Lecture Notes

CAAM 453

Numerical Analysis I

Rice University

Mark Embree

Lecture 1: Introduction to Numerical Analysis

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 down 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 are able to construct *approximate* solutions to these otherwise intractable problems with remarkable speed. Trefethen defines *numerical analysis* to be 'the study of algorithms for the problems of continuous mathematics'.[†] This course takes a tour through many such algorithms, sampling a variety of techniques suitable across many applications. We aim to assess alternative methods based on both accuracy and efficiency, to discern well-posed problems from ill-posed ones, and to see these methods in action through computer implementation.

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

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

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

Though numerical analysis has flourished in the past sixty years, its roots go back centuries, where numerical approximations were necessary for foundational work 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 (often women) 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:

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

> 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 the notions of continuity, differentiability, Taylor expansion, and convergence of sequences and series. Should you need to improve your analysis background, we recommend

- Walter Rudin, *Principles of Mathematical Analysis*, 3rd ed., McGraw–Hill, New York, 1976.

The methods we study typically require continuous variables to be approximated at finitely many points, that is, *discretized*. Nonlinearities are often finessed by *linearization*. These two compromises reduce a wide range of equations to familiar finite-dimensional, linear algebra problems, and thus we organize our study around a set of fundamental matrix algorithms that we revisit and refine as the semester progresses.

Use the following wonderful books to hone your matrix analysis skills:

- Peter Lax, *Linear Algebra*, Wiley, New York, 1997;
- Carl Meyer, *Applied Matrix Analysis and Linear Algebra*, SIAM, Philadelphia, 2000;
- Gilbert Strang, *Linear Algebra and Its Applications*, 3rd Ed., Harcourt, 1988.

These lecture notes were developed for a course that was supplemented by two texts: *Numerical Linear Algebra* by Trefethen and Bau, and either *Numerical Analysis* by Kincaid and Cheney, or *An Introduction to Numerical Analysis* by Süli and Mayers. These notes have benefited from this pedigree, and thus 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.

Lecture 2: Matrix Analysis and Norms

1. Orthogonal Matrix Factorization and Applications.

Amid the millennial fever that swept a decade ago, a group of numerical analysts compiled a list of the 20th Century's 'Top 10 Algorithms' (see *Computing in Science & Engineering* vol. 2 (2000)). Prominent among these is 'the decompositional approach to matrix computations', a clunky name but an essential idea that has enabled great progress in numerical analysis over the past 50 years.

Before this breakthrough, typical matrix problems were solved by focusing on manipulations made to the individual entries of a matrix, a recipe for tedium. In place of this technical approach, algorithms are now formulated by first *decomposing* a generic matrix into the product of several simpler matrices, each of which is much easier to work with than the original matrix. This new perspective facilitates the design and analysis of algorithms; at some point one must inevitably handle individual elements, but this should not be the initial focus.

We begin this course by studying one of these decompositions, the *QR factorization*—a matrix $\mathbf{A}$ can be written as the product of a unitary matrix and an upper triangular matrix. This tool will reappear throughout the semester as we solve linear systems, least squares problems, and eigenvalue problems. We shall also study other matrix factorizations, including the singular value decomposition (SVD), the LU decomposition, and the Schur decomposition.

Before we embark on our study of such algorithms, we must review a few basic concepts from matrix theory and establish a method for measuring the size of vectors and matrices.

1.1. Concepts from matrix analysis.

Throughout these notes, matrices are denoted by bold capital letters; column vectors are denoted with bold lower case letters; scalars will never be bold, and almost always will be lower case. The matrix $\mathbf{A}$ with m rows and n columns with real entries is denoted $\mathbf{A} \in \mathbb{R}^{m\times n}$; if $\mathbf{A}$ has complex entries, we write $\mathbf{A} \in \mathbb{C}^{m\times n}$. The element of $\mathbf{A}$ in row j and column k is denoted a_{jk}.

A set of vectors $\mathcal{U} \subset \mathbb{C}^n$ is a *subspace* if it is closed under vector addition and scalar multiplication. That is, (1) for all $\mathbf{u}_1, \mathbf{u}_2 \in \mathcal{U}$, we also have $\mathbf{u}_1 + \mathbf{u}_2 \in \mathcal{U}$, and (2) for all $\mathbf{u} \in \mathcal{U}$ and $\alpha \in \mathbb{C}$, we have $\alpha\mathbf{u} \in \mathcal{U}$. (When only working with real numbers, replace $\alpha \in \mathbb{C}$ by $\alpha \in \mathbb{R}$.)

We now consider several important subspaces associated with a matrix $\mathbf{A} \in \mathbb{C}^{m\times n}$. The *range* (or *column space*) of $\mathbf{A}$ is defined as

$$\mathrm{Ran}(\mathbf{A}) = \{\mathbf{A}\mathbf{x} : \mathbf{x} \in \mathbb{C}^n\} \subseteq \mathbb{C}^m.$$

The *kernel* (or *null space*) of $\mathbf{A}$ is defined as

$$\mathrm{Ker}(\mathbf{A}) = \{\mathbf{x} \in \mathbb{C}^n : \mathbf{A}\mathbf{x} = \mathbf{0}\} \subseteq \mathbb{C}^n.$$

The *rank* of $\mathbf{A}$, denoted $\mathrm{rank}(\mathbf{A})$, is the dimension of the range of $\mathbf{A}$, i.e., the number of linearly independent vectors in a basis for $\mathrm{Ran}(\mathbf{A})$. We say the square matrix $\mathbf{A} \in \mathbb{C}^{n\times n}$ is *nonsingular* provided the following equivalent conditions hold:

- $\mathbf{A}^{-1}$ exists;
- $\mathrm{Ran}(\mathbf{A}) = \mathbb{C}^n$ (i.e., $\mathrm{rank}(\mathbf{A})) = n$);
- $\mathrm{Ker}(\mathbf{A}) = \{\mathbf{0}\}$;
- $\mathbf{A}$ has no zero eigenvalues.

The *identity matrix* is always denoted by $\mathbf{I}$, the *zero matrix* by $\mathbf{0}$.

The transpose of a vector $\mathbf{x} \in \mathbb{C}^n$ is denoted by $\mathbf{x}^T$, and the conjugate-transpose is denoted by $\mathbf{x}^*$; thus, $\mathbf{x}^* = \overline{\mathbf{x}}^T \in \mathbb{C}^{1\times n}$. Note that $\mathbf{x}^T$ and $\mathbf{x}^*$ are row vectors. (If $\mathbf{x}$ has only real entries, then $\mathbf{x}^T = \mathbf{x}^*$.) The conjugate-transpose generalizes to matrices, where for $\mathbf{A} \in \mathbb{C}^{m\times n}$, we have $\mathbf{A}^* = \overline{\mathbf{A}}^T \in \mathbb{C}^{n\times m}$.

Two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{C}^n$ are *orthogonal* provided $\mathbf{x}^*\mathbf{y} = 0$. Two subspaces $\mathcal{U} \subseteq \mathbb{C}^n$ and $\mathcal{V} \subseteq \mathbb{C}^n$ are orthogonal provided $\mathbf{u}^*\mathbf{v} = 0$ for all $\mathbf{u} \in \mathcal{U}$ and $\mathbf{v} \in \mathcal{V}$. If $\mathcal{U}$ and $\mathcal{V}$ are orthogonal, we write $\mathcal{U} \perp \mathcal{V}$. The set of all vectors $\mathbf{v} \in \mathbb{C}^n$ that are orthogonal to the subspace $\mathcal{U}$ is denoted by

$$\mathcal{U}^\perp = \{\mathbf{v} \in \mathbb{C}^n : \mathbf{v}^*\mathbf{u} = 0 \text{ for all } \mathbf{u} \in \mathcal{U}\}.$$

Suppose that $\mathbf{x} \in \mathrm{Ran}(\mathbf{A})$ and $\mathbf{z} \in \mathrm{Ker}(\mathbf{A}^*)$. Then there exists some $\mathbf{y} \in \mathbb{C}^n$ such that $\mathbf{x} = \mathbf{A}\mathbf{y}$, and we have $(\mathbf{z}^*\mathbf{x})^* = (\mathbf{z}^*\mathbf{A}\mathbf{y})^* = \mathbf{y}^*\mathbf{A}^*\mathbf{z} = \mathbf{y}^*\mathbf{0} = 0$, which implies $\mathrm{Ran}(\mathbf{A}) \perp \mathrm{Ker}(\mathbf{A}^*)$. Swapping the role of $\mathbf{A}$ and $\mathbf{A}^*$ gives $\mathrm{Ran}(\mathbf{A}^*) \perp \mathrm{Ker}(\mathbf{A})$. After considering the dimensions of these spaces, we arrive at what Gilbert Strang calls the *Fundamental Theorem of Linear Algebra*:

$$\begin{aligned} \mathbb{C}^m &= \mathrm{Ran}(\mathbf{A}) \oplus \mathrm{Ker}(\mathbf{A}^*), \qquad \mathrm{Ran}(\mathbf{A}) \perp \mathrm{Ker}(\mathbf{A}^*) \\ \mathbb{C}^n &= \mathrm{Ran}(\mathbf{A}^*) \oplus \mathrm{Ker}(\mathbf{A}), \qquad \mathrm{Ran}(\mathbf{A}^*) \perp \mathrm{Ker}(\mathbf{A}). \end{aligned}$$

The singular value decomposition, which will be covered in a few weeks, provides a natural means for untangling the four fundamental subspaces $\mathrm{Ran}(\mathbf{A})$, $\mathrm{Ran}(\mathbf{A}^*)$, $\mathrm{Ker}(\mathbf{A})$ and $\mathrm{Ker}(\mathbf{A}^*)$, and observing their orthogonality.

It is worth noting some important classes of square matrices:

- $\mathbf{A} \in \mathbb{C}^{n\times n}$ is *Hermitian* provided $\mathbf{A}^* = \mathbf{A}$. $\mathbf{A}$ is *symmetric* provided $\mathbf{A}^T = \mathbf{A}$. (For real matrices, these terms can be used interchangeably. However, some applications, e.g., in electromagnetics, give rise to complex symmetric matrices, which lack many of the fine properties enjoyed by Hermitian matrices.)
- $\mathbf{Q} \in \mathbb{C}^{n\times n}$ is *unitary* provided $\mathbf{Q}^*\mathbf{Q} = \mathbf{I}$. Since $\mathbf{Q}$ is square, we must have that $\mathbf{Q}^{-1} = \mathbf{Q}^*$, and hence also that $\mathbf{Q}\mathbf{Q}^* = \mathbf{I}$. A real unitary matrix is also called an *orthogonal matrix*. (Notice that if $\mathbf{Q}^*\mathbf{Q} = \mathbf{I} \in \mathbb{C}^{n\times n}$ for the rectangular matrix $\mathbf{Q} \in \mathbb{C}^{m\times n}$ with $m > n$, then $\mathbf{Q}\mathbf{Q}^* \neq \mathbf{I} \in \mathbb{C}^{m\times m}$. Can you explain why?)

1.1.1. Vector and matrix norms.

As we study numerical *analysis*, we shall require a means of measuring distance.

Definition. A function $\|\cdot\| : \mathbb{C}^n \to \mathbb{R}$ is a *norm* provided:

- $\|\mathbf{x}\| \geq 0$ for all $\mathbf{x} \in \mathbb{C}^n$; $\|\mathbf{x}\| = 0$ if and only if $\mathbf{x} = \mathbf{0}$ (positivity);
- $\|\alpha\mathbf{x}\| = |\alpha|\|\mathbf{x}\|$ for all $\alpha \in \mathbb{C}$ and $\mathbf{x} \in \mathbb{C}^n$ (scaling);
- $\|\mathbf{x}+\mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$ for all $\mathbf{x}, \mathbf{y} \in \mathbb{C}^n$ (triangle inequality).

In this course, the only vector norms we shall use are the *p-norms*:

$$\|\mathbf{x}\|_p = \Big(\sum_{j=1}^n |x_j|^p\Big)^{1/p}$$

for $p \ge 1$. The values $p = 1, 2, \infty$ are by far the most common:

$$\|\mathbf{x}\|_1 = \sum_{j=1}^n |x_j|; \qquad \|\mathbf{x}\|_2 = \Big(\sum_{j=1}^n |x_j|^2\Big)^{1/2} = \sqrt{\mathbf{x}^*\mathbf{x}}; \qquad \|\mathbf{x}\|_\infty = \max_{j=1,\ldots,n} |x_j|.$$

Among innumerable useful norm relationships, pride of place belongs to (1) the *Cauchy–Schwarz* inequality:

$$|\mathbf{x}^*\mathbf{y}| \;\le\; \|\mathbf{x}\|_2\|\mathbf{y}\|_2,$$

with equality holding when $\mathbf{x}$ and $\mathbf{y}$ are collinear, and (2) the *Pythagorean Theorem*: for orthogonal $\mathbf{x}$ and $\mathbf{y}$,

$$\|\mathbf{x}+\mathbf{y}\|_2^2 = \|\mathbf{x}\|_2^2 + \|\mathbf{y}\|_2^2.$$

The vector 2-norm enjoys another important property: it is *unitarily invariant*: for any unitary matrix $\mathbf{U} \in \mathbb{C}^{n\times n}$,

$$\|\mathbf{U}\mathbf{x}\|_2 = \|\mathbf{x}\|_2$$

for all $\mathbf{x} \in \mathbb{C}^n$. This is easy to prove: since $\mathbf{U}^*\mathbf{U} = \mathbf{I}$,

$$\|\mathbf{U}\mathbf{x}\|_2^2 = (\mathbf{U}\mathbf{x})^*(\mathbf{U}\mathbf{x}) = \mathbf{x}^*\mathbf{U}^*\mathbf{U}\mathbf{x} = \mathbf{x}^*\mathbf{x} = \|\mathbf{x}\|_2^2.$$

This fact has a natural 'physical' interpretation: If $\mathbf{U} = [\mathbf{u}_1\ \mathbf{u}_2\ \cdots\ \mathbf{u}_n]$ is unitary, then its columns $\mathbf{u}_1, \ldots, \mathbf{u}_n$ form an orthonormal basis for $\mathbb{C}^n$. The vector $\mathbf{U}\mathbf{x} = \sum_{j=1}^n x_j\mathbf{u}_j$ is a representation of the vector $\mathbf{x}$ in the coordinate system whose axes are given by $\mathbf{u}_1, \ldots, \mathbf{u}_n$. The statement $\|\mathbf{U}\mathbf{x}\|_2 = \|\mathbf{x}\|_2$ simply means, 'the length of $\mathbf{x}$ does not change when we convert from the standard orthonormal basis (columns of the identity matrix) to the new orthonormal basis (columns of $\mathbf{U}$).'

