \quad y_1(0) = [\mathbf{V}^{-1}\mathbf{x}_0]_1, \\ &\vdots \qquad\qquad\qquad \vdots \\ y_n'(t) &= \lambda_n y_n(t), \quad y_n(0) = [\mathbf{V}^{-1}\mathbf{x}_0]_n, \end{aligned}$$

each of which has a simple solution of the form

$$y_j(t) = \mathrm{e}^{\lambda_j t} y_j(0).$$

Now use the relationship $\mathbf{x}(t) = \mathbf{V}\mathbf{y}(t)$ to transform back to the original coordinates. Collect the exponentials $\mathrm{e}^{\lambda_j t}$ into a diagonal matrix,

$$\mathrm{e}^{\mathbf{\Lambda} t} := \begin{bmatrix} \mathrm{e}^{t\lambda_1} & & \\ & \ddots & \\ & & \mathrm{e}^{t\lambda_n} \end{bmatrix}.$$

Then write

$$\mathbf{x}(t) \;=\; \mathbf{V}\mathbf{y}(t) \;=\; \mathbf{V}\mathrm{e}^{\mathbf{\Lambda} t}\mathbf{y}(0) \;=\; \mathbf{V}\mathrm{e}^{\mathbf{\Lambda} t}\mathbf{V}^{-1}\mathbf{x}_0, \tag{5.10}$$

which motivates the definition of the *matrix exponential*,

$$\mathrm{e}^{\mathbf{A}t} := \mathbf{V}\mathrm{e}^{\mathbf{\Lambda} t}\mathbf{V}^{-1},$$

giving the solution $\mathbf{x}(t)$ has the convenient form

$$\mathbf{x}(t) = \mathrm{e}^{\mathbf{A}t}\mathbf{x}_0.$$

For scalar equations, we considered when $|x(t)| \to 0$. The analogy for $\mathbf{x}(t) \in \mathbb{R}^n$ is $\|\mathbf{x}(t)\|_2 \to 0$. What properties of $\mathbf{A}$ ensure this convergence?

We can bound the solution using norm inequalities,

$$\|\mathbf{x}(t)\|_2 \;\le\; \|\mathbf{V}\|_2 \, \|\mathrm{e}^{\mathbf{\Lambda} t}\|_2 \, \|\mathbf{V}^{-1}\|_2 \, \|\mathbf{x}_0\|_2.$$

Since $\mathrm{e}^{\mathbf{\Lambda} t}$ is a diagonal matrix, its 2-norm is the largest magnitude of its entries:

$$\|\mathrm{e}^{\mathbf{\Lambda} t}\|_2 = \max_{1 \le j \le n} |\mathrm{e}^{t\lambda_j}|,$$

and hence

$$\frac{\|\mathbf{x}(t)\|_2}{\|\mathbf{x}_0\|_2} \;\le\; \|\mathbf{V}\|_2 \, \|\mathbf{V}^{-1}\|_2 \, \max_{1 \le j \le n} |\mathrm{e}^{t\lambda_j}|. \tag{5.11}$$

Thus the asymptotic decay rate of $\|\mathbf{x}(t)\|_2$ is controlled by the right-most eigenvalue of $\mathbf{A}$ in the complex plane. If *all* eigenvalues of $\mathbf{A}$ have negative real part, then $\|\mathbf{x}(t)\|_2 \to 0$ as $t \to \infty$ for all initial conditions $\mathbf{x}(0)$. Such systems are called *stable*.

When $\|\mathbf{V}\|_2 \|\mathbf{V}^{-1}\|_2 > 1$, it is possible that $\|\mathbf{x}(t)\|_2 / \|\mathbf{x}_0\|_2 > 1$ for small $t > 0$, even if this ratio must eventually decay to zero as $t \to 0$.The possibility of this *transient growth* complicates the analysis of dynamical systems with non-symmetric coefficient matrices, and turns out to be closely related the sensitivity of the eigenvalues of $\mathbf{A}$ to perturbations. This behavior is both fascinating and physically important, but regrettably beyond the scope of these lectures.

Note that the definition $\mathrm{e}^{t\mathbf{A}} = \mathbf{V}\mathrm{e}^{t\mathbf{\Lambda}}\mathbf{V}^{-1}$ is consistent with the more general definition obtained by substituting $t\mathbf{A}$ into the same Taylor series that defines the scalar exponential:

$$\mathrm{e}^{t\mathbf{A}} = \mathbf{I} + t\mathbf{A} + \frac{1}{2!}t^2\mathbf{A}^2 + \frac{1}{3!}t^3\mathbf{A}^3 + \frac{1}{4!}t^4\mathbf{A}^4 + \cdots.$$

Setting $\mathbf{x}(t) = \mathrm{e}^{t\mathbf{A}}\mathbf{x}_0$ with this series definition of $\mathrm{e}^{t\mathbf{A}}$,

$$\begin{aligned}
\mathbf{x}'(t) &= \frac{\mathrm{d}}{\mathrm{d}t}\Big(\mathrm{e}^{t\mathbf{A}}\mathbf{x}_0\Big) \\
&= \frac{\mathrm{d}}{\mathrm{d}t}\Big(\mathbf{I} + t\mathbf{A} + \frac{t^2}{2!}\mathbf{A}^2 + \frac{t^3}{3!}\mathbf{A}^3 + \cdots\Big)\mathbf{x}_0 \\
&= \Big(\mathbf{0} + \mathbf{A} + t\mathbf{A}^2 + \frac{t^2}{2!}\mathbf{A}^3 + \frac{t^3}{3!}\mathbf{A}^4 + \cdots\Big)\mathbf{x}_0 \\
&= \mathbf{A}\Big(\mathbf{I} + t\mathbf{A} + \frac{t^2}{2!}\mathbf{A}^2 + \frac{t^3}{3!}\mathbf{A}^3 + \cdots\Big)\mathbf{x}_0 \\
&= \mathbf{A}\mathrm{e}^{t\mathbf{A}}\mathbf{x}_0 \\
&= \mathbf{A}\mathbf{x}(t).
\end{aligned}$$

Hence $\mathbf{x}(t) = e^{t\mathbf{A}}\mathbf{x}_0$ solves the equation $\mathbf{x}'(t) = \mathbf{A}\mathbf{x}(t)$, and satisfies the initial condition $\mathbf{x}(0) = \mathbf{x}_0$.

5.8.1 *Linear multistep methods for linear systems*

What can be said of the behavior of a linear multistep method applied to $\mathbf{x}'(t) = \mathbf{A}\mathbf{x}(t)$? Euler's method, for example, takes the form

$$\begin{aligned}\mathbf{x}_{k+1} &= \mathbf{x}_k + h\mathbf{A}\mathbf{x}_k \\ &= (\mathbf{I} + h\mathbf{A})\mathbf{x}_k.\end{aligned}$$

Iterating from the initial condition,

$$\begin{aligned}\mathbf{x}_1 &= (\mathbf{I} + h\mathbf{A})\mathbf{x}_0 \\ \mathbf{x}_2 &= (\mathbf{I} + h\mathbf{A})\mathbf{x}_1 = (\mathbf{I} + h\mathbf{A})^2\mathbf{x}_0 \\ \mathbf{x}_3 &= (\mathbf{I} + h\mathbf{A})\mathbf{x}_2 = (\mathbf{I} + h\mathbf{A})^3\mathbf{x}_0 \\ &\vdots\end{aligned}$$

and, in general,

$$\mathbf{x}_k = (\mathbf{I} + h\mathbf{A})^k\mathbf{x}_0.$$

We can understand the *asymptotic* behavior of $(\mathbf{I} + h\mathbf{A})^k$ by examining the eigenvalues of $(\mathbf{I} + h\mathbf{A})^k$: the quantity $(\mathbf{I} + h\mathbf{A})^k \to \mathbf{0}$ if and only if all the eigenvalues of $\mathbf{I} + h\mathbf{A}$ are less than one in magnitude. The *spectral mapping theorem* ensures that if $(\lambda_j, \mathbf{v}_j)$ is an eigenvalue-eigenvector pair for $\mathbf{A}$, then $(1 + h\lambda_j, \mathbf{v}_j)$ is an eigenpair of $\mathbf{I} + h\mathbf{A}$. This is easy to verify by a direct computation: If $\mathbf{A}\mathbf{v}_j = \lambda_j\mathbf{v}_j$, then $(\mathbf{I} + h\mathbf{A})\mathbf{v}_j = \mathbf{v}_j + h\mathbf{A}\mathbf{v}_j = (1 + h\lambda_j)\mathbf{v}_j$.