We can use norms to measure the magnitude of a matrix. The same axioms stated above for vector norms apply here.

Definition. A function $\|\cdot\| : \mathbb{C}^{m\times n} \to \mathbb{R}$ is a *matrix norm* provided:

- $\|\mathbf{A}\| \ge 0$ for all $\mathbf{A} \in \mathbb{C}^{m\times n}$; $\|\mathbf{A}\| = 0$ if and only if $\mathbf{A} = \mathbf{0}$ (positivity);
- $\|\alpha\mathbf{A}\| = |\alpha|\|\mathbf{A}\|$ for all $\alpha \in \mathbb{C}$ and $\mathbf{A} \in \mathbb{C}^{m\times n}$ (scaling);
- $\|\mathbf{A}+\mathbf{B}\| \le \|\mathbf{A}\| + \|\mathbf{B}\|$ for all $\mathbf{A}, \mathbf{B} \in \mathbb{C}^{m\times n}$ (triangle inequality).

The most important class of norms are the *induced* matrix norms, which are defined in terms of some vector norm $\|\cdot\|$:

$$\|\mathbf{A}\| = \max_{\mathbf{x}\ne\mathbf{0}} \frac{\|\mathbf{A}\mathbf{x}\|}{\|\mathbf{x}\|}.$$

That is, induced matrix norms measure the maximum amount a matrix can stretch a vector beyond its original length. It is often handy to use the equivalent definition:

$$\|\mathbf{A}\| = \max_{\|\mathbf{x}\|=1} \|\mathbf{A}\mathbf{x}\|.$$

The matrix norms induced by the vector p-norms are particularly useful. When $p = 1$ and $p = \infty$, we have the simple formulas

$$\|\mathbf{A}\|_1 = \max_{1\le k\le n} \sum_{j=1}^{m} |a_{jk}|; \qquad \|\mathbf{A}\|_\infty = \max_{1\le j\le m} \sum_{k=1}^{n} |a_{jk}|.$$

That is, the 1-norm is the maximum absolute column sum, while the ∞-norm is the maximum absolute row sum. Most useful for many applications is the induced matrix 2-norm (called by some the *spectral norm*):

$$\|\mathbf{A}\|_2 = \max_{\mathbf{x}\ne\mathbf{0}} \frac{\|\mathbf{A}\mathbf{x}\|_2}{\|\mathbf{x}\|_2}.$$

The matrix 2-norm inherits *unitary invariance* from the vector 2-norm: for any unitary matrices $\mathbf{U}$ and $\mathbf{V}$, $\|\mathbf{UAV}\|_2 = \|\mathbf{A}\|_2$. We shall derive a formula for the 2-norm from the singular value decomposition in a few weeks:

$$\|\mathbf{A}\|_2 = \max\{\sqrt{\lambda} : \lambda \text{ is an eigenvalue of } \mathbf{A}^*\mathbf{A}\}.$$

For values of p other than 1, 2, and ∞, there is no simple formula for the induced matrix p-norm.

Trefethen and Bau present a very nice graphical description of induced matrix norms in their Figure 3.1 (page 20), which we highly recommend. (See `norm_demo.m` on the class website.)

The useful *Frobenius norm* is not induced by any vector norm:

$$\|\mathbf{A}\|_F = \Big(\sum_{j=1}^{m}\sum_{k=1}^{n} |a_{jk}|^2\Big)^{1/2}.$$

Many matrix norms are *submultiplicative*,

$$\|\mathbf{AB}\| \le \|\mathbf{A}\|\,\|\mathbf{B}\|,$$

a property that enables much analysis. (In fact, some authors add this condition as a fourth requirement in the definition of a 'matrix norm'.) To prove that the induced matrix norms are submultiplicative, observe that

$$\|\mathbf{AB}\| = \max_{\mathbf{x}\ne\mathbf{0}} \frac{\|\mathbf{ABx}\|}{\|\mathbf{x}\|} = \max_{\mathbf{x}\ne\mathbf{0}} \frac{\|\mathbf{ABx}\|}{\|\mathbf{Bx}\|}\frac{\|\mathbf{Bx}\|}{\|\mathbf{x}\|} \le \Big(\max_{\mathbf{x}\ne\mathbf{0}} \frac{\|\mathbf{ABx}\|}{\|\mathbf{Bx}\|}\Big)\Big(\max_{\mathbf{x}\ne\mathbf{0}} \frac{\|\mathbf{Bx}\|}{\|\mathbf{x}\|}\Big) \le \|\mathbf{A}\|\|\mathbf{B}\|.$$

(Can you explain both inequalities, and confirm that division by zero is not a major concern?)

The Frobenius norm, too, is submultiplicative. However, there exist norms that satisfy the three basic matrix norm axioms, but are not submultiplicative, e.g.,

$$\|\mathbf{A}\| = \max_{j,k} |a_{jk}|$$

satisfies the positivity, scaling, and triangle inequality properties, yet one can construct $\mathbf{A}$ and $\mathbf{B}$ such that $\|\mathbf{AB}\| > \|\mathbf{A}\|\,\|\mathbf{B}\|$. (Try it!)

► An extensive discussion of vector and matrix norms can be found in Chapter 5 of *Matrix Analysis* by R. A. Horn and C. R. Johnson, Cambridge, 1985. For a sophisticated treatment of the class of *unitarily invariant norms*, see Chapter 4 of *Matrix Analysis* by R. Bhatia, Springer, 1997.

Lecture 3: Projectors and Reflectors

1.1.2. Projectors.

Definition. A matrix $\mathbf{P} \in \mathbb{C}^{n\times n}$ is a *projector* provided $\mathbf{P}^2 = \mathbf{P}$. If $\mathbf{P}$ is also Hermitian, then $\mathbf{P}$ is an *orthogonal projector*. (Matrix powers imply repeated matrix multiplication: $\mathbf{P}^2 = \mathbf{PP}$, etc.)

In this course, we shall be chiefly concerned with orthogonal projectors, which have appealing analytical and numerical properties.[†] The term 'orthogonal' refers to the nature of the projection: any vector orthogonal to the range of $\mathbf{P}$ is projected to the zero vector. Recall that the Fundamental Theorem of Linear Algebra ensures that $\mathrm{Ran}(\mathbf{P}) \perp \mathrm{Ker}(\mathbf{P}^*)$. When $\mathbf{P}$ is Hermitian, i.e., $\mathbf{P} = \mathbf{P}^*$, then $\mathrm{Ker}(\mathbf{P}^*) = \mathrm{Ker}(\mathbf{P})$, so $\mathrm{Ran}(\mathbf{P}) \perp \mathrm{Ker}(\mathbf{P})$.

Example. For any nonzero $\mathbf{v} \in \mathbb{C}^n$, the matrix

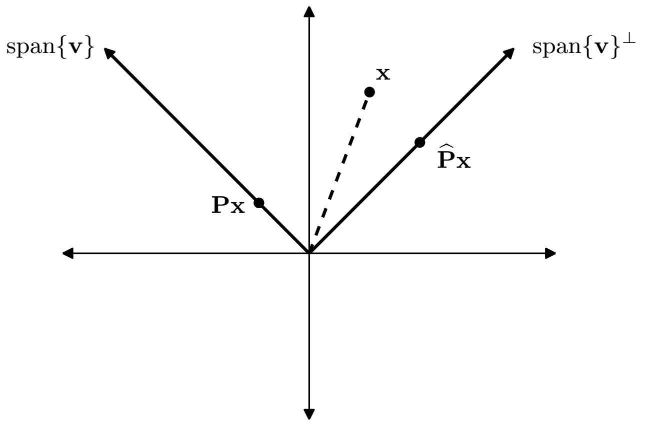
$$\mathbf{P} = \frac{\mathbf{vv}^*}{\mathbf{v}^*\mathbf{v}}$$

is an orthogonal projector. Note that $\mathrm{Ran}(\mathbf{P}) = \mathrm{span}\{\mathbf{v}\}$, while $\mathrm{Ker}(\mathbf{P}) = \mathrm{span}\{\mathbf{v}\}^\perp = \{\mathbf{y} \in \mathbb{C}^n : \mathbf{y}^*\mathbf{v} = 0\}$. The matrix

$$\widehat{\mathbf{P}} = \mathbf{I} - \mathbf{P} = \mathbf{I} - \frac{\mathbf{vv}^*}{\mathbf{v}^*\mathbf{v}}$$

is also an orthogonal projector, with $\mathrm{Ran}(\widehat{\mathbf{P}}) = \mathrm{span}\{\mathbf{v}\}^\perp$ and $\mathrm{Ker}(\widehat{\mathbf{P}}) = \mathrm{span}\{\mathbf{v}\}$.

The illustration below shows the effect of the projectors $\mathbf{P}$ and $\widehat{\mathbf{P}}$ in $\mathbb{R}^2$.



Example. For vectors $\mathbf{u}, \mathbf{v} \in \mathbb{C}^n$ with $\mathbf{u}^*\mathbf{v} \neq 0$, the matrix

$$\mathbf{\Pi} = \frac{\mathbf{vu}^*}{\mathbf{u}^*\mathbf{v}},$$

is a projector. When $\mathbf{u}$ and $\mathbf{v}$ are not collinear, $\mathbf{\Pi}$ is not Hermitian, and hence it is an *oblique* (not orthogonal) projector. For this example, $\mathrm{Ran}(\mathbf{\Pi}) = \mathrm{span}\{\mathbf{v}\}$ and $\mathrm{Ker}(\mathbf{\Pi}) = \mathrm{span}\{\mathbf{u}\}^\perp$. Can you replicate the sketch above, but now using this oblique projector?

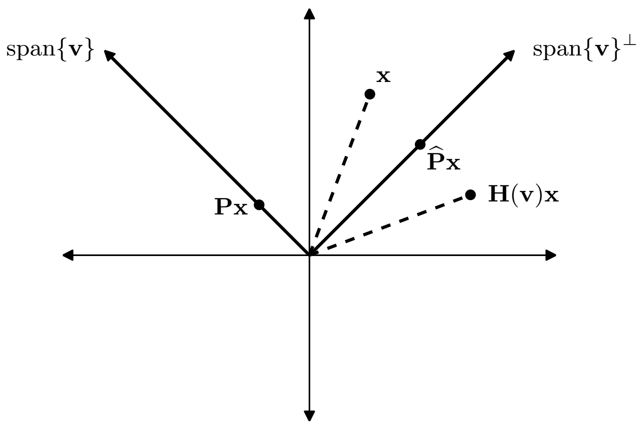
[†] Be sure to note the difference between *orthogonal projectors* and *orthogonal matrices*. The latter term refers to unitary matrices with real entries. Such matrices have full rank, and all columns are orthogonal and have norm 1. Orthogonal projectors have rank less than n (except in the trivial case of $\mathbf{P} = \mathbf{I}$), and the columns have norm less than or equal to one.

1.2. QR factorization.

The QR factorization is the first matrix decomposition we shall study. The goal is to write any $\mathbf{A} \in \mathbb{C}^{m\times n}$ in the form $\mathbf{A} = \mathbf{QR}$, where $\mathbf{Q} \in \mathbb{C}^{m\times m}$ is unitary and $\mathbf{R} \in \mathbb{C}^{m\times n}$ is *upper triangular*, i.e., $r_{jk} = 0$ if $j > k$. This decomposition is a workhorse: it will enable the efficient and stable solution of linear systems and least squares problems. Moreover, this factorization forms the cornerstone of the classic QR algorithm for eigenvalue computations.

1.2.1. Householder reflectors.

Householder reflectors are unitary matrices that are closely allied with the orthogonal projectors $\mathbf{P}$ and $\widehat{\mathbf{P}}$ discussed above. We wish to reflect a given vector $\mathbf{x} \in \mathbb{C}^n$ across an $n-1$ dimensional hyperplane. Perhaps this is most easily explained by amending the two-dimensional projector illustration given above.



We wish to reflect $\mathbf{x} \in \mathbb{C}^n$ over the $n-1$ dimensional subspace $\text{span}\{\mathbf{v}\}^\perp$. We will encode this operation in the matrix $\mathbf{H}(\mathbf{v}) \in \mathbb{C}^{n\times n}$, so that $\mathbf{H}(\mathbf{v})\mathbf{x} \in \mathbb{C}^n$ is the reflected vector.

How should $\mathbf{H}(\mathbf{v})$ be constructed? Look again at the illustration above. If we subtract $\mathbf{Px}$ from $\mathbf{x}$, (using simple head-to-tail vector subtraction) we get halfway to our goal – that is, we get $\widehat{\mathbf{P}}\mathbf{x} \in \text{span}\{\mathbf{v}\}^\perp$. To get the complete reflection across $\text{span}\{\mathbf{v}\}^\perp$, we simply subtract $\mathbf{Px}$ once more. In summary, this gives

$$\mathbf{H}(\mathbf{v})\mathbf{x} \;=\; \mathbf{x} - 2\,\mathbf{Px} \;=\; (\mathbf{I} - 2\mathbf{P})\mathbf{x} \;=\; \Big(\mathbf{I} - 2\,\frac{\mathbf{vv}^*}{\mathbf{v}^*\mathbf{v}}\Big)\mathbf{x}.$$

Definition. The matrix

$$\mathbf{H}(\mathbf{v}) = \mathbf{I} - 2\frac{\mathbf{vv}^*}{\mathbf{v}^*\mathbf{v}}$$

is called a *Householder reflector*.[‡] For any $\mathbf{x} \in \mathbb{C}^n$, the vector $\mathbf{H}(\mathbf{v})\mathbf{x}$ is the reflection of $\mathbf{x}$ over the $n-1$ dimensional hyperplane $\text{span}\{\mathbf{v}\}^\perp$.

Notice in the above figure that $\mathbf{H}(\mathbf{v})\mathbf{x}$ has the same Euclidean length (i.e., 2-norm) as $\mathbf{x}$. This is no accident; it is a consequence of the fact that $\mathbf{H}(\mathbf{v})$ is a unitary matrix. You can verify this algebraically,

$$\mathbf{H}(\mathbf{v})^*\mathbf{H}(\mathbf{v}) = \Big(\mathbf{I} - 2\frac{\mathbf{vv}^*}{\mathbf{v}^*\mathbf{v}}\Big)\Big(\mathbf{I} - 2\frac{\mathbf{vv}^*}{\mathbf{v}^*\mathbf{v}}\Big) = \mathbf{I} - 2\frac{\mathbf{vv}^*}{\mathbf{v}^*\mathbf{v}} - 2\frac{\mathbf{vv}^*}{\mathbf{v}^*\mathbf{v}} + 4\frac{\mathbf{vv}^*\mathbf{vv}^*}{\mathbf{v}^*\mathbf{vv}^*\mathbf{v}} = \mathbf{I} - 4\frac{\mathbf{vv}^*}{\mathbf{v}^*\mathbf{v}} + 4\frac{\mathbf{vv}^*}{\mathbf{v}^*\mathbf{v}} = \mathbf{I}.$$

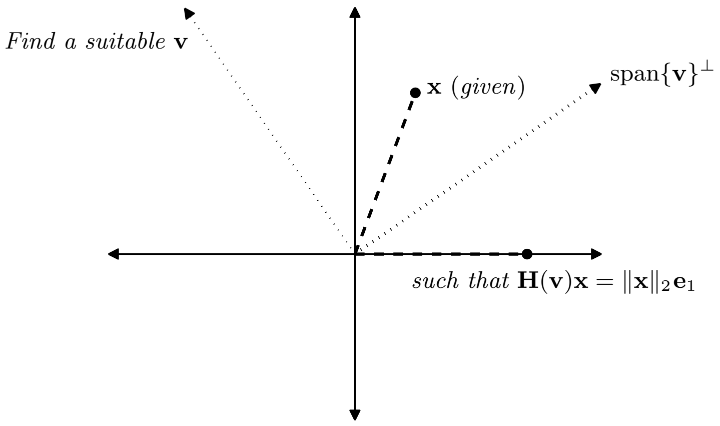
[‡]The reflectors are named for Alston Householder, who proposed this reflection and the QR factorization itself in the seminal 4-page paper "Unitary Triangularization of a Nonsymmetric Matrix," *J. ACM* 5 (1958) 339–342.

You can also appeal to geometric intuition: if you reflect a vector twice over the same hyperplane, you must get back to exactly where you started, so $\mathbf{H}(\mathbf{v})^2 = \mathbf{I}$. Since $\mathbf{H}(\mathbf{v})$ is Hermitian, this implies that $\mathbf{H}(\mathbf{v})^*\mathbf{H}(\mathbf{v}) = \mathbf{I}$, i.e., $\mathbf{H}(\mathbf{v})$ is unitary.

1.2.2. Using Householder reflectors to zero entries of a vector.

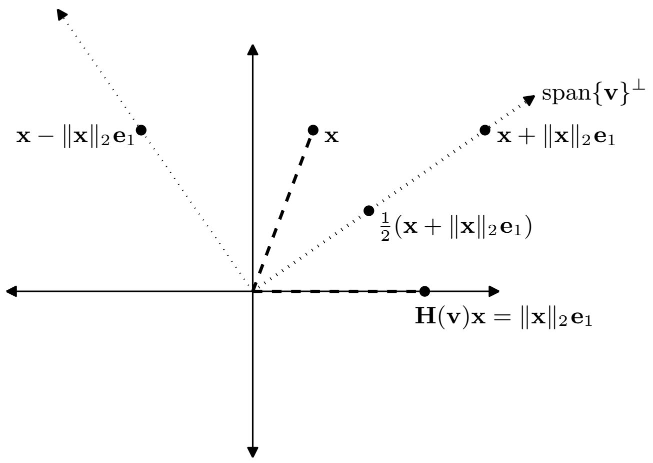
Householder reflectors are the fundamental tool needed for QR factorization. We shall see that the entire operation can be reduced to one sub-problem: Given a vector $\mathbf{x}$, find the vector $\mathbf{v}$ such that the reflector $\mathbf{H}(\mathbf{v})$ maps $\mathbf{x}$ to the vector $\|\mathbf{x}\|_2\mathbf{e}_1$, where $\mathbf{e}_1 = [1, 0, \ldots, 0]^T$; i.e.,

$$\mathbf{H}(\mathbf{v})\mathbf{x} = \begin{bmatrix} \|\mathbf{x}\|_2 \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$


Find a suitable $\mathbf{v}$
$\mathbf{x}$ (*given*)
span$\{\mathbf{v}\}^\perp$
such that $\mathbf{H}(\mathbf{v})\mathbf{x} = \|\mathbf{x}\|_2\mathbf{e}_1$

Here is one way to derive $\mathbf{v}$, illustrated in the figure below. Note that the midpoint $\frac{1}{2}(\mathbf{x} + \|\mathbf{x}\|_2\mathbf{e}_1)$ between $\mathbf{x}$ and $\|\mathbf{x}\|_2\mathbf{e}_1$ must lie on span$\{\mathbf{v}\}^\perp$, and so too must the collinear point $\mathbf{x} + \|\mathbf{x}\|_2\mathbf{e}_1$. To find $\mathbf{v}$, we simply need to find a vector that is both orthogonal to $\mathbf{x} + \|\mathbf{x}\|_2\mathbf{e}_1$ and in the plane spanned by $\mathbf{x}$ and $\mathbf{e}_1$.[§] For real $\mathbf{x}$, verify (by hand, and also by eye on the plot below) that a suitable choice for this orthogonal vector is

$$\mathbf{v} = \mathbf{x} - \|\mathbf{x}\|_2\mathbf{e}_1.$$


$\mathbf{x} - \|\mathbf{x}\|_2\mathbf{e}_1$
$\mathbf{x}$
span$\{\mathbf{v}\}^\perp$
$\mathbf{x} + \|\mathbf{x}\|_2\mathbf{e}_1$
$\frac{1}{2}(\mathbf{x} + \|\mathbf{x}\|_2\mathbf{e}_1)$
$\mathbf{H}(\mathbf{v})\mathbf{x} = \|\mathbf{x}\|_2\mathbf{e}_1$

[§]If $\mathbf{v}$ is orthogonal to $\mathbf{x} + \|\mathbf{x}\|_2\mathbf{e}_1$ but not in the plane spanned by $\mathbf{x}$ and $\mathbf{e}_1$, then $\mathbf{x}$ will not generally be reflected to $\|\mathbf{x}\|_2\mathbf{e}_1$. To observe this, one has to look in three (or more) dimensions.

We shall not dwell on the case of complex $\mathbf{x}$, which requires a bit more care. (For completeness, note that we can set $\mathbf{v} = \mathbf{x} - \mathrm{e}^{\mathrm{i}\theta}\|\mathbf{x}\|\mathbf{e}_1$, giving $\mathbf{H}(\mathbf{v})\mathbf{x} = \mathrm{e}^{\mathrm{i}\theta}\|\mathbf{x}\|_2\mathbf{e}_1$, where $\theta = \arg(\mathbf{e}_1^*\mathbf{x})$.)

Example. We can verify the efficacy of the Householder reflector we have constructed with a simple MATLAB calculation.

```
>> x = [1;2;3;4;5]
x =
     1
     2
     3
     4
     5

>> norm(x)
ans =
    7.4162

>> v = x - norm(x)*[1;0;0;0;0]
v =
   -6.4162
    2.0000
    3.0000
    4.0000
    5.0000

>> Hv = eye(5)-2*v*v'/(v'*v)
Hv =
    0.1348    0.2697    0.4045    0.5394    0.6742
    0.2697    0.9159   -0.1261   -0.1681   -0.2102
    0.4045   -0.1261    0.8109   -0.2522   -0.3152
    0.5394   -0.1681   -0.2522    0.6638   -0.4203
    0.6742   -0.2102   -0.3152   -0.4203    0.4746

>> Hv*x
ans =
    7.4162
    0.0000
    0.0000
   -0.0000
    0.0000
```

Lecture 4: The QR Decomposition

1.2.3. QR decomposition.

The technique of reflecting a vector onto the first coordinate axis (i.e., zeroing out all but the first entry) that we developed in the previous lecture is the essential building block for our first method for constructing the QR decomposition of a general matrix $\mathbf{A} \in \mathbb{C}^{m\times n}$, arguably the most fundamental technique in all of numerical linear algebra. We wish to write $\mathbf{A} = \mathbf{QR}$, where $\mathbf{Q} \in \mathbb{C}^{m\times m}$ is a unitary matrix and $\mathbf{R} \in \mathbb{C}^{m\times n}$ is upper triangular (i.e., $r_{jk} = 0$ if $j > k$). In general, we shall assume that $m \ge n$ throughout; this covers the most common situations encountered in applications, and saves us from making a few technical caveats along the way.

We shall follow this methodology: repeatedly apply Householder reflectors on the left of $\mathbf{A}$, with the jth transformation zeroing entries below the diagonal entry in the jth column. Since the product of unitary matrices is also unitary,[†] the product of these Householder reflectors forms a unitary matrix; call it $\mathbf{Q}^*$. Then we have $\mathbf{Q}^*\mathbf{A} = \mathbf{R}$, which implies $\mathbf{A} = \mathbf{QR}$. Now, let us fill in the details.

For simplicity, we shall work in real arithmetic: suppose $\mathbf{A} \in \mathbb{R}^{m\times n}$, and write $\mathbf{A}$ in the form

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_1 & \widehat{\mathbf{A}}_1 \end{bmatrix},$$

where $\mathbf{a}_1 \in \mathbb{R}^m$ and $\widehat{\mathbf{A}}_1 \in \mathbb{R}^{m\times(n-1)}$. To construct a Householder transformation that will reflect $\mathbf{a}_1$, the first column of $\mathbf{A}$, onto the first coordinate direction $\mathbf{e}_1 \in \mathbb{R}^m$, we set

$$\mathbf{v}_1 = \mathbf{a}_1 - \|\mathbf{a}_1\|_2\mathbf{e}_1,$$

as described in the last lecture. This choice[‡] gives

$$\begin{aligned}\mathbf{H}(\mathbf{v}_1)\mathbf{A} &= \begin{bmatrix} \mathbf{H}(\mathbf{v}_1)\mathbf{a}_1 & \mathbf{H}(\mathbf{v}_1)\widehat{\mathbf{A}}_1 \end{bmatrix} \\ &= \begin{bmatrix} \|\mathbf{a}_1\|_2\mathbf{e}_1 & \mathbf{H}(\mathbf{v}_1)\widehat{\mathbf{A}}_1 \end{bmatrix}.\end{aligned}$$

Define $\mathbf{Q}_1 = \mathbf{H}(\mathbf{v}_1)$. Furthermore, set $r_{11} = \|\mathbf{a}_1\|_2$, let $[r_{12}, r_{13}, \ldots, r_{1n}]$ denote the first row of $\mathbf{Q}_1\widehat{\mathbf{A}}_1$, and let $\mathbf{A}_2 \in \mathbb{R}^{(m-1)\times(n-1)}$ denote the remaining portion of $\mathbf{H}(\mathbf{v}_1)\widehat{\mathbf{A}}_1$. With this notation, we have

$$\mathbf{Q}_1\mathbf{A} = \begin{bmatrix} r_{11} & r_{12} \quad \cdots \quad r_{1n} \\ 0 & \\ \vdots & \boxed{\quad \mathbf{A}_2 \quad} \\ 0 & \end{bmatrix}.$$

[†]Proof: If $\mathbf{Q}_1$ and $\mathbf{Q}_2$ are unitary, then $(\mathbf{Q}_1\mathbf{Q}_2)^*(\mathbf{Q}_1\mathbf{Q}_2) = \mathbf{Q}_2^*\mathbf{Q}_1^*\mathbf{Q}_1\mathbf{Q}_2 = \mathbf{I}$.