Recall that $\mathbf{M}^k \to \mathbf{0}$ as $k \to \infty$ if and only if all eigenvalues of $\mathbf{M}$ have magnitude strictly less than one.

It follows that the numerical solution $\mathbf{x}_k$ computed by Euler's method will decay to zero if $|1 + h\lambda_j| < 1$ for *all* eigenvalues λ_j of $\mathbf{A}$. In the language of the last lecture, this requires that $h\lambda_j$ falls in the absolute stability region for the forward Euler method for *all* eigenvalues λ_j of $\mathbf{A}$.

For a general linear multistep method, this criterion generalizes to the requirement that $h\lambda_j$ be located in the method's absolute stability region for all eigenvalues λ_j of $\mathbf{A}$. This phenomenon is illustrated in Figures 5.13 and 5.14. Here $\mathbf{A}$ is a 16×16 matrix with all its eigenvalues in the left half of the complex plane. We wish to solve $\mathbf{x}'(t) = \mathbf{A}\mathbf{x}(t)$ using the second-order Adams–Bashforth method, whose stability region was plotted in the last lecture. The plots in Figure 5.13 show $h\lambda_j$ as crosses for the eigenvalues $\lambda_1, \ldots, \lambda_{16}$ of $\mathbf{A}$. If *any* value of $h\lambda_j$ is outside the stability region (shown in blue), then the iteration will *grow exponentially*. If h is sufficiently small that $h\lambda_j$

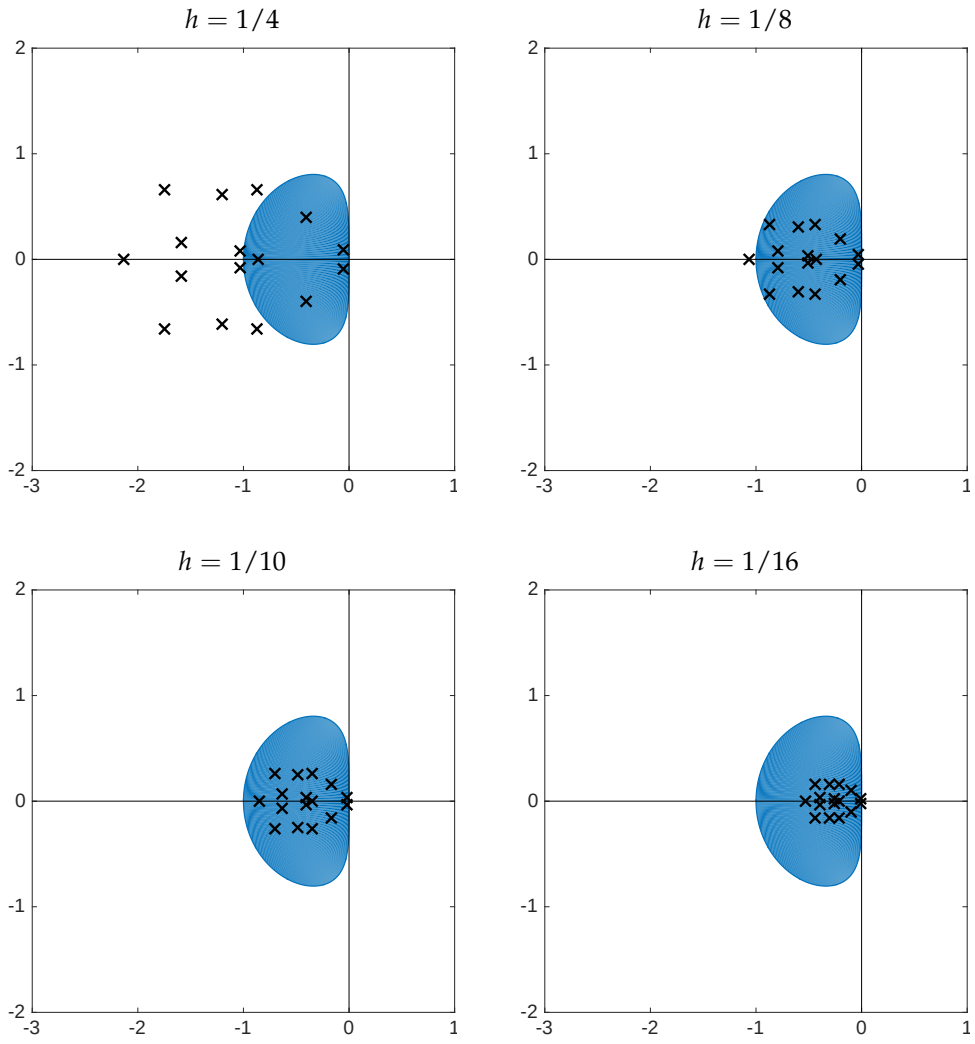


Figure 5.13: Values of $h\lambda_j$ for all eigenvalues of a 16×16 matrix $\mathbf{A}$. For $h = 1/4$ many values of $h\lambda_j$ fall outside the stability region of the second-order Adams–Bashforth method. For $h = 1/8$, only one $h\lambda_j$ is outside the stability region, but that is enough to mean that $\|\mathbf{x}_k\| \to \infty$ as $k \to \infty$. For $h = 1/10$ and $h = 1/16$, *all* values of $h\lambda_j$ are in the stability region, so $\|\mathbf{x}_k\| \to 0$ as $k \to \infty$.

is in the stability region for *all* eigenvalues λ_j, then $\mathbf{x}_k \to \mathbf{0}$ as $k \to \infty$, consistent with the behavior of the exact solution, $\mathbf{x}(t) \to \mathbf{0}$ as $t \to \infty$. Figure 5.13 shows $\|\mathbf{x}_k\|_2$ as a function of t_k for three values of h (two unstable and one stable).

This example deserves closer scrutiny. Suppose $\mathbf{A}$ is diagonalizable, so we can write $\mathbf{A} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^{-1}$. Thus,

$$\begin{aligned}
\mathbf{x}_k &= (\mathbf{I} + h\mathbf{A})^k \mathbf{x}_0 \\
&= (\mathbf{I} + h\mathbf{V}\mathbf{\Lambda}\mathbf{V}^{-1})^k \mathbf{x}_0 \\
&= (\mathbf{V}\mathbf{V}^{-1} + h\mathbf{V}\mathbf{\Lambda}\mathbf{V}^{-1})^k \mathbf{x}_0 \\
&= \mathbf{V}(\mathbf{I} + h\mathbf{\Lambda})^k \mathbf{V}^{-1} \mathbf{x}_0.
\end{aligned}$$

Compare this last expression to the formula (5.10) for the true solu-

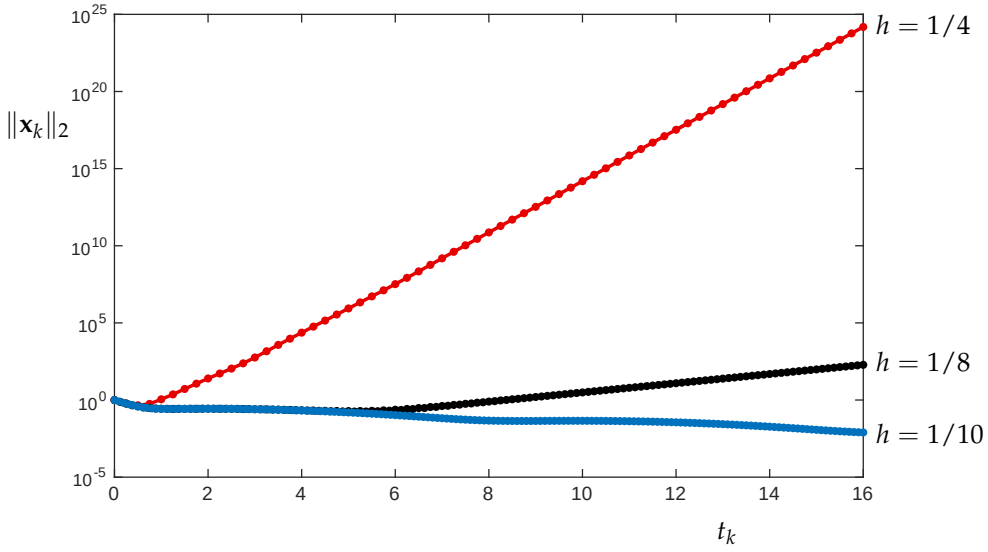


Figure 5.14: The second-order Adams–Bashforth method applied to $\mathbf{x}'(t) = \mathbf{A}\mathbf{x}(t)$ for the same matrix $\mathbf{A}$ used for Figure 5.13. As seen in that figure, for step-sizes $h = 1/4$ and $h = 1/8$ the method is unstable, and $\|\mathbf{x}_k\|_2 \to \infty$ as $k \to \infty$. When $h = 1/10$, $h\lambda_j$ is in the stability region for all eigenvalues λ_j of $\mathbf{A}$, and hence $\|\mathbf{x}_k\|_2 \to 0$ as $k \to \infty$.

tion $\mathbf{x}(t)$ in terms of the matrix exponential. As we did in that case, we can bound $\|\mathbf{x}_k\|_2$:

$$\begin{aligned}\|\mathbf{x}_k\|_2 &= \|\mathbf{V}(\mathbf{I} + h\mathbf{\Lambda})^k\mathbf{V}^{-1}\mathbf{x}_0\|_2 \\ &= \|\mathbf{V}(\mathbf{I} + h\mathbf{\Lambda})^k\mathbf{V}^{-1}\|_2\|\mathbf{x}_0\|_2 \\ &= \|\mathbf{V}\|_2\|\mathbf{V}^{-1}\|_2\|(\mathbf{I} + h\mathbf{\Lambda})^k\|_2\|\mathbf{x}_0\|_2.\end{aligned}$$

Since $\mathbf{I} + h\mathbf{\Lambda}$ is a diagonal matrix,

$$(\mathbf{I} + h\mathbf{\Lambda})^k = \begin{bmatrix} (1+h\lambda_1)^k & & & \\ & (1+h\lambda_2)^k & & \\ & & \ddots & \\ & & & (1+h\lambda_n)^k \end{bmatrix},$$

giving

$$\|(\mathbf{I} + h\mathbf{\Lambda})^k\|_2 = \max_{1\le j\le n} |1 + h\lambda_j|^k.$$

Thus, we arrive at the bound

$$\frac{\|\mathbf{x}_k\|_2}{\|\mathbf{x}_0\|_2} \le \|\mathbf{V}\|_2\|\mathbf{V}^{-1}\|_2 \max_{1\le j\le n} |1 + h\lambda_j|^k,$$

which is analogous to the bound (5.11) for the exact solution.

We can glean just a bit more from our analysis of $\mathbf{x}_k$. Since $\mathbf{A}$ is diagonalizable, its eigenvectors $\mathbf{v}_1, \ldots, \mathbf{v}_n$ for a basis for $\mathbb{C}^n$. Expand the initial condition $\mathbf{x}_0$ in this basis:

$$\mathbf{x}_0 = \sum_{j=1}^{n} c_j\mathbf{v}_j = \mathbf{V}\mathbf{c}.$$

Now, our earlier expression for $\mathbf{x}_k$ gives

$$\mathbf{x}_k = \mathbf{V}(\mathbf{I} + h\mathbf{\Lambda})^k \mathbf{V}^{-1}\mathbf{x}_0 = \mathbf{V}(\mathbf{I} + h\mathbf{\Lambda})^k \mathbf{V}^{-1}\mathbf{V}\mathbf{c} = \mathbf{V}(\mathbf{I} + h\mathbf{\Lambda})^k \mathbf{c}.$$

Since

$$\begin{bmatrix} (1+h\lambda_1)^k & & & \\ & (1+h\lambda_2)^k & & \\ & & \ddots & \\ & & & (1+h\lambda_n)^k \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} (1+h\lambda_1)^k c_1 \\ (1+h\lambda_2)^k c_2 \\ \vdots \\ (1+h\lambda_n)^k c_n \end{bmatrix},$$

we have

(5.12)

$$\mathbf{x}_k = \begin{bmatrix} \mathbf{v}_1 & \mathbf{v}_2 & \cdots & \mathbf{v}_n \end{bmatrix} \begin{bmatrix} (1+h\lambda_1)^k c_1 \\ (1+h\lambda_2)^k c_2 \\ \vdots \\ (1+h\lambda_n)^k c_n \end{bmatrix} = \sum_{j=1}^{n} c_j (1+h\lambda_j)^k \mathbf{v}_j.$$

Thus as $k \to \infty$, the approximate solution $\mathbf{x}_k$ will start to look more and more like (a scaled version of) the vector $\mathbf{v}_\ell$, where ℓ is the index that maximizes $|1 + h\lambda_j|$:

$$|1 + h\lambda_\ell| = \max_{1 \le j \le n} |1 + h\lambda_j|.$$

5.8.2 Stiff differential equations

In the example in Figures 5.13 and 5.14 the step-size did not need to be very small for all $h\lambda_j$ to be contained within the stability region. However, most practical examples in science and engineering yield matrices $\mathbf{A}$ whose eigenvalues span multiple orders of magnitude – and in this case, the stability requirement is far more difficult to satisfy. First consider the following simple example. Let

$$\mathbf{A} = \begin{bmatrix} -199 & -198 \\ 99 & 98 \end{bmatrix}$$

which has the diagonalization

$$\mathbf{A} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^{-1} = \begin{bmatrix} -1 & 2 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} -1 & 0 \\ 0 & -100 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 1 & 1 \end{bmatrix}.$$

The eigenvalues are $\lambda_1 = -1$ and $\lambda_2 = -100$, and the exact solution takes the form

$$\mathbf{x}(t) = \mathrm{e}^{t\mathbf{A}}\mathbf{x}_0 = \mathbf{V} \begin{bmatrix} \mathrm{e}^{-t} & 0 \\ 0 & \mathrm{e}^{-100t} \end{bmatrix} \mathbf{V}^{-1}\mathbf{x}_0.$$

Writing the initial condition as

$$\mathbf{x}_0 = \mathbf{V}\begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = c_1\begin{bmatrix} -1 \\ 1 \end{bmatrix} + c_2\begin{bmatrix} 2 \\ -1 \end{bmatrix},$$

Note that $\mathbf{c} = \mathbf{V}^{-1}\mathbf{x}_0$.

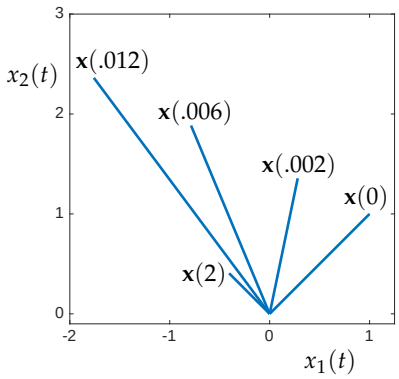
the solution is

$$\begin{aligned}\mathbf{x}(t) &= \mathbf{V}\begin{bmatrix} \mathrm{e}^{-t} & 0 \\ 0 & \mathrm{e}^{-100t} \end{bmatrix}\mathbf{V}^{-1}\mathbf{x}_0 \\ &= \mathbf{V}\begin{bmatrix} \mathrm{e}^{-t} & 0 \\ 0 & \mathrm{e}^{-100t} \end{bmatrix}\begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = c_1\mathrm{e}^{-t}\begin{bmatrix} -1 \\ 1 \end{bmatrix} + c_2\mathrm{e}^{-100t}\begin{bmatrix} 2 \\ -1 \end{bmatrix},\end{aligned}$$

and so $\mathbf{x}(t) \to \mathbf{0}$ as $t \to \infty$. The eigenvalue $\lambda_2 = -100$ corresponds to a *fast transient*, a component of the solution that decays very rapidly; the eigenvalue $\lambda_1 = -1$ corresponds to a *slow transient*, a component of the solution that decays much more slowly. Using this insight we can describe the behavior of the system as $t \to 0$ more precisely than merely saying $\mathbf{x}(t) \to \mathbf{0}$. Since e^{-100t} decays much more quickly than e^{-t}, the solution will be dominated by the λ_1 term:

$$\mathbf{x}(t) \sim c_1\mathrm{e}^{-t}\begin{bmatrix} -1 \\ 1 \end{bmatrix}, \quad t \to \infty,$$

provided $c_1 \neq 0$. This means that the solution vector $\mathbf{x}(t)$ will quickly align in the $\mathbf{v}_1$ direction as it converges toward zero.