[‡]Actually, we could just as easily reflect $\mathbf{a}_1$ to the vector $-\|\mathbf{a}_1\|_2\mathbf{e}_1$. The choice of sign has important implications for *numerical stability*, i.e., the robustness of the algorithm to rounding errors on a computer. For this reason, one does not wish to reflect $\mathbf{a}_1$ to a nearby vector, as subtracting two like quantities can result in a phenomenon called *catastrophic cancellation* that we shall discuss in more detail in later lectures on floating point computer arithmetic. The choice of $\mathbf{v}$ that reflects $\mathbf{a}_1$ farthest is $\mathbf{v}_1 = \mathbf{a}_1 + \text{sign}(a_{11})\|\mathbf{a}_1\|_2\mathbf{e}_1$, i.e., $\mathbf{a}_1$ is reflected to $-\text{sign}(a_{11})\|\mathbf{a}_1\|_2\mathbf{e}_1$. For complex $\mathbf{A}$, we take $\mathbf{v}_1 = \mathbf{a}_1 \pm e^{i\theta}\|\mathbf{a}_1\|_2\mathbf{e}_1$, where θ is the argument of the first entry in $\mathbf{a}_1$.

The key now is to reflect the first column of $\mathbf{A}_2$ onto the first coordinate direction, $\mathbf{e}_1 \in \mathbb{R}^{m-1}$. On its own, this would be a simple procedure: Like before, partition $\mathbf{A}_2$ into

$$\mathbf{A}_2 = \begin{bmatrix} \mathbf{a}_2 & \widehat{\mathbf{A}}_2 \end{bmatrix}$$

with first column $\mathbf{a}_2 \in \mathbb{R}^{m-1}$. The required reflector then takes the form

$$\mathbf{v}_2 = \mathbf{a}_2 - \|\mathbf{a}_2\|_2 \mathbf{e}_1,$$

so that $\mathbf{H}(\mathbf{v}_2) \in \mathbb{R}^{(m-1)\times(m-1)}$.

However, to build a **QR** decomposition, we need to apply the reflector to $\mathbf{Q}_1\mathbf{A}$, not just to the submatrix $\mathbf{A}_2$. Procedures like this arise often in matrix decomposition algorithms, and they are a bit fragile: *we want to altar the submatrix* $\mathbf{A}_1$ *without disturbing the zero entries we have already created in the first column of* $\mathbf{Q}_1\mathbf{A}$. In this case, the fix is simple: define

$$\mathbf{Q}_2 = \begin{bmatrix} 1 & \mathbf{0} \\ \mathbf{0} & \mathbf{H}(\mathbf{v}_2) \end{bmatrix},$$

which one can verify is also a unitary matrix. Now, we have

$$\mathbf{Q}_2\mathbf{Q}_1\mathbf{A} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & \cdots & r_{1n} \\ 0 & r_{22} & r_{23} & \cdots & r_{2n} \\ 0 & 0 & & & \\ \vdots & \vdots & & \boxed{\quad\mathbf{A}_3\quad} & \\ 0 & 0 & & & \end{bmatrix}$$

with $\mathbf{A}_3 \in \mathbb{R}^{(m-2)\times(n-2)}$.

Now that we have two columns with zeros below the diagonal, the pattern for future eliminations should be clear. In general,

$$\mathbf{Q}_k = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{H}(\mathbf{v}_k) \end{bmatrix},$$

where $\mathbf{I}$ is the $(k-1)\times(k-1)$ identity matrix, and $\mathbf{H}(\mathbf{v}_k) \in \mathbb{R}^{(m-k+1)\times(m-k+1)}$. Since $\mathbf{A}$ has n columns, we require n reflectors[§] to zero out all subdiagonal entries of $\mathbf{A}$. All together, we have

$$\mathbf{Q}_n\mathbf{Q}_{n-1}\cdots\mathbf{Q}_1\mathbf{A} = \mathbf{R}.$$

Since our ultimate goal is to obtain a factorization of $\mathbf{A}$, we will now move all the unitary matrices to the right hand side of the equation. We can do this by multiplying on the left by $\mathbf{Q}_n^*$, then $\mathbf{Q}_{n-1}^*$, and so on. Since Householder reflectors are Hermitian matrices ($\mathbf{Q}_k^* = \mathbf{Q}_k$ for all k), we have

$$\mathbf{A} = \mathbf{Q}_1\mathbf{Q}_2\cdots\mathbf{Q}_n\mathbf{R}.$$

We define

$$\mathbf{Q} = \mathbf{Q}_1\mathbf{Q}_2\cdots\mathbf{Q}_n,$$

and thus arrive at

$$\mathbf{A} = \mathbf{Q}\mathbf{R}.$$

[§]If $m = n$, then the final column of $\mathbf{A}$ has no subdiagonal entries, so only $n-1$ transformations are necessary.

We have just produced a constructive proof to the following theorem.

Theorem. For any matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ with $m \geq n$, there exists a unitary matrix $\mathbf{Q} \in \mathbb{R}^{m\times m}$ and an upper triangular matrix $\mathbf{R} \in \mathbb{R}^{m\times n}$ such that $\mathbf{A} = \mathbf{QR}$.

The following MATLAB code provides a basic implementation of the QR algorithm described above. To ease your translation of mathematics into MATLAB, this code is given in an extremely inefficient manner. Can you find at least two or three ways to speed up this code while still producing the same `Q` and `R`?

```
  function [Q,R] = slow_householder_qr(A)
% Compute the QR factorization of real A using Householder reflectors
% ** implementation designed for clarity, not efficiency **

  [m,n] = size(A);
  Q = eye(m);
  for k=1:min(m-1,n)
     ak  = A(k:end,k);                               % vector to be zeroed out
     vk  = ak + sign(ak(1))*norm(ak)*[1;zeros(m-k,1)];  % construct v_k that defines the reflector
     Hk = eye(m-k+1) - 2*vk*vk'/(vk'*vk);            % construct reflector
     Qk = [eye(k-1) zeros(k-1,m-k+1); zeros(m-k+1,k-1) Hk];
     A = Qk*A;                                       % update A
     Q = Q*Qk;                                       % accumulate Q
  end
  R = A;
```

1.2.4. Built-in MATLAB commands for the QR decomposition.

While it is important to understand the fundamentals of the QR decomposition, you should not use `slow_householder_qr.m` (or your personal implementation) to compute industrial-strength QR decompositions. Excellent versions, both computationally efficient and stable to round-off errors, are available. The LAPACK library, a free collection of numerical linear algebra routines in FORTRAN, is the most popular source for such robust codes.[¶] These codes, which descend from the famous EISPACK and LINPACK libraries, have been written, refined, and tested over several decades; as such, it contains excellent implementations of the best algorithms.

MATLAB (from version 6.0 onward) uses compiled LAPACK routines for its basic linear algebra computations. To compute a QR decomposition of the matrix `A` in MATLAB, simply type

```
[Q,R] = qr(A);
```

Many applications give rise to matrices $\mathbf{A} \in \mathbb{R}^{m\times n}$ with m much larger than n. In this case, you might rightly be concerned about forming the $m \times m$ matrix $\mathbf{Q}$, which will require far more storage than the matrix $\mathbf{A}$ itself. (Moreover, columns $n+1, \ldots, m$ of $\mathbf{Q}$ are superfluous, in that they multiply against zero entries of $\mathbf{R}$.) There are two common solutions to this concern, details of which follow in the next lecture. First, you can simply store the vectors $\mathbf{v}_1$, $\mathbf{v}_2$, …, $\mathbf{v}_n$ used to form the Householder reflectors: with this data stored, you can compute the vector $\mathbf{Qx}$ for any $\mathbf{x}$ without explicitly forming the matrix $\mathbf{Q}$. The second approach is to compute the QR factorization

[¶]You can download LAPACK (and get reference information) from the NETLIB mathematical software repository, `www.netlib.org`.

through a completely different method, Gram-Schmidt orthogonalization. This procedure results in a *skinny QR decomposition*, $\mathbf{A} = \mathbf{QR}$ with $\mathbf{Q} \in \mathbb{C}^{m\times n}$, $\mathbf{R} \in \mathbb{C}^{n\times n}$, and $\mathbf{Q}^*\mathbf{Q} = \mathbf{I}$), which you can compute in matrix with the command

```
[Q,R] = qr(A,0);
```

Compare some timings in MATLAB for `qr(A)` and `qr(A,0)` for matrices $\mathbf{A}$ with $m \gg n$. This should convice you that for large m, you will generally want to avoid forming the full-size $\mathbf{Q}$!

Lecture 5: Householder QR details; QR via Gram–Schmidt

1.2.5. Computational complexity of the QR decomposition.

The following MATLAB program, an implementation of Trefethen and Bau's Algorithm 10.1, provides a more efficient implementation of the Householder QR factorization than the one presented in the last lecture.

```
  function [V,R] = householder_qr(A)

% Compute the QR factorization of real A using Householder reflectors
% See Trefethen and Bau, Numerical Linear Algebra, Algorithm 10.1 (page 73)
% The object V is a "cell array":  vk (a vector of length m-k+1) is stored in V{k}.

  [m,n] = size(A);
  Q = eye(m);
  for k=1:min(m-1,n)
     ak  = A(k:m,k);                                   %    vector to be zeroed out
     vk  = ak; vk(1) = vk(1) + sign(ak(1))*norm(ak);   % 1. construct vk that defines the reflector
     vk  = vk/norm(vk);                                % 2. normalize vk
     A(k:m,k:n) = A(k:m,k:n) - 2*vk*(vk'*A(k:m,k:n));  % 3. update A
     V{k} = vk;                                        %    store vk in a cell array
  end
  R = A;
```

How much time will it take this algorithm to run for a given m and n? An important aspect of numerical analysis is the determination of the *computational complexity* of a given algorithm. The first step, the only one considered in this course, involves counting floating point operations. (More subtle aspects of this craft include the analysis of how efficiently data is accessed through a computer's memory hierarchy, and how effectively the algorithm may be implemented on a parallel computer. Topics of this sort are addressed in CAAM 420 and 520.)

First we count the number of operations for the lines numbered `1`, `2`, and `3` in the `householder_qr` MATLAB code above. We let $\ell = m-k+1$ and $p = n-k+1$. We assume that all basic arithmetic operations (add, subtract, multiply, divide, and square root) are each accomplished in the same about of time. (In practice, the square root and sometimes division require considerably more time than the other operations.)

`1.` one 2-norm (2ℓ operations), one addition
`2.` one 2-norm (2ℓ operations), ℓ divisions
`3.` ℓ scalar multiplications (for `2*vk`)
p inner products of length ℓ vectors (for `vk'*A(k:m,k:n)`): ($2\ell p - p$ operations)
one outer product update (the subtraction): ($2\ell p$ operations for one multiply, add per entry)

Adding these up, we see that each pass of the loop requires $4\ell p + 6\ell - p + 1$ operations. (Recall that $\ell = m-k+1$ and $p = n-k+1$.) The $4\ell p$ term will dominate. Summing over $k = 1, \ldots, n$, we find the number of floating point operations required, to leading order, is

$$\sum_{k=1}^{n} 4\ell p \;=\; \sum_{k=1}^{n} 4(m-k+1)(n-k+1) \;\approx\; 2mn^2 - \frac{2n^3}{3}.$$

1.2.6. The matrix $\mathbf{Q}$.

Note that the `householder_qr` program does not compute the $m \times m$ matrix $\mathbf{Q}$. This is because one does not typically need (or want) compute the matrix $\mathbf{Q}$ explicitly. Instead, one simply saves the vectors `vk` at each step; from these vectors, one can compute products of the form $\mathbf{Qx}$ or $\mathbf{Q}^*\mathbf{b}$ more efficiently than if $\mathbf{Q}$ was itself computed.

For example, the following MATLAB code will compute $\mathbf{Q}^*\mathbf{b}$, using the cell array `V` computed by `householder_qr`. (This is Algorithm 10.2 in Trefethen and Bau, page 74.) We apply $\mathbf{Q}^* = \mathbf{Q}_n\mathbf{Q}_{n-1}\cdots\mathbf{Q}_1$ (using the notation from the last lecture) one Householder reflector at a time. Since the upper-left $(k-1)\times(k-1)$ block of $\mathbf{Q}_k \in \mathbb{C}^{m\times m}$ is the identity matrix, we need only work with entries k through m of $\mathbf{b}$ (that is, `b(k:m)`) at step k.

```
for k=1:min(m-1,n)
     b(k:m) = b(k:m) - 2*V{k}*(V{k}'*b(k:m));
end
```

How would this algorithm change if we wanted to work with $\mathbf{Q}$ instead of $\mathbf{Q}^*$?

In many applications the matrix $\mathbf{Q}$ is actually quite useful. Suppose that $r_{jj} \neq 0$ for $j = 1, \ldots, n$. Then $\mathbf{A}$ is *full rank*, i.e., all its columns are linearly independent. In this case, the first n columns of $\mathbf{Q} \in \mathbb{C}^{m\times m}$ form an orthonormal basis for $\text{Ran}(\mathbf{A})$. One could form $\mathbf{Q}$ explicitly from the Householder reflectors; an alternative, as we shall see, is to bypass the m-by-m matrix $\mathbf{Q}$, computing the leading n columns of $\mathbf{Q}$ directly. The result is called a 'skinny QR factorization'.

1.2.7. QR Decomposition via the Gram–Schmidt Algorithm.

While the Householder QR algorithm described in the last lecture is a novel idea for many students, there is another way to obtain a QR factorization that is probably more familiar, though perhaps you have never seen it written down in QR-style: the Gram–Schmidt algorithm for computing an orthonormal basis for a set of vectors.

Given a set of linearly independent vectors $\mathbf{a}_1, \ldots, \mathbf{a}_n \in \mathbb{C}^m$ for $m \geq n$, the Gram–Schmidt process constructs an orthonormal basis $\{\mathbf{q}_1, \ldots, \mathbf{q}_n\}$ for $\text{span}\{\mathbf{a}_1, \ldots, \mathbf{a}_n\}$ as follows:

$$\begin{aligned}
\mathbf{q}_1 &= \mathbf{a}_1/\|\mathbf{a}_1\|_2 \\[1em]
\widehat{\mathbf{q}}_2 &= (\mathbf{I} - \mathbf{q}_1\mathbf{q}_1^*)\mathbf{a}_2 \\
\mathbf{q}_2 &= \widehat{\mathbf{q}}_2/\|\widehat{\mathbf{q}}_2\|_2 \\[1em]
\widehat{\mathbf{q}}_3 &= (\mathbf{I} - \mathbf{q}_1\mathbf{q}_1^* - \mathbf{q}_2\mathbf{q}_2^*)\mathbf{a}_3 \\
\mathbf{q}_3 &= \widehat{\mathbf{q}}_3/\|\widehat{\mathbf{q}}_3\|_2 \\
&\;\;\vdots
\end{aligned}$$

Note that since $\|\mathbf{q}_j\|_2 = 1$, the matrix $\mathbf{q}_j\mathbf{q}_j^*$ is an orthogonal projector. So too is

$$\mathbf{I} - \mathbf{q}_1\mathbf{q}_1^* - \cdots - \mathbf{q}_k\mathbf{q}_k^*;$$

it projects onto the orthogonal complement of $\text{span}\{\mathbf{q}_1, \ldots, \mathbf{q}_k\}$. Hence, the Gram–Schmidt algorithm constructs a new vector $\mathbf{q}_{k+1}$ by projecting $\mathbf{a}_{k+1}$ onto the orthogonal complement of the previous basis vectors, and then normalizing.

Can you spot the underlying QR factorization? The notation is suggestive. Let

$$\mathbf{A} = [\mathbf{a}_1\ \mathbf{a}_2\ \cdots\ \mathbf{a}_n]$$

and

$$\mathbf{Q} = [\mathbf{q}_1\ \mathbf{q}_2\ \cdots\ \mathbf{q}_n]$$

and define $\mathbf{R}$ entrywise as

$$r_{j,k} = \begin{cases} \mathbf{q}_j^*\mathbf{a}_k, & j < k; \\ \|\widehat{\mathbf{q}}_j\|_2, & j = k; \\ 0, & j > k. \end{cases}$$

Thus the Gram–Schmidt factorization computes $\mathbf{A} = \mathbf{QR}$, with $\mathbf{Q} \in \mathbb{C}^{m\times n}$ and $\mathbf{R} \in \mathbb{C}^{n\times n}$; the columns of $\mathbf{Q}$ form an orthonormal basis for $\text{Ran}(\mathbf{A})$. When $m \neq n$, this matrix $\mathbf{Q}$ is not square, and hence it is not unitary. Since the columns of $\mathbf{Q}$ are orthonormal, we have $\mathbf{Q}^*\mathbf{Q} = \mathbf{I} \in \mathbb{C}^{n\times n}$, but $\mathbf{Q}\mathbf{Q}^* \neq \mathbf{I} \in \mathbb{C}^{m\times m}$. (Note the important difference between this $\mathbf{Q}$ and the square, unitary matrix $\mathbf{Q}$ one obtains from the Householder QR algorithm, for which $\mathbf{Q}^*\mathbf{Q} = \mathbf{Q}\mathbf{Q}^* = \mathbf{I} \in \mathbb{C}^{m\times m}$.)

The MATLAB implementation suggested by the above construction is known as the *classical Gram–Schmidt algorithm.*

```
function [Q,R] = cgs_qr(A)
% Computation of the "skinny" QR decomposition of an m-by-n matrix
% (m>=n) using the the Classical Gram-Schmidt algorithm.
% See Trefethen and Bau, Algorithm 8.1

 [m,n] = size(A);
 if m<n, fprintf('ERROR: A should be an m-by-n matrix with m >= n.\n'); end

 Q = zeros(m,n);
 R = zeros(n,n);
 for k=1:n
    Q(:,k) = A(:,k);
    for j=1:k-1
        R(j,k) = Q(:,j)'*A(:,k);
        Q(:,k) = Q(:,k) - R(j,k)*Q(:,j);
    end
    R(k,k) = norm(Q(:,k));
    Q(:,k) = Q(:,k)/R(k,k);
 end
```

As we shall see in class, this implementation turns out to have unfortunate numerical properties: the vectors $\mathbf{q}_1, \ldots, \mathbf{q}_n$ constructed as the columns of $\mathbf{Q}$ can actually be far from orthogonal, due to rounding errors in finite precision arithmetic. Notice that the classical Gram–Schmidt algorithm forms

$$\widehat{\mathbf{q}}_k = \mathbf{a}_k - r_{1,k}\mathbf{q}_1 - \cdots - r_{1,k-1}\mathbf{q}_{k-1},$$

where

$$r_{j,k} = \mathbf{q}_j^* \mathbf{a}_k. \tag{1}$$

Effectively, we obtain $\widehat{\mathbf{q}}_k$ by subtracting from $\mathbf{q}_k$ all the orthogonal projections of $\mathbf{a}_k$ along the directions $\mathbf{q}_1, \ldots, \mathbf{q}_{k-1}$. Notice, however, that for all $j = 1, \ldots, k-1$,

$$\mathbf{q}_j^* \mathbf{a}_k = \mathbf{q}_j^*(\mathbf{a}_k - r_{1,k}\mathbf{q}_1 - \cdots - r_{j-1,k}\mathbf{q}_{j-1})$$

since $\mathbf{q}_j^* \mathbf{q}_\ell = 0$ for all $\ell = 1, \ldots, j-1$. Thus, we could have instead computed

$$r_{j,k} = \mathbf{q}_j^*(\mathbf{a}_k - r_{1,k}\mathbf{q}_1 - \cdots - r_{j-1,k}\mathbf{q}_{j-1}). \tag{2}$$

The formulation (2) might look like more work than (1) at first, but we can arrange the computations efficiently, according to the following Modified Gram–Schmidt algorithm.

```
for $j = 1, \ldots, n$
    $\widehat{\mathbf{q}}_j := \mathbf{a}_j$
end

for $j = 1, \ldots, n$
    $r_{j,j} := \|\widehat{\mathbf{q}}_j\|_2$
    $\mathbf{q}_j := \widehat{\mathbf{q}}_j / r_{j,j}$
    for $k = j+1, \ldots, n$
        $r_{j,k} := \mathbf{q}_j^* \widehat{\mathbf{q}}_k$
        $\widehat{\mathbf{q}}_k := \widehat{\mathbf{q}}_k - r_{j,k}\mathbf{q}_j$
    end
end
```

Why should this formulation be superior? Here is a heuristic explanation: We compute $\widehat{\mathbf{q}}_k$ by successively subtracting off terms like $r_{j,k}\mathbf{q}_j$. Small computer arithmetic errors will inevitably be made in the formation of $r_{j,k}$ and $\mathbf{q}_j$. Since the Modified Gram–Schmidt algorithm computes $r_{j,k}$ from the formula (2), the coefficient $r_{j,k}$ it computes will adjust to these errors, in some sense, attempting to 'project out the errors' at each step.

An illustrative example should clarify this situation.[†] Let

$$\mathbf{A} = \begin{bmatrix} 1 & 1 & 1 \\ \varepsilon & 0 & 0 \\ 0 & \varepsilon & 0 \\ 0 & 0 & \varepsilon \end{bmatrix} \in \mathbb{R}^{4\times 3},$$

where $\varepsilon \in \mathbb{R}$ is some number sufficiently small that computer arithmetic will round $1 + \varepsilon^2$ down to 1. (See Lecture 8 for further details on such arithmetic systems.) In MATLAB, $\varepsilon \le 10^{-8}$ will suffice.

We first apply the Classical Gram–Schmidt algorithm, the steps of which we summarize below.

$$\widehat{\mathbf{q}}_1 = \begin{bmatrix} 1 \\ \varepsilon \\ 0 \\ 0 \end{bmatrix}, \qquad r_{1,1} = 1 \text{ (rounded)}, \qquad \mathbf{q}_1 = \begin{bmatrix} 1 \\ \varepsilon \\ 0 \\ 0 \end{bmatrix}$$

[†]This is Problem/Solution 18.9 in N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, SIAM, Philadelphia, 1996. Higham attributes the example to Björck (1967).

$$r_{1,2} = \mathbf{q}_1^*\mathbf{a}_2 = 1, \qquad \widehat{\mathbf{q}}_2 = \begin{bmatrix}1\\0\\\varepsilon\\0\end{bmatrix} - 1\begin{bmatrix}1\\\varepsilon\\0\\0\end{bmatrix} = \begin{bmatrix}0\\-\varepsilon\\\varepsilon\\0\end{bmatrix}, \qquad r_{2,2} = \varepsilon\sqrt{2}$$

$$\mathbf{q}_2 = \begin{bmatrix}0\\-1/\sqrt{2}\\1/\sqrt{2}\\0\end{bmatrix}$$

$$r_{1,3} = \mathbf{q}_1^*\mathbf{a}_3 = 1, \quad r_{2,3} = \mathbf{q}_2^*\mathbf{a}_3 = 0, \quad \widehat{\mathbf{q}}_3 = \begin{bmatrix}1\\0\\0\\\varepsilon\end{bmatrix} - 1\begin{bmatrix}1\\\varepsilon\\0\\0\end{bmatrix} - 0\begin{bmatrix}1\\0\\\varepsilon\\0\end{bmatrix} = \begin{bmatrix}0\\-\varepsilon\\0\\\varepsilon\end{bmatrix}, \quad r_{3,3} = \varepsilon\sqrt{2}$$

$$\mathbf{q}_3 = \begin{bmatrix}0\\-1/\sqrt{2}\\0\\1/\sqrt{2}\end{bmatrix}.$$

But note that $\mathbf{q}_2$ and $\mathbf{q}_3$ are far from being orthogonal!

Now repeat the same calculation with Modified Gram–Schmidt. The first two steps, leading to $\mathbf{q}_1$ and $\mathbf{q}_2$, are identical, but now to form $\mathbf{q}_3$ we have the following operations:

$$r_{1,3} = \mathbf{q}_1^*\mathbf{a}_3 = 1, \qquad r_{2,3} = \mathbf{q}_2^*(\mathbf{a}_3 - r_{1,3}\mathbf{q}_1) = [0 \quad -1/\sqrt{2} \quad 1/\sqrt{2} \quad 0]\begin{bmatrix}0\\-\varepsilon\\0\\\varepsilon\end{bmatrix} = \varepsilon/\sqrt{2}.$$

The coefficient $r_{1,3}$ is the same as for Classical Gram–Schmidt, but now $r_{2,3} = \varepsilon/\sqrt{2}$ instead of $r_{2,3} = 0$. This modest change makes all the difference, for now we compute

$$\widehat{\mathbf{q}}_3 = (\mathbf{a}_3 - r_{1,3}\mathbf{q}_1) - r_{2,3}\mathbf{q}_2 = \begin{bmatrix}0\\-\varepsilon\\0\\\varepsilon\end{bmatrix} - \frac{\varepsilon}{\sqrt{2}}\begin{bmatrix}0\\-1/\sqrt{2}\\1/\sqrt{2}\\0\end{bmatrix} = \begin{bmatrix}0\\-\varepsilon/2\\-\varepsilon/2\\\varepsilon\end{bmatrix},$$

so $r_{3,3} = \sqrt{3/2}$ and hence

$$\mathbf{q}_3 = \begin{bmatrix}0\\-1/\sqrt{6}\\-1/\sqrt{6}\\\sqrt{2/3}\end{bmatrix}.$$

This vector is entirely different from vector $\mathbf{q}_3$ computed by the Classical Gram–Schmidt process.

Note that the $\mathbf{q}_1$, $\mathbf{q}_2$, and $\mathbf{q}_3$ vectors computed by the Modified Gram–Schmidt process are nearly orthogonal – the best we can hope for in computer arithmetic.