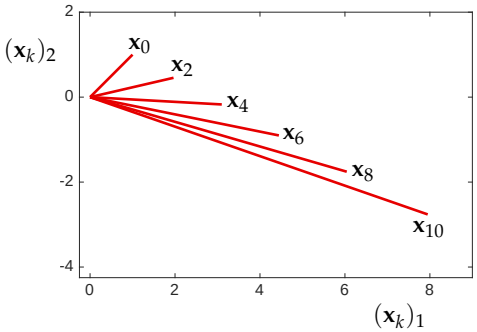
Some snapshots of the exact solution $\mathbf{x}(t)$ for the example with $\lambda_1 = -1$ and $\lambda_2 = -100$, using initial condition $\mathbf{x}(0) = [1,1]^T$. The solution decays, $\|\mathbf{x}(t)\|_2 \to \mathbf{0}$ as $t \to \infty$, and as it does so, the solution aligns in the direction of the eigenvector associated with $\lambda_1 = -1$, $\mathbf{v}_1 = [-1,1]^T$.

Now apply the forward Euler method to this problem. From the general expression (5.12), the iterate $\mathbf{x}_k$ can be written in the basis of eigenvectors as

$$\begin{aligned}\mathbf{x}_k &= c_1(1+h\lambda_1)^k\begin{bmatrix} -1 \\ 1 \end{bmatrix} + c_2(1+h\lambda_2)^k\begin{bmatrix} 2 \\ -1 \end{bmatrix} \\ &= c_1(1-h)^k\begin{bmatrix} -1 \\ 1 \end{bmatrix} + c_2(1-100h)^k\begin{bmatrix} 2 \\ -1 \end{bmatrix}.\end{aligned}$$

To obtain a numerical solution $\{\mathbf{x}_k\}$ that mimics the asymptotic behavior of the true solution, $\mathbf{x}(t) \to \mathbf{0}$, one must choose h sufficiently small that $|1+h\lambda_1| = |1-h| < 1$ and $|1+h\lambda_2| = |1-100h| < 1$. The first condition requires $h \in (0,2)$, while the second condition is far more restrictive: $h \in (0,1/50)$. The more restrictive condition describes the values of h that will give $\mathbf{x}_k \to \mathbf{0}$ for all $\mathbf{x}_0$.

Take note of this phenomenon: *the faster a component decays from the true solution (like e^{-100t} in our example), the smaller the time step must be for the forward Euler method (and other explicit schemes).*



Some iterates $\mathbf{x}_k$ of the forward Euler method for the example with $\lambda_1 = -1$ and $\lambda_2 = -100$, using time-step $h = 0.021$. This time-step is slightly larger than the stability limit $h < 0.02$, so $\|\mathbf{x}_k\|_2 \to \infty$ as k increases. Moreover, the solution aligns in the direction of the the eigenvector associated with the most unstable eigenvalue, $\mathbf{v}_2 = [2,-1]^T$.

Problems for which $\mathbf{A}$ has eigenvalues with significantly different magnitudes are called *stiff differential equations*. For such problems, implicit methods – which generally have much larger stability regions – are generally favored.

Thus far we have only sought $\mathbf{x}_k \to \mathbf{0}$ as $k \to \infty$. In some cases, we merely wish for $\mathbf{x}_k$ to be bounded. In this case, it is acceptable to have an eigenvalue $h\lambda_j$ on the boundary of the absolute stability region of a method, provided it is not a repeated eigenvalue (more precisely, provided it is associated with 1×1 Jordan blocks, i.e., it is not *defective*).

5.8.3 Closing example: heat equation

We can draw together many themes from this course in one elementary but vital example, the solution of the linear partial differential equation

$$u_t(x,t) = u_{xx}(x,t)$$

posed on the domain $x \in [0,1]$ and $t \geq 0$. This *heat equation* equation models the temperature in a long bar. The length of the bar is spanned by the x variable; t refers to time, starting from $t = 0$ when the state of the system is specified,

$$u(x,0) = U_0(x), \quad x \in [0,1].$$

Quenching both ends of this bar in an ice bath equates to the *homogeneous Dirichlet* boundary conditions

$$u(0,t) = u(1,t) = 0.$$

One common approach to numerically solving this equation is called the *method of lines*. Here is the basic idea: discretize the continuum $x \in [0,1]$ into a set of discrete points, then approximate u_{xx} by a finite difference approximation in the x variable. This discretization converts the original *partial* differential equation into *ordinary* differential equations, which can be readily solved using any of the techniques discussed in this chapter.

Let us get more specific. Discretize $[0,1]$ into the points $x_0, \ldots, x_{n+1}$, uniformly spaced a distance

$$\Delta x = \frac{1}{n+1}$$

apart from one another,

$$x_j = j\Delta x, \qquad j = 0, \ldots, n+1.$$

1. Recall from Section 1.7 that one can approximate derivatives by differentiating interpolating polynomials. Indeed, in Section 1.7 we

already saw how the second derivative can be approximated via the difference

$$u_{xx}(x_j,t) \approx \frac{u(x_{j-1},t) - 2u(x_j,t) + u(x_{j+1},t)}{(\Delta x)^2}, \tag{5.13}$$

which is just an adaptation of equation (1.22). In making these approximations, we shall no longer have access to the values of the exact solution such as $u(x_j,t)$, so we will introduce a set of new functions

$$u_j(t) \approx u(x_j,t), \qquad j = 0,\ldots,n+1.$$

Then (5.13) becomes

$$u_{xx}(x_j,t) \approx \frac{u_{j-1}(t) - 2u_j(t) + u_{j+1}(t)}{(\Delta x)^2}. \tag{5.14}$$

Notice that the boundary conditions from the differential equation,

$$u(0,t) = u(1,t) = 0, \qquad t \geq 0,$$

directly imply that $u_0(t) = u_{n+1}(t) = 0$ for all $t \geq 0$. Now our goal is to find $u_j(t)$ for $j = 1,\ldots,n$.

This explains why this approach is called the 'method of lines'. It will develop approximations $u_j(t)$ to the solution $u(x,t)$ on 'lines' of constant $x = x_j$ values in the (x,t) plane.

2. Now recall the partial differential equation $u_t(x,t) = u_{xx}(x,t)$. Replacing $u_{xx}(x,t)$ with the finite difference approximation, the differential equation suggests that we find $u_j(t)$ so that

$$\frac{\partial}{\partial t}u_j(t) = \frac{u_{j-1}(t) - 2u_j(t) + u_{j+1}(t)}{(\Delta x)^2}, \qquad j = 1,\ldots,n.$$

These equations form a coupled linear system. Perhaps it helps to write them out individually for a few values of j:

$$\begin{aligned}
\frac{\partial}{\partial t}u_1(t) &= \frac{u_0(t) - 2u_1(t) + u_2(t)}{(\Delta x)^2} = \frac{-2u_1(t) + u_2(t)}{(\Delta x)^2} \\
\frac{\partial}{\partial t}u_2(t) &= \frac{u_1(t) - 2u_2(t) + u_3(t)}{(\Delta x)^2} \\
&\vdots \\
\frac{\partial}{\partial t}u_{n-1}(t) &= \frac{u_{n-2}(t) - 2u_{n-1}(t) + u_n(t)}{(\Delta x)^2} \\
\frac{\partial}{\partial t}u_n(t) &= \frac{u_{n-1}(t) - 2u_n(t) + u_{n+1}(t)}{(\Delta x)^2} = \frac{u_{n-1}(t) - 2u_n(t)}{(\Delta x)^2}.
\end{aligned}$$