The following MATLAB code implements Modified Gram–Schmidt orthogonalization.

```
function [Q,R] = mgs_qr(A)
% Computation of the "skinny" QR decomposition of an m-by-n matrix
% (m>=n) using the the Modified Gram-Schmidt algorithm.
% See Trefethen and Bau, Algorithm 8.1

 [m,n] = size(A);
 if m<n, fprintf('ERROR: A should be an m-by-n matrix with m >= n.\n'); end
 Q = zeros(m,n);
 R = zeros(n,n);
 Q = A;
 for j=1:n
    R(j,j) = norm(Q(:,j));
    Q(:,j) = Q(:,j)/R(j,j);
    for k=j+1:n
        R(j,k) = Q(:,j)'*Q(:,k);
        Q(:,k) = Q(:,k) - R(j,k)*Q(:,j);
    end
 end
```

1.2.8. QR for rank-deficient matrices.

What if the columns of $\mathbf{A}$ are not linearly independent? In this case, the standard QR algorithm can fail. For example, the Gram–Schmidt orthogonalization process will eventually construct $\widehat{\mathbf{q}}_k = \mathbf{0}$ for some $k \le n$, giving an error for $\mathbf{q}_k = \widehat{\mathbf{q}}_k/\|\widehat{\mathbf{q}}_k\|_2$.

If one knows the rank of $\mathbf{A}$, say $\operatorname{rank}(\mathbf{A}) = \dim(\operatorname{Ran}\mathbf{A}) = r$, then one can swap the columns of $\mathbf{A}$ to allow a factorization of the form

$$\mathbf{Q}^*\mathbf{A}\mathbf{\Pi} = \begin{bmatrix} \mathbf{R}_{11} & \mathbf{R}_{12} \\ \mathbf{0} & \mathbf{0} \end{bmatrix},$$

where $\mathbf{\Pi} \in \mathbb{C}^{n\times n}$ is a permutation matrix[‡] that affects the column swapping and $\mathbf{R}_{11} \in \mathbb{C}^{r\times r}$ is an upper triangular matrix with all diagonal entries nonzero. For details and an explanation of how to implement such a factorization using Householder reflectors, see Section 5.4.1 of G. H. Golub and C. F. Van Loan, *Matrix Computations*, 3rd ed., Johns Hopkins, Baltimore, 1996. In MATLAB, one can obtain a column-pivoted QR factorization by calling the `qr` routine as

```
[Q,R,Pi] = qr(A);
```

which computes the factorization $\mathbf{A}\mathbf{\Pi} = \mathbf{Q}\mathbf{R}$.

[‡]A permutation matrix is a square matrix in which each row and each column has exactly one entry equal to one, with all other entries equal to zero.

Lecture 6: Using a QR Decomposition to Solve Linear Systems

1.3 Solving linear systems of equations using the QR decomposition.

Suppose $\mathbf{A} \in \mathbb{C}^{n\times n}$ is a nonsingular (i.e., invertible) matrix, and we wish to solve the system of linear equations $\mathbf{A}\mathbf{x} = \mathbf{b}$ for the unknown $\mathbf{x} \in \mathbb{C}^n$. The QR decomposition, $\mathbf{A} = \mathbf{Q}\mathbf{R}$, allows us to transform this general system into a simpler problem:

$$\mathbf{A}\mathbf{x} = \mathbf{b} \qquad \Longleftrightarrow \qquad \mathbf{Q}\mathbf{R}\mathbf{x} = \mathbf{b} \qquad \Longleftrightarrow \qquad \mathbf{R}\mathbf{x} = \mathbf{Q}^*\mathbf{b}.$$

This final system involves the upper triangular matrix $\mathbf{R}$, and thus can be quickly solved with *backward substitution.*

1.3.1 Solving upper triangular linear systems.

Now we address the simple problem of solving a linear system of the form $\mathbf{R}\mathbf{x} = \mathbf{c}$, where $\mathbf{R} \in \mathbb{C}^{n\times n}$ is upper triangular, i.e., $r_{jk} = 0$ if $j > k$. (Of course, $\mathbf{c}$ here would be replaced by $\mathbf{Q}^*\mathbf{b}$ in the context of the original linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$.) This equation takes the form

$$\begin{bmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ & r_{22} & \cdots & r_{2n} \\ & & \ddots & \vdots \\ & & & r_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix}. \tag{1}$$

The last row of this vector equation is scalar equation

$$r_{nn}x_n = c_n,$$

from which we can immediately find x_n:

$$x_n = c_n/r_{nn}.$$

The penultimate row of (1) gives

$$r_{n-1,n-1}x_{n-1} + r_{n-1,n}x_n = c_{n-1}.$$

Since we have already computed x_n, the only remaining unknown is x_{n-1}, for which we find

$$x_{n-1} = \frac{1}{r_{n-1,n-1}}(c_{n-1} - r_{n-1,n}x_n).$$

We find $x_{n-2}, \ldots, x_1$ similarly:

$$x_j = \frac{1}{r_{jj}}\Big(c_j - \sum_{k=j+1}^{n} r_{j,k}x_k\Big).$$

Since we work from the bottom of (1) up, this procedure is called *back substitution.* Clearly it is much simpler than applying Gaussian elimination to a general (non-triangular) linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$: all the hard work was accomplished when we computed the QR factorization.

The following MATLAB code gives an efficient implementation of back substitution, written in a concise form. Can you follow it?

```
  function x = backsolve(R,c)
% Solves R*x = c for x, where R is a nonsingular (square) upper triangular matrix.
  [n n] = size(R);
  x = zeros(n,1);
  for j=n:-1:1
     x(j) = (c(j) - R(j,j+1:n)*x(j+1:n))/R(j,j);
  end
```

In the first pass through the `for` loop, we have `j=n`, so the range `j+1:n` is empty. Fear not; MATLAB is smart enough to handle this: `R(j,j+1:n)` and `x(j+1:n))` are 1-by-0 and 0-by-1 empty vectors, and MATLAB has the convention that the product of such empty vectors is zero.

1.3.2 Overall complexity of solving linear systems.

What is the total computational cost of solving $\mathbf{A}\mathbf{x} = \mathbf{b}$ using the QR decomposition, followed by back substitution?

In the last lecture we determined the that roughly

$$2mn^2 - \tfrac{2}{3}n^3$$

floating point operations are required to compute a QR factorization with Householder reflectors. When $\mathbf{A}$ is square, i.e., $m = n$, this simplifies to $\frac{4}{3}n^3$ operations.

To the cost of the factorization, one needs to add the expense of forming $\mathbf{c} = \mathbf{Q}^*\mathbf{b}$ and performing back substitution. Can you work out the number of floating point operations required for these operations from the MATLAB algorithms given in this lecture and the last?

In many applications, one is presented with a sequence of linear systems of the form

$$\mathbf{A}\mathbf{x}_k = \mathbf{b}_k, \qquad k = 1, 2, \ldots,$$

where $\mathbf{b}_k$ might depend on $\mathbf{x}_{k-1}$, the solution to the previous system. Once we have computed a QR factorization for $\mathbf{A}$, how expensive is it to solve p linear systems of the form $\mathbf{A}\mathbf{x}_k = \mathbf{b}_k$?

Many of you know that one can solve linear systems in MATLAB with the *backslash command*: `x=A\b` solves $\mathbf{A}\mathbf{x} = \mathbf{b}$. Before solving this system, MATLAB automatically checks if the matrix is upper triangular or has other special structure to exploit. (You can improve performance by explicitly telling MATLAB that your coefficient matrix has special structure with the `linsolve` command.) The following codes each solve 100 systems of the form $\mathbf{A}\mathbf{x}_k = \mathbf{b}_k$ with $\mathbf{A}$ a random 500×500 matrix. Do you see why the code on the left runs about ten times faster on my laptop?

```
n = 500; p = 100;
A = randn(n); B = randn(n,p);
tic
[Q,R] = qr(A);
for k=1:p
   b = B(:,k);
   x = R\(Q'*b);
end
fprintf('elapsed time = %10.7f\n', toc)
```

```
n = 500; p = 100;
A = randn(n); B = randn(n,p);
tic
for k=1:p
   b = B(:,k);
   x = A\b;
end
fprintf('elapsed time = %10.7f\n', toc)
```

Lecture 7: Conditioning and Stability

1.4 Conditioning and Stability.

One can solve linear systems in a finite number of algebraic operations: Given the entries of $\mathbf{A}$ and $\mathbf{b}$, with sufficient stamina we could compute – by hand, even – the exact answer $\mathbf{x}$ to the system $\mathbf{Ax} = \mathbf{b}$. One does this in elementary linear algebra class with matrices of dimension $n = 3$ or perhaps, if the instructor has a sadistic streak, $n = 4$ or 5. With considerable effort and sufficient motivation (historical episodes involving architecture and warfare come to mind) one could solve dense systems with, say, $n = 10$ or a bit more.

Computers make quick work of systems of such small size. While one could, in principle, work in exact arithmetic, this approach is rather slow for the large systems that arise from practical applications. Instead, MATLAB (and similar systems) perform *floating point arithmetic*, rounding the entries in $\mathbf{A}$ and $\mathbf{b}$ to nearest values in a finite number system, then performing approximate arithmetic in that system. The implementation in general use today, *IEEE double precision arithmetic*, provides roughly sixteen digits of relative accuracy when it stores and operates on numbers. We will explore the particular features of such arithmetic systems in the next lecture. As present, we are more concerned with how small changes in $\mathbf{A}$ and $\mathbf{b}$ will affect the solution $\mathbf{x}$.

Since the entries of $\mathbf{A}$ and $\mathbf{b}$ are only stored approximately, we cannot expect the 'solution' $\mathbf{x}$ that we subsequently compute to be exact. At best, $\mathbf{x}$ will be the *the exact solution to a 'nearby' linear system.* We need to understand how large the discrepancy (or *residual*) $\mathbf{Ax} - \mathbf{b}$ can be. The issue breaks into two subordinate questions. First, how do small changes in $\mathbf{A}$ and $\mathbf{b}$ affect the solution $\mathbf{x}$? Next, how does the algorithm that computed $\mathbf{x}$ perform if the arithmetic operations involved are not exact, but only accurate to, say, fifteen digits? The first of these questions concerns the *conditioning* of the mathematical problem; the second concerns the *stability* of the numerical algorithm that we used to compute 'the solution'. The distinction between conditioning and stability is a fundamental concept in the design and analysis of numerical algorithms.

1.4.1 Abstract condition numbers.

First we address the conditioning of the linear system $\mathbf{Ax} = \mathbf{b}$. The *condition number* of a problem can be abstractly defined as follows.[†] Suppose we wish to compute some (vector-valued) function $\mathbf{f}(\mathbf{y})$. How does the value of $\mathbf{f}$ change when $\mathbf{y}$ is subjected to a perturbation $\boldsymbol{\delta}\mathbf{y}$? Define

$$\boldsymbol{\delta}\mathbf{f} := \mathbf{f}(\mathbf{y} + \boldsymbol{\delta}\mathbf{y}) - \mathbf{f}(\mathbf{y}).$$

(Note that '$\boldsymbol{\delta}\mathbf{y}$' and '$\boldsymbol{\delta}\mathbf{f}$' are both names of vectors; $\boldsymbol{\delta}\mathbf{y}$ *does not* mean 'the scalar δ times the vector $\mathbf{y}$'; thus, e.g., you cannot peel the '$\boldsymbol{\delta}$' away from the '$\mathbf{y}$' in $\boldsymbol{\delta}\mathbf{y}$. This usage departs from our notational convention, but is standard for this style of analysis.) We need to compare the size of $\boldsymbol{\delta}\mathbf{f}$ to that of $\boldsymbol{\delta}\mathbf{y}$. Of course, the magnitudes of $\boldsymbol{\delta}\mathbf{y}$ and $\boldsymbol{\delta}\mathbf{f}$ depend on the size of $\mathbf{y}$ and $\mathbf{f}(\mathbf{y})$, and hence we are interested in the *relative* size of these perturbations:

$$\frac{\|\boldsymbol{\delta}\mathbf{f}\|/\|\mathbf{f}(\mathbf{y})\|}{\|\boldsymbol{\delta}\mathbf{y}\|/\|\mathbf{y}\|}.$$

Can the numerator be large when the denominator is small? That is, can small relative changes in $\mathbf{y}$ induce large shifts in the solution? Here $\boldsymbol{\delta}\mathbf{y}$ is a vector, and one expects that some choices for

[†]See Lecture 12 of Trefethen and Bau, which provides the background for the present notes.

that vector in $\mathbb{C}^n$ might stimulate more significant errors than others. To characterize the worst case, maximize over all $\boldsymbol{\delta}\mathbf{y}$ of some fixed size, and continue what happens as the size of that error, $\|\boldsymbol{\delta}\mathbf{y}\|$, goes to zero. The *condition number* of $\mathbf{f}(\mathbf{y})$ is thus defined as

$$\kappa \;=\; \lim_{\Delta\to 0}\;\max_{\|\boldsymbol{\delta}\mathbf{y}\|\le\Delta}\;\frac{\|\boldsymbol{\delta}\mathbf{f}\|/\|\mathbf{f}(\mathbf{y})\|}{\|\boldsymbol{\delta}\mathbf{y}\|/\|\mathbf{y}\|}.$$

1.4.2 Condition number for solving linear systems.

We wish to apply these ideas to analyze the conditioning of the linear system $\mathbf{A}\mathbf{x}=\mathbf{b}$. Here the data '$\mathbf{y}$' comprises $\mathbf{A}$ and $\mathbf{b}$, and '$\mathbf{f}(\mathbf{y})$' is $\mathbf{A}^{-1}\mathbf{b}$. Rather than attempting to plug these values into the above formula, we perform a more direct analysis.

Suppose $\mathbf{A}\mathbf{x}=\mathbf{b}$ for nonzero $\mathbf{x}$ and $\mathbf{b}$, and $\mathbf{A}$ is invertible. First, introduce an *infinitesimal* perturbation $\boldsymbol{\delta}\mathbf{A}$ to $\mathbf{A}$ that changes the solution by $\boldsymbol{\delta}\mathbf{x}$:

$$(\mathbf{A}+\boldsymbol{\delta}\mathbf{A})(\mathbf{x}+\boldsymbol{\delta}\mathbf{x})=\mathbf{b}.$$

Multiplying this out gives $\mathbf{A}\mathbf{x}+\mathbf{A}(\boldsymbol{\delta}\mathbf{x})+(\boldsymbol{\delta}\mathbf{A})\mathbf{x}+(\boldsymbol{\delta}\mathbf{A})(\boldsymbol{\delta}\mathbf{x})=\mathbf{b}$. Since $\boldsymbol{\delta}\mathbf{A}$ and $\boldsymbol{\delta}\mathbf{x}$ are infinitesimal quantities, we can ignore the quadratic term,

$$\mathbf{A}\mathbf{x}+\mathbf{A}(\boldsymbol{\delta}\mathbf{x})+(\boldsymbol{\delta}\mathbf{A})\mathbf{x}=\mathbf{b}.$$

Since $\mathbf{A}\mathbf{x}=\mathbf{b}$, we can subtract the first terms on each side and rearrange to obtain

$$\mathbf{A}(\boldsymbol{\delta}\mathbf{x})=-(\boldsymbol{\delta}\mathbf{A})\mathbf{x}.$$

Inverting $\mathbf{A}$ and taking norms leads to

$$\|\boldsymbol{\delta}\mathbf{x}\| \;=\; \|\mathbf{A}^{-1}(\boldsymbol{\delta}\mathbf{A})\mathbf{x}\| \;\le\; \|\mathbf{A}^{-1}\|\,\|\boldsymbol{\delta}\mathbf{A}\|\,\|\mathbf{x}\|.$$

As we seek to compare the *relative* change in the solution with the change in the $\mathbf{A}$, we want a $\|\boldsymbol{\delta}\mathbf{A}\|/\|\mathbf{A}\|$ term on the right. Multiplying the right hand side by $1=\|\mathbf{A}\|/\|\mathbf{A}\|$ and dividing by $\|\mathbf{x}\|$ yields

$$\frac{\|\boldsymbol{\delta}\mathbf{x}\|}{\|\mathbf{x}\|} \;\le\; \|\mathbf{A}\|\,\|\mathbf{A}^{-1}\|\frac{\|\boldsymbol{\delta}\mathbf{A}\|}{\|\mathbf{A}\|}.$$

It follows that infinitesimal changes in $\mathbf{A}$ can be magnified by no more than

$$\kappa(\mathbf{A}) \;:=\; \|\mathbf{A}\|\,\|\mathbf{A}^{-1}\|,$$

which is called the *condition number of* $\mathbf{A}$ *with respect to solving linear systems.* This is what most people mean when they casually speak of 'the condition number of $\mathbf{A}$'.

It is even simpler to consider how an infinitesimal change $\boldsymbol{\delta}\mathbf{b}$ to $\mathbf{b}$ can affect $\mathbf{A}\mathbf{x}=\mathbf{b}$. Now we have

$$\mathbf{A}(\mathbf{x}+\boldsymbol{\delta}\mathbf{x})=\mathbf{b}+\boldsymbol{\delta}\mathbf{b},$$

which, after canceling $\mathbf{A}\mathbf{x}$ and $\mathbf{b}$ from either side, reduces to

$$\boldsymbol{\delta}\mathbf{x}=\mathbf{A}^{-1}(\boldsymbol{\delta}\mathbf{b}).$$

Taking norms and multiplying by $1 = \|\mathbf{Ax}\|/\|\mathbf{b}\|$, we have

$$\|\boldsymbol{\delta}\mathbf{x}\| = \|\mathbf{A}^{-1}(\boldsymbol{\delta}\mathbf{b})\|\frac{\|\mathbf{Ax}\|}{\|\mathbf{b}\|} \;\le\; \|\mathbf{A}\|\|\mathbf{A}^{-1}\|\frac{\|\boldsymbol{\delta}\mathbf{b}\|}{\|\mathbf{b}\|}\|\mathbf{x}\|,$$

which implies

$$\frac{\|\boldsymbol{\delta}\mathbf{x}\|}{\|\mathbf{x}\|} \;\le\; \|\mathbf{A}\|\,\|\mathbf{A}^{-1}\|\frac{\|\boldsymbol{\delta}\mathbf{b}\|}{\|\mathbf{b}\|}.$$

Again, the condition number $\kappa(\mathbf{A})$ is the factor that magnifies the perturbation in the data.

It is a useful exercise to piece together the two previous arguments to handle perturbations to both $\mathbf{A}$ and $\mathbf{b}$ simultaneously:

$$(\mathbf{A}+\boldsymbol{\delta}\mathbf{A})(\mathbf{x}+\boldsymbol{\delta}\mathbf{x}) = \mathbf{b}+\boldsymbol{\delta}\mathbf{b}.$$

How should we interpret the condition number? It measures the distance of $\mathbf{A}$ from singularity. In particular, one can show (in any induced norm) that there exists a perturbation $\mathbf{E}$ with $\|\mathbf{E}\| = 1/\|\mathbf{A}^{-1}\|$ such that $\mathbf{A}+\mathbf{E}$ is singular (i.e., $\mathbf{A}+\mathbf{E}$ has a zero eigenvalue), and this is the smallest perturbation that makes $\mathbf{A}$ singular. Thus, the *relative* size of the smallest perturbation is $1/\kappa(\mathbf{A})$.

Is there any connection between $\kappa(\mathbf{A})$ and the determinant, $\det(\mathbf{A})$?

Lecture 8: Floating Point Number Systems

1.5 Floating point arithmetic.

To this point we have mainly considered pure algorithms, with only tangential concern for how these algorithms behave when executed on a computer. Yet in our comparison of two implementations of the skinny QR factorization (via classical and modified Gram–Schmidt orthogonalization), we observed that mathematically equivalent methods can yield significantly different results when implemented on a computer. Now we pause to consider the computer arithmetic that gives rise to such behavior.

First, it is important to note two important styles of computation.

- *Symbolic computing*, practiced by Mathematica and Maple, applies rules of algebra and calculus to symbolic expressions that represent variables. When numerical values are used, they are stored exactly as rational numbers, radicals, and special constants such as π. As these quantities are manipulated and combined, the size of the resulting numerators and denominators can grow rapidly: this increases memory requirements and degrades performance. Such calculations can be slow and produce results that are difficult to simplify, but they are exact.
- *Numerical computing*, practiced by MATLAB, converts all input numbers into a *floating point number system* internal to the computer. Because the floating point system contains only finitely many numbers, most input quantities must be rounded to the nearest represented number. This system is not *closed* under basic arithmetic operations, e.g., the sum of two floating point numbers need not be a floating point number. Thus, the intermediate quantities computed by algorithms are also rounded to the nearest floating point number. Floating point computations, typically performed in hardware, are typically much more efficient than symbolic ones. The key question is, how does the accumulation of rounding errors pollute the final answer an algorithm produces?

The above software descriptions are oversimplified. For example, Mathematica supports numerical computing (to a user-specified precision), while MATLAB's Symbolic Toolbox provides some symbolic operations and an interface to the Maple symbolic computing system. A notable third style of computing, *interval arithmetic*, resembles the numerical approach, but now upper and lower bounds on all rounded quantities are stored. Thus, interval algorithms can guarantee a bound on the desired solution, while still enjoying the many benefits of numerical computing. The primary challenge comes in designing algorithms that yield small intervals.

While symbolic computing and interval arithmetic have their places, the great bulk of engineering computations are performed in floating point arithmetic, and that shall be our focus.

A floating point number system is described in terms of four parameters:

β,	the base;
t,	the precision;
$e_{\min}$,	the minimum exponent;
$e_{\max}$,	the maximum exponent.

With these parameters, the floating point system is defined as follows (m and e must be integers):

$$\mathbb{F} = \{\pm m\beta^{e-t} : \text{either } \beta^{t-1} \le m < \beta^t \text{ and } e_{\min} \le e \le e_{\max}, \text{ or } 0 \le m < \beta^{t-1} \text{ and } e = e_{\min}\}.$$

The numbers $\pm m\beta^{e-t}$ for $\beta^{t-1} \le m < \beta^t$ and $e_{\min} \le e \le e_{\max}$ (and zero, too) are called *normal numbers*, for a reason that shall become apparent when we see how these numbers are stored on a computer. The numbers $\pm m\beta^{e-t}$ for $0 \le m < \beta^{t-1}$ and $e = e_{\min}$ are *subnormal numbers.*

All modern computers use the base $\beta = 2$ (i.e., binary numbers), where each binary digit (*bit*) takes a value 0 or 1. though some calculators reportedly use $\beta = 10$. The parameter t affects how densely packed the floating point numbers are, while $e_{\min}$ and $e_{\max}$ control the largest and smallest representable numbers.

Example. It may be difficult to visualize this definition of $\mathbb{F}$, but working out a small example will flesh these numbers out.[†] Take $\beta = 2$, $t = 3$, $e_{\min} = -1$, $e_{\max} = 2$. Recall the definition

$$\mathbb{F} = \{\pm m\beta^{e-t} : \text{either } \beta^{t-1} \le m < \beta^t \text{ and } e_{\min} \le e \le e_{\max}, \text{ or } 0 \le m < \beta^{t-1} \text{ and } e = e_{\min}\}.$$

For simplicity, we will only list the nonnegative floating point numbers. First, we will enumerate all the normal numbers by stepping e from $e_{\min}$ to $e_{\max}$. For these fixed values of e, m can take any value between $\beta^{t-1} = 2^{3-1} = 4$ and $\beta^t - 1 = 2^3 - 1 = 7$.

$\boxed{e = -1,\ \beta^{e-t} = 2^{-4} = 1/16}$

m	$m\beta^{e-t}$
4	$4/16 = 1/4$
5	$5/16$
6	$6/16 = 3/8$
7	$7/16$

$\boxed{e = 0,\ \beta^{e-t} = 2^{-3} = 1/8}$

m	$m\beta^{e-t}$
4	$4/8 = 1/2$
5	$5/8$
6	$6/8 = 3/4$
7	$7/8$

$\boxed{e = 1,\ \beta^{e-t} = 2^{-2} = 1/4}$

m	$m\beta^{e-t}$
4	$4/4 = 1$
5	$5/4$
6	$6/4 = 3/2$
7	$7/4$

$\boxed{e = 2,\ \beta^{e-t} = 2^{-1} = 1/2}$

m	$m\beta^{e-t}$
4	$4/2 = 2$
5	$5/2$
6	$6/2 = 3$
7	$7/2$

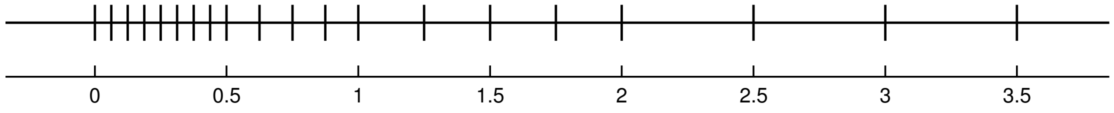
Note that for each fixed value of e, the gap between successive numbers is always the same: β^{e-t}. Finally, we enumerate zero and the subnormal numbers, for which the gap between successive numbers is the same as for the smallest normal numbers, $\beta^{e_{\min}-t}$.

$\boxed{e = e_{\min} = -1,\ \beta^{e-t} = 2^{-4} = 1/16}$

m	$m\beta^{e-t}$
0	$0/16 = 0$
1	$1/16$
2	$2/16 = 1/8$
3	$3/16$

[†]We draw this example from Nicholas J. Higham's outstanding treatise, *Accuracy and Stability of Numerical Algorithms*, 2nd ed., SIAM, Philadelphia, 2002.

It helps to see where these quantities fall on the number line.



As the numbers get larger, the spacing between them increases. (This is a hallmark of floating point numbers, as opposed to the *fixed point* number systems that were seen as viable alternatives in the 1950s.) There is a convenient way to describe the precision of a floating point number system.

Definition. The *machine epsilon*, written $\varepsilon_{\text{mach}}$ or $\varepsilon_{\text{machine}}$, is the gap between 1 and the next largest floating point number, i.e., $\varepsilon_{\text{mach}} = \beta^{1-t}$.