Compile these equations into the matrix form

$$\frac{\partial}{\partial t}\begin{bmatrix} u_1(t) \\ u_2(t) \\ \vdots \\ u_{n-1}(t) \\ u_n(t) \end{bmatrix} = \frac{1}{(\Delta x)^2}\begin{bmatrix} -2 & 1 & & & \\ 1 & -2 & \ddots & & \\ & \ddots & \ddots & 1 & \\ & & 1 & -2 & 1 \\ & & & 1 & -2 \end{bmatrix}\begin{bmatrix} u_1(t) \\ u_2(t) \\ \vdots \\ u_{n-1}(t) \\ u_n(t) \end{bmatrix},$$

which we summarize as

$$\mathbf{u}'(t) = \mathbf{A}\mathbf{u}(t),$$

Note that $\mathbf{A}$ contains the leading $1/(\Delta x)^2$ factor.

a system of linear ordinary differential equations, with initial condition derived from initial data $U_0(x)$ given for this problem:

$$\mathbf{u}_0 = \mathbf{u}(0) = \begin{bmatrix} u_1(0) \\ \vdots \\ u_n(0) \end{bmatrix} = \begin{bmatrix} U_0(x_1) \\ \vdots \\ U_0(x_n) \end{bmatrix}.$$

3. This approximation is called the *semi-discretized* form of the problem, since it is discretized in space, but time remains a continuous variable. Now one could quickly express the solution using the exponential of the matrix $\mathbf{A}$, as discussed earlier in this section,

$$\mathbf{u}(t) = \mathrm{e}^{t\mathbf{A}}\mathbf{u}(0).$$

However, for large $\mathbf{A}$ computation of $\mathrm{e}^{t\mathbf{A}}$ (e.g., using MATLAB's `expm` command), is quite expensive, and so we wish to *approximate the solution of the ordinary differential equation* using one of the techniques studied in this chapter.

Such large $\mathbf{A}$ arise when we have partial differential equations in two and three physical dimensions. The one-dimensional example here is easy by comparison.

For example, one could fix a time-step $\Delta t > 0$ and seek an approximation

$$\mathbf{u}_k \approx \mathbf{u}(t_k).$$

The forward Euler method gives

$$\mathbf{u}_{k+1} = \mathbf{u}_k + (\Delta t)\mathbf{A}\mathbf{u}_k,$$

while the (implicit) backward Euler method

$$\mathbf{u}_{k+1} = \mathbf{u}_k + (\Delta t)\mathbf{A}\mathbf{u}_{k+1}$$

leads to the linear system of equations

$$(\mathbf{I} - (\Delta t)\mathbf{A})\mathbf{u}_{k+1} = \mathbf{u}_k$$

that must be solved (e.g., via Gaussian elimination) at each step to find $\mathbf{u}_k$.

If Δt is fixed, then one would compute a Cholesky or LU factorization of $\mathbf{I} - \Delta t\mathbf{A}$, thus expediting the solution of this system at each step. If $\mathbf{A}$ is banded, as it is in this example, such factorizations are very fast.

The main question, then, is: given a choice of numerical integrator (forward Euler, backward Euler, etc.), how large can the time step Δt be to maintain stability?

4. To answer this question, diagonalize $\mathbf{A}$ to reveal its eigenvalues. Thankfully, explicit formulas are available for these eigenvalues,

$$\lambda_j = \frac{2\cos(j\pi/(n+1)) - 2}{(\Delta x)^2}, \qquad j = 1, \ldots, n;$$

the associated eigenvectors have a beautifully elegant formula:

$$\mathbf{v}_j = \begin{bmatrix} \sin\left(\frac{j\pi}{n+1}\right) \\ \sin\left(\frac{2j\pi}{n+1}\right) \\ \vdots \\ \sin\left(\frac{nj\pi}{n+1}\right) \end{bmatrix} = \begin{bmatrix} \sin(jx_1) \\ \sin(jx_2) \\ \vdots \\ \sin(jx_n) \end{bmatrix}, \qquad j = 1, \ldots, n.$$

These eigenvalues are all real. The rightmost eigenvalue is

$$\lambda_1 = \frac{2\cos(\pi/(n+1)) - 2}{(\Delta x)^2} \approx \frac{2(1 - \pi^2/(2(n+1)^2)) - 2}{(\Delta x)^2} = -\pi^2,$$

Here we use the Taylor approximation for cosine: as $\theta \to 0$,

$$\cos(\theta) = 1 - \frac{1}{2}\theta^2 + \mathcal{O}(\theta)^4.$$

while the leftmost eigenvalue is

$$\lambda_n = \frac{2\cos(n\pi/(n+1)) - 2}{(\Delta x)^2} \approx \frac{-4}{(\Delta x)^2} = -4(n+1)^2.$$

Notice that the eigenvalues of $\mathbf{A}$ are all negative, so

$$\mathbf{u}(t) = \mathrm{e}^{t\mathbf{A}}\mathbf{u}(0) \to \mathbf{0}$$

as $t \to \infty$. However, as n increases, the leftmost eigenvalue increases in magnitude like $\mathcal{O}(n^2)$.

n	λ_1	λ_n
16	$-9.841548\ldots$	$-1146.158\ldots$
32	$-9.862152\ldots$	$-4346.137\ldots$
64	$-9.867683\ldots$	$-16890.132\ldots$

It will help our later discussion to visualize the eigenvectors of $\mathbf{A}$. Figure 5.15 shows all the eigenvectors for $n = 16$.

5. Consider the implications of these eigenvalues for the forward Euler method: since $\lambda_n > -4(n+1)^2$,

$$\lambda_n = \frac{2\cos(n\pi/(n+1)) - 2}{(\Delta x)^2} > -4(n+1)^2,$$

the eigenvalues of $\mathbf{A}$ are contained in the interval

$$[-4(n+1)^2, 0].$$

To ensure stability of the method, choose the forward Euler time-step Δt so that for all $\lambda \in [-4(n+1)^2, 0]$, $\lambda\Delta t$ is in the stability

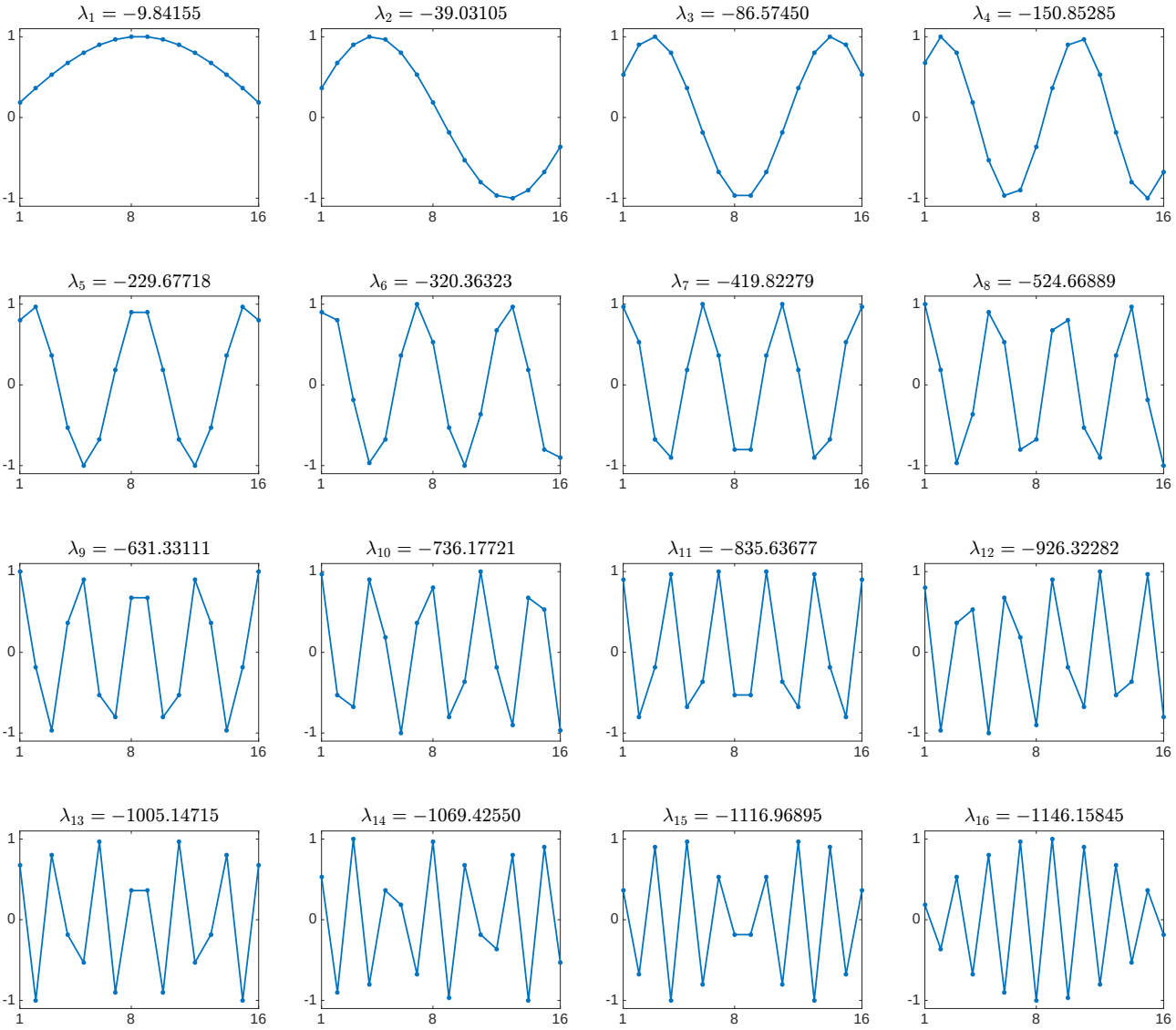


Figure 5.15: Eigenvectors of the matrix $\mathbf{A}$ from the method of lines discretization of the heat equation, for $n = 16$. The plot for λ_j shows the n entries of the corresponding eigenvector $\mathbf{v}_j$. (Lines are drawn between the entries in each eigenvector to help you appreciate that these eigenvectors approximate continuous functions of $x \in [0, 1]$ as $n \to \infty$.) Notice that as the eigenvalue gets increasingly negative, the eigenvector increases in frequency (i.e., it oscillates more rapidly).

region for the method. Since $\lambda \Delta t \in \mathbb{R}$, we need only consider the intersection of the forward Euler stability region with the real axis, i.e., $(-2, 0)$. To get

$$\lambda \Delta t \in (-2, 0)$$

for all $\lambda \in (-4(n+1)^2, 0)$, take

$$\Delta t \leq \frac{1}{2(n+1)^2} = \frac{(\Delta x)^2}{2}.$$

This is a crucial condition:

If you halve Δx, you must quarter Δt.

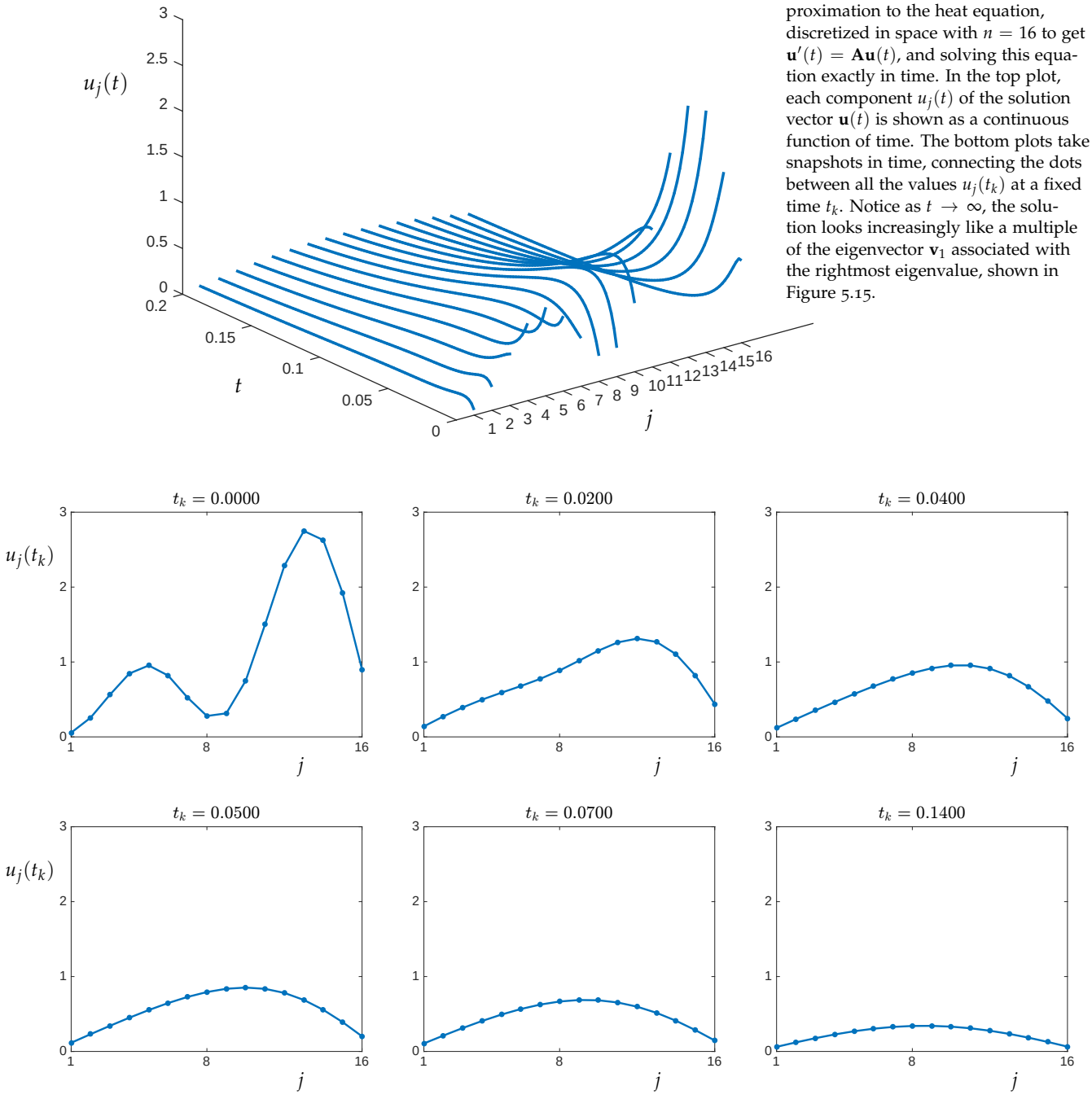


Figure 5.16: Method of lines approximation to the heat equation, discretized in space with $n = 16$ to get $\mathbf{u}'(t) = \mathbf{A}\mathbf{u}(t)$, and solving this equation exactly in time. In the top plot, each component $u_j(t)$ of the solution vector $\mathbf{u}(t)$ is shown as a continuous function of time. The bottom plots take snapshots in time, connecting the dots between all the values $u_j(t_k)$ at a fixed time t_k. Notice as $t \to \infty$, the solution looks increasingly like a multiple of the eigenvector $\mathbf{v}_1$ associated with the rightmost eigenvalue, shown in Figure 5.15.

Thus, improving the spatial resolution of the discretization requires extreme refinement in the time-step for forward Euler. This requirement is known as the *CFL condition* for this problem, named after its discovers Richard Courant, Kurt Otto Friedrichs, and Hans Lewy (1928).

6. Figure 5.16 shows the solution of the heat equation with initial

condition $U_0(x) = 10x^2(1-x)(1.2+\sin(3\pi x))$. The eigenvalues and eigenvectors explain the behavior seen in these plots. From our earlier discussion, recall that the exact solution of the semi-discretized problem is

$$\mathbf{u}(t) = \mathrm{e}^{t\mathbf{A}}\mathbf{u}(0) = \sum_{j=1}^{n} c_j \mathrm{e}^{t\lambda_j}\mathbf{v}_j, \tag{5.15}$$

where the coefficients $\{c_j\}$ come from $\mathbf{c} = \mathbf{V}^{-1}\mathbf{u}(0)$. As with the two-dimensional example considered in Section 5.8.2, the formula (5.15) explains how $\mathbf{u}(t)$ behaves as $t \to 0$. As t increases, the sum will be increasingly dominated by the term involving the rightmost eigenvalue, $\lambda_1 \approx -\pi^2$. Indeed, presuming $c_1 \neq 0$, we expect

$$\mathbf{u}(t) \sim c_1 \mathrm{e}^{t\lambda_1}\mathbf{v}_1,$$

so the solution should increasingly resemble the eigenvector $\mathbf{v}_1$ as t increases. This is evident in Figure 5.16.

Now solve this same problem using the forward Euler method. In the eigenvector basis, the forward Euler approximation is

$$\mathbf{u}_k = (\mathbf{I} + (\Delta t)\mathbf{A})^k \mathbf{u}_0 = \sum_{j=1}^{n} c_j (1 + (\Delta t)\lambda_j)^k \mathbf{v}_j. \tag{5.16}$$

One gains a very rich insight by combining this formula with some insight from analysis. A smooth initial condition $U_0(x)$ to the original problem, which satisfies the boundary conditions $U_0(0) = U_0(1) = 0$, can be approximated very well with a Fourier sine series. Similarly, the discretized initial condition $\mathbf{u}_0$ on the spatial grid is approximated well by the leading eigenvectors of $\mathbf{A}$ (which are samples of sine functions). You can intuitively appreciate this fact by comparing the initial condition in Figure 5.16 (corresponding to $t_k = 0$) with the eigenvectors shown in Figure 5.15. The initial condition much more closely resembles the first few (smooth) eigenvectors than it does the highly oscillatory eigenvectors corresponding to the most negative eigenvalues. Figure 5.17 shows the decay of the c_j coefficients for $n = 16$ and $U_0(x) = 10x^2(1-x)(1.2+\sin(3\pi x))$.

There is a deep tie to the *eigenfunctions* of the underlying differential operator $Lu = -u''$, defined on the domain of twice differentiable functions satisfying the homogeneous Dirichlet boundary conditions.

With $n = 16$, we previously noted that $\lambda_n = -1146.158\ldots$, so to maintain stability the forward Euler method requires

$$\Delta t < 0.001744959\ldots.$$

Figure 5.18 shows the result of running forward Euler with a time-step $\Delta t = 0.002$ that is *slightly too large*. The computation proceeds reasonably for the first ten time steps or so, but by $k = 20$ the

forward Euler iterate $\mathbf{u}_k$ begins to bear the fingerprint of a high-frequency oscillation. By the time $k = 70$, that instability completely dominates the solution. (Notice the different vertical scales in the $k = 35$ and $k = 70$ plots: the instability is growing fast!) Compare the approximation $\mathbf{u}_{70}$ to the eigenvector $\mathbf{v}_{16}$ shown in Figure 5.15. The close resemblance is no coincidence. In the eigenvector basis expansion (5.16) for $\mathbf{u}_k$, the dominant term will be

$$c_n(1 + (\Delta t)\lambda_n)^k \mathbf{v}_n,$$

since $|1 + (\Delta t)\lambda_n|$ is larger than any of the other $|1 + (\Delta t)\lambda_j|$ values. Why does this term not start dominating right away? Because the magnitude of c_n is much smaller than other c_j values, as seen in Figure 5.17. Thus it takes a number of iterations before the growth of $(1 + (\Delta t)\lambda_n)^k$ counteracts the small value of c_n. When k gets sufficiently large, this λ_n term completely takes over.

Lest we end the lecture on a negative note, Figure 5.19 shows the result of running the forward Euler method with time-step $\Delta t = 0.0015$, just within the stability condition. The solution $\mathbf{u}_k$, for the same values of k as shown in Figure 5.18, looks much closer to the solution from the matrix exponential shown in Figure 5.16.

An astute reader might notice one subtlety. This Δt obeys the stability condition,

$$|1 + (\Delta t)\lambda_j| < 1, \quad j = 1, \ldots, n,$$

but the term corresponding to λ_n is still the largest of these terms

$$|1 + (\Delta t)\lambda_n)| > |1 + (\Delta t)\lambda_j|, \qquad j = 1, \ldots, n-1.$$

Thus the $(1 + (\Delta t)\lambda_n)^k$ term decays most slowly in the sum (5.16). Will it not eventually dominate, causing the solution to again

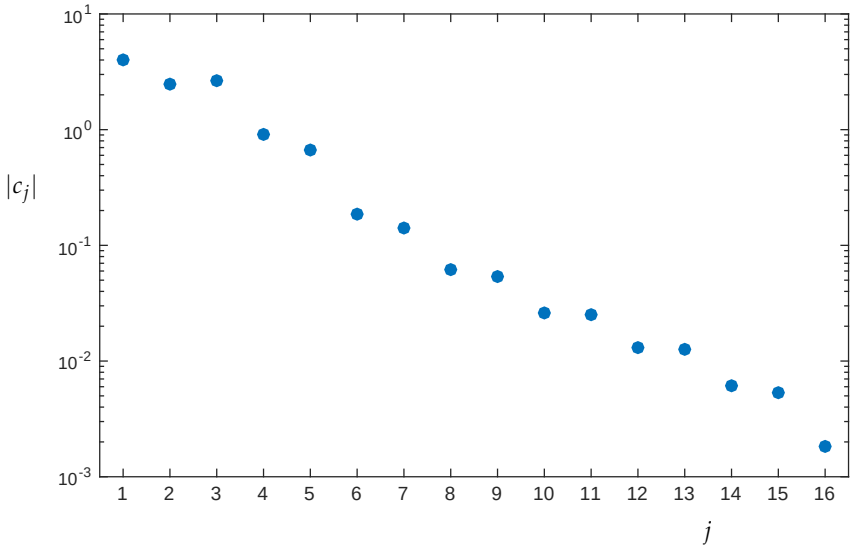


Figure 5.17: The magnitude of the coefficients c_j for the initial condition $\mathbf{u}_0$ expanded in the basis of eigenvectors of $\mathbf{A}$, i.e., $\mathbf{c} = \mathbf{V}^{-1}\mathbf{u}_0$. Notice that the coefficients decrease rapidly as j increases: the essence of a smooth initial condition is captured by the eigenvectors associated with the rightmost eigenvalues.

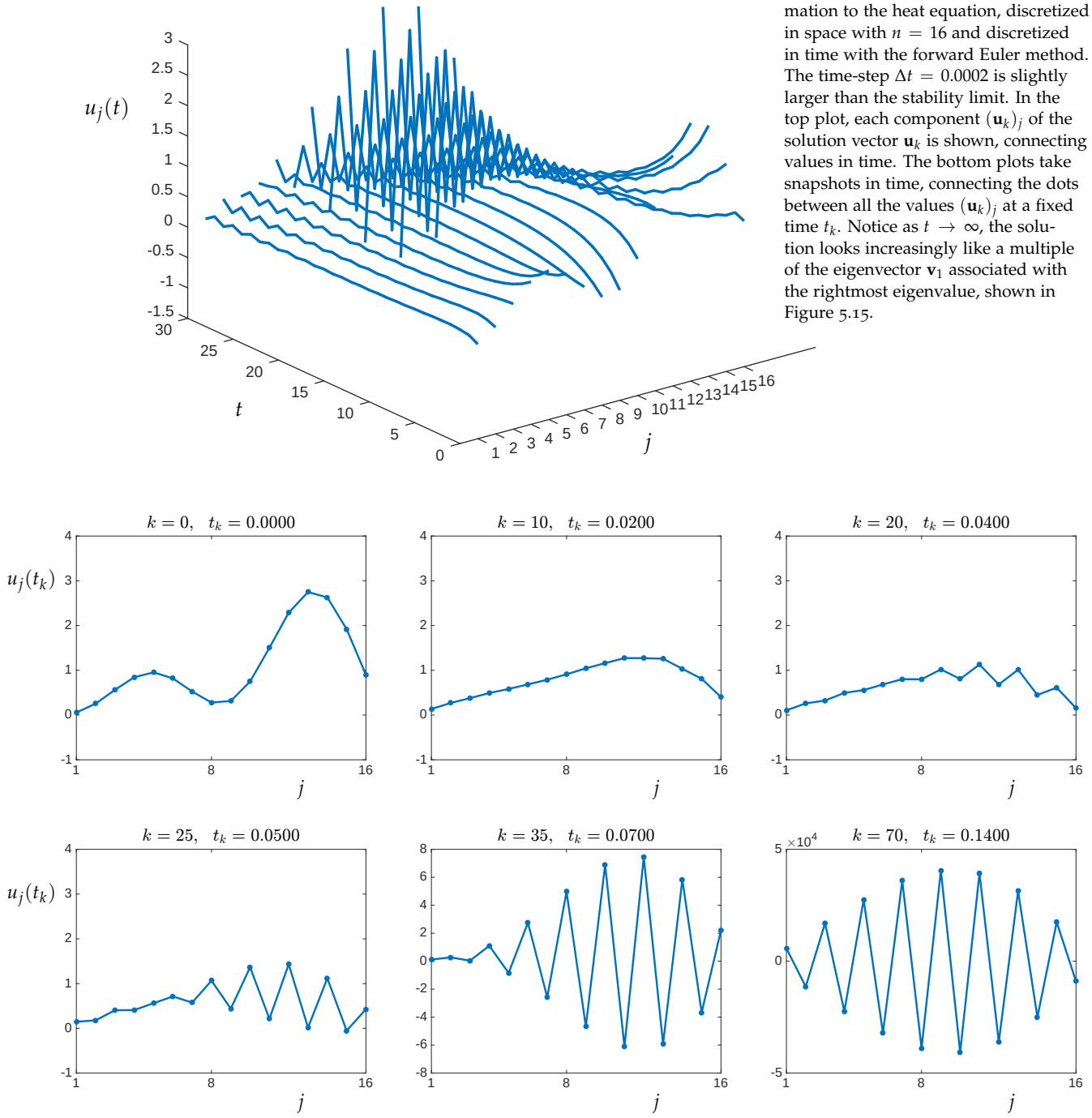


Figure 5.18: Method of lines approximation to the heat equation, discretized in space with $n = 16$ and discretized in time with the forward Euler method. The time-step $\Delta t = 0.0002$ is slightly larger than the stability limit. In the top plot, each component $(\mathbf{u}_k)_j$ of the solution vector $\mathbf{u}_k$ is shown, connecting values in time. The bottom plots take snapshots in time, connecting the dots between all the values $(\mathbf{u}_k)_j$ at a fixed time t_k. Notice as $t \to \infty$, the solution looks increasingly like a multiple of the eigenvector $\mathbf{v}_1$ associated with the rightmost eigenvalue, shown in Figure 5.15.

resemble the shape of $\mathbf{v}_n$ as it decays to zero? Yes, indeed: but the small value of c_n delays this effect, often well beyond the span of time we care about. Moreover, all the terms are getting smaller as k increases. However, if you take enough steps and zoom in on the small magnitude solution, you will indeed see $\mathbf{v}_n$ emerge.

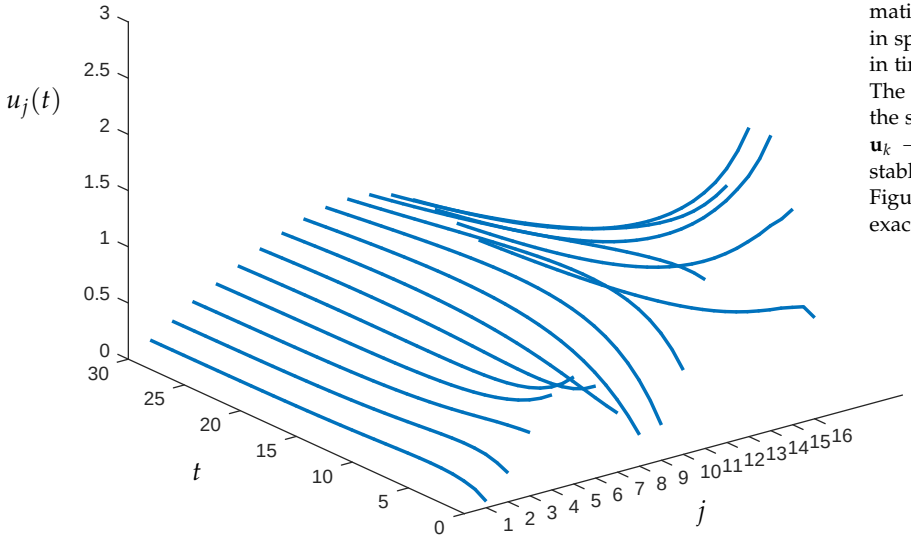
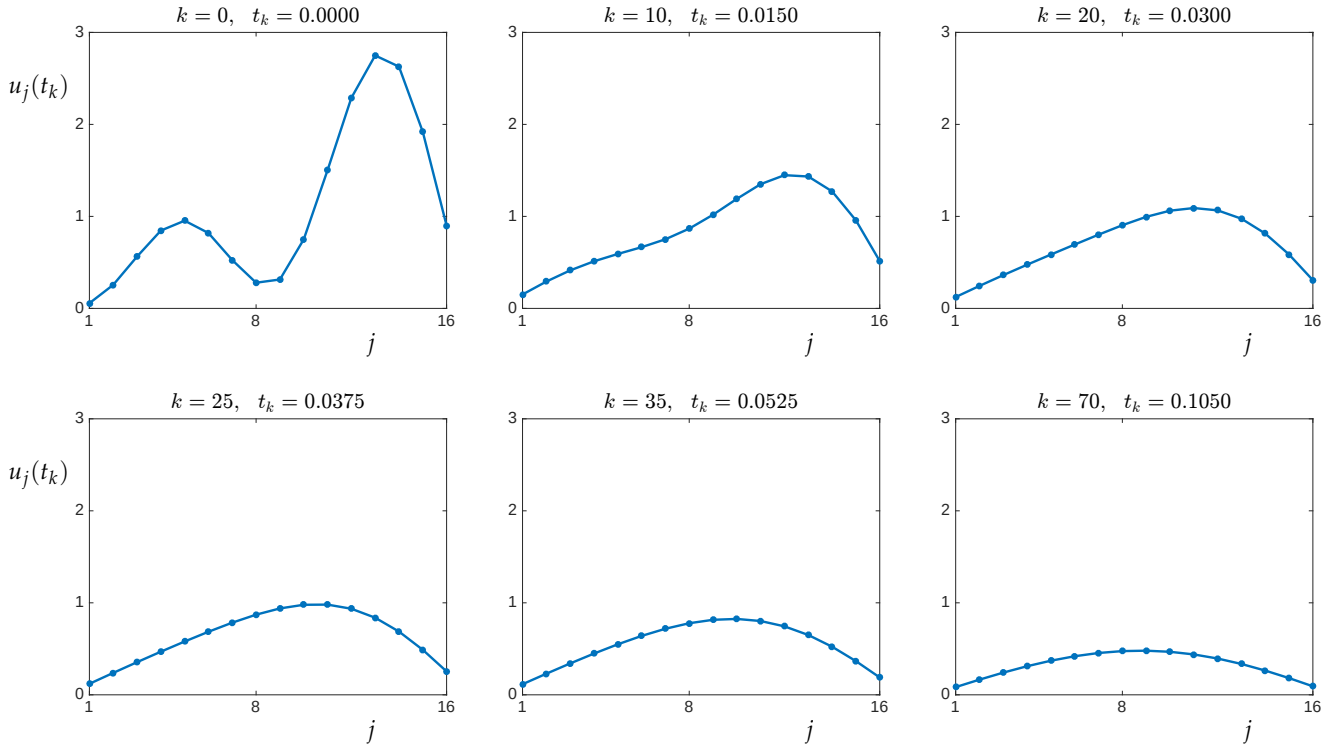


Figure 5.19: Method of lines approximation to the heat equation, discretized in space with $n = 16$ and discretized in time with the forward Euler method. The time-step $\Delta t = 0.00015$ obeys the stability condition, ensuring that $\mathbf{u}_k \to \mathbf{0}$ as $k \to \infty$. Compare these stable computations to the solution in Figure 5.16, which solves the equation exactly in time.



I hope our investigations this semester have given you a taste of the beautiful mathematics that empower numerical computations, the discrimination to pick the right algorithm to suit your given problem, the insight to identify those problems that are inherently ill-conditioned, and the tenacity to always seek clever, efficient solutions.