Suppose that $x \in \mathbb{R}$ is a real number within the limits of the normalized floating point numbers, and let $\mathrm{f}\ell(x) \in \mathbb{F}$ denote the floating point approximation of x. Then we can be sure that

$$\mathrm{f}\ell(x) = x(1+\delta), \qquad |\delta| \le \varepsilon_{\text{mach}}, \tag{8.1}$$

i.e., the *relative error* is controlled by $\varepsilon_{\text{mach}}$:

$$\frac{|\mathrm{f}\ell(x) - x|}{|x|} \le \varepsilon_{\text{mach}}.$$

In the floating point system with $t = 3$ that we worked out above, $\varepsilon_{\text{mach}} = \beta^{1-t} = 1/4$. Returning to our earlier example, suppose $x = 2.25$. This number is not in the floating point system, so it is rounded to the nearest number in $\mathbb{F}$, e.g., $\mathrm{f}\ell(2.25) = 2.5$, giving

$$\frac{|\mathrm{f}\ell(2.25) - 2.25|}{2.25} = \frac{.25}{2.25} = 1/9,$$

which indeed is less than $\varepsilon_{\text{mach}} = 1/4$. (Actually, the key property (8.1) would also hold if we define $\varepsilon_{\text{mach}}$ to instead be $\frac{1}{2}\beta^{1-t}$, which is preferred by some authors. In this case, $1+\varepsilon_{\text{mach}}$ would be the first real number that could be rounded to a floating point number larger than 1.)

1.5.1 Accuracy of simple floating point computations.

The property (8.1) describes the error made when a real number is stored in a floating point system, but in order to analyze algorithms, we require more information. As mentioned above, the floating point system is not *closed* under basic arithmetic operations. For example, the numbers 7/8 and 1 are both in our toy system detailed above, but their sum, $7/8 + 1 = 15/8$ is not. How should the computer handle the addition of such numbers? Computer engineers design floating point arithmetic units to ensure that several basic axioms analogous to the property (8.1) hold. Suppose that x, y are both normalized floating point numbers. Then provided the result of the exact arithmetic operation is also within the range of the normalized floating point numbers, we have

$$\begin{aligned}
\mathrm{f}\ell(x+y) &= (x+y)(1+\delta), \qquad |\delta| \le \varepsilon_{\text{mach}},\\
\mathrm{f}\ell(x-y) &= (x-y)(1+\delta), \qquad |\delta| \le \varepsilon_{\text{mach}},\\
\mathrm{f}\ell(x\times y) &= (x\times y)(1+\delta), \qquad |\delta| \le \varepsilon_{\text{mach}},\\
\mathrm{f}\ell(x\div y) &= (x\div y)(1+\delta), \qquad |\delta| \le \varepsilon_{\text{mach}}.
\end{aligned}$$

In other words, basic arithmetic operations are implemented so that there is a small *relative* error. This may seem to be just about perfect, but perplexing results can still occur. Note that the above axioms required that $x, y \in \mathbb{F}$. If we wish, for example, to subtract two *real* numbers, we must first convert each number to something in $\mathbb{F}$, then add those two floating point numbers, resulting in a total of three rounding errors. More precisely, suppose $x, y \in \mathbb{R}$, and define

$$\begin{aligned}\widehat{x} &= \mathrm{f}\ell(x) = x(1+\delta_1)\\ \widehat{y} &= \mathrm{f}\ell(y) = y(1+\delta_2),\end{aligned}$$

with $|\delta_1|, |\delta_2| \le \varepsilon$. Now subtract these numbers:

$$\begin{aligned}\mathrm{f}\ell(\widehat{x}-\widehat{y}) &= (\widehat{x}-\widehat{y})(1+\delta_3)\\ &= (x-y+\delta_1 x-\delta_2 y)(1+\delta_3).\end{aligned}$$

Suppose that we have the good fortune that $\delta_3 = 0$, i.e., the subtraction is performed exactly. Then

$$\frac{|\mathrm{f}\ell(\widehat{x}-\widehat{y})-(x-y)|}{|x-y|} \le \frac{|\delta_1 x - \delta_2 y|}{|x-y|} \le \max\{|\delta_1|, |\delta_2|\}\frac{|x|+|y|}{|x-y|}.$$

This upper bound suggests that when $x \approx y$, there can be a very large relative error. Suppose we wish to subtract $y = 2$ from $x = 2.25$ in the demonstration system above. Then $\widehat{x} = \mathrm{f}\ell(x) = 2.5$, rounding up, and $\widehat{y} = \mathrm{f}\ell(y) = 2$ (no rounding). We can then compute the relative error in subtraction (note that $\widehat{x} - \widehat{y}$ is exactly represented in this floating point system):

$$\frac{|\mathrm{f}\ell(\widehat{x}-\widehat{y})-(x-y)|}{|x-y|} = \frac{|.5-.25|}{|.25|} = 1.$$

We were hoping for a relative error like $\varepsilon_{\text{mach}}$, but instead we have a much larger error; no digits in the computed solution are correct. This phenomenon is known as *catastrophic cancellation*: the difference of two nearby floating point numbers may have very few accurate digits, and this error can go on to spoil subsequent computations.

1.4.3 Binary representation of floating point numbers.

In the past, different computers used different floating point systems, meaning that a program need not produce the same answer when executed on various computers. In the 1980s there was a significant move toward standardization, and now almost all modern computers use the IEEE floating point standard, which specifies formats for 'single' and 'double' precision systems, along with several special numbers (Not a Number, Infinity, etc.).

	β	t	$e_{\min}$	$e_{\max}$	$\varepsilon_{\text{mach}}$	range	storage
IEEE single precision	2	24	-125	128	6×10^{-8}	$10^{\pm 38}$	32 bits
IEEE double precision	2	53	-1021	1024	1×10^{-16}	$10^{\pm 308}$	64 bits

By default, MATLAB uses IEEE double precision arithmetic. You can view $\varepsilon_{\text{mach}}$ by typing the command `eps`. Type `help eps` to find other interesting properties of this command; see also `realmin`, and `realmax`. Recent versions of MATLAB also allow you to experiment with single precision arithmetic; type `help single` for details.

An excellent resource to learn about the IEEE floating point system is: Michael L. Overton, *Numerical Computing with IEEE Floating Point Arithmetic*, SIAM, Philadelphia, 2001.

Lecture 9: Polynomial Interpolation

2. Polynomial Interpolation.

As an application of solving linear systems, we turn to the problem of modeling a continuous real function $f : [a, b] \to \mathbb{R}$ by a polynomial. 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 $[a, b] \subset \mathbb{R}$.

2.1. Basic definitions and notation.

Definition. 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. 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

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

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 $f \in \mathbb{C}[a, b]$ 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)?
- 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.

2.2. 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; such 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

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

Matrices of this form, called *Vandermonde* matrices, arise in a wide range of applications.[†] Provided all the interpolation points $\{x_j\}$ are distinct, one can show that this matrix is invertible. (In fact, the determinant takes the simple form

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

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

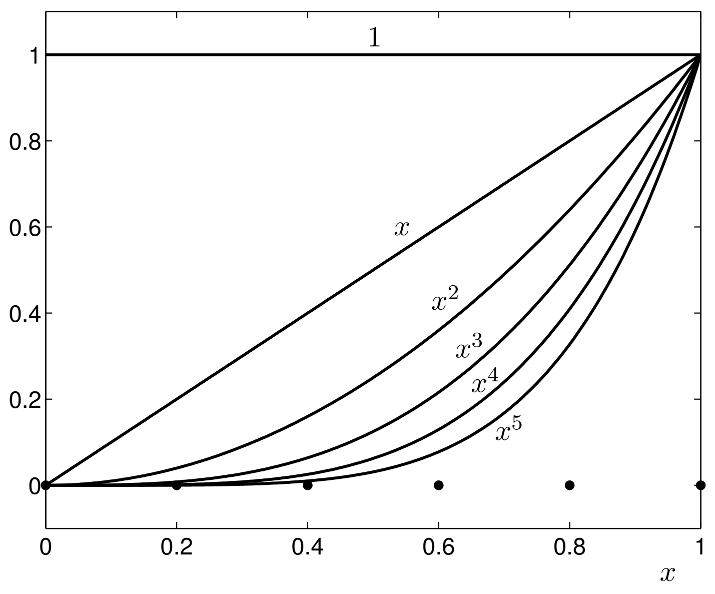
To determine the coefficients $\{c_j\}$, we could solve the above linear system using the QR decomposition of the Vandermonde matrix, as described in previous lectures. Alternatively, we could use Gaussian elimination, which we will discuss in future lectures. (This is what MATLAB's \ command uses.) There exists a third alternative, specialized algorithms that exploit the Vandermonde

[†]Higham presents many interesting properties of Vandermonde matrices and related computations in Chapter 21 of *Accuracy and Stability of Numerical Algorithms* (SIAM, Philadelphia, 1996).

structure to determine the coefficients $\{c_j\}$ in $O(n^2)$ operations, a vast improvement over the $O(n^3)$ operations required by Gaussian elimination or QR decomposition.[‡]

2.2.1. 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. The following plot illustrates this on the interval $[a, b] = [0, 1]$ with $n = 5$ and $x_j = j/5$.



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 strictly speaking they are linearly independent. We can write the vector $(1, 1)^T$ as a unique linear combination of these basis vectors,

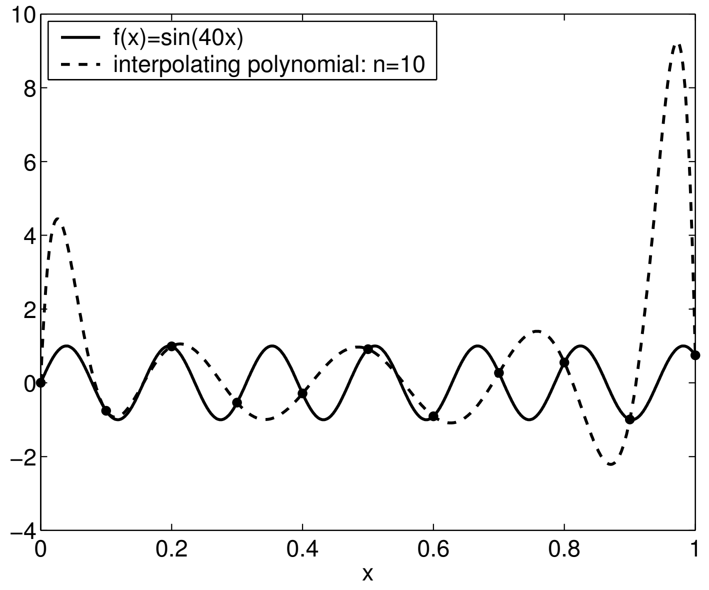
$$\begin{bmatrix} 1 \\ 1 \end{bmatrix} = 10,000,000,000 \begin{bmatrix} 1 \\ 10^{-10} \end{bmatrix} - 9,999,999,999 \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

Although the vector we are expanding and the basis vectors themselves are all small in norm, the expansion coefficients are enormous. Furthermore, small changes to the vector we are expanding

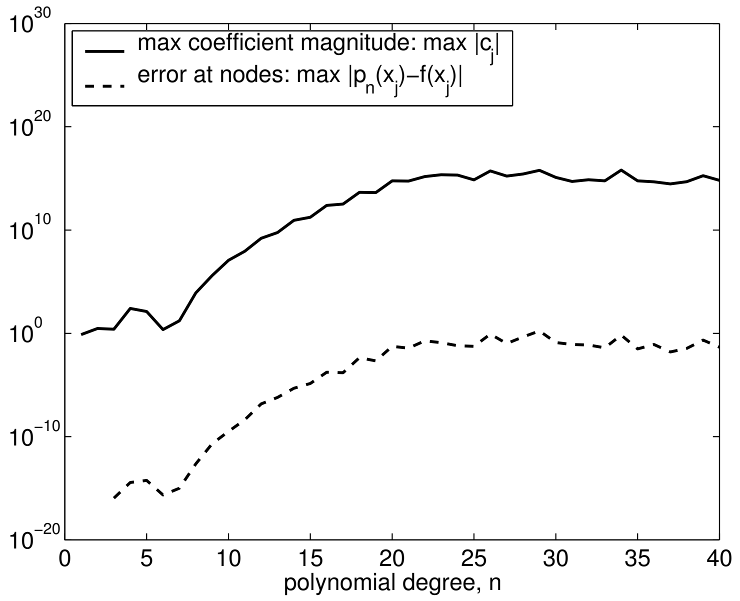
[‡]See Higham's book for details and stability analysis of specialized Vandermonde algorithms.

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 $f(x) = \sin(40x)$ at uniformly spaced x_j in the interval $[0, 1]$. Note that $f \in C^\infty[0, 1]$, and $|f(x)| \in [0, 1]$. As seen in the following plot, f oscillates modestly on the interval $[0, 1]$, but it certainly does not grow excessively large in magnitude or exhibit any nasty singularities.



It turns out that as $n \to \infty$, the interpolating polynomial converges to the true function in a manner we shall make precise in coming lectures. However, our MATLAB computation that solves the Vandermonde system (using \) and then evaluates the polynomial (using `polyval`) is remarkably inaccurate due to the magnitude of the expansion coefficients. The plot below shows how the computed coefficients grow to roughly 10^{15} in magnitude as n increases to 40. We compare this error to the quantity $\max_{0 \le j \le n} |f(x_j) - p_n(x_j)|$. In theory, this quantity should be zero, since p_n interpolates f at the points $\{x_j\}$, but in practice, numerical errors pollute the evaluation of p_n.



We are prepared to accept errors of roughly 10^{-15} in size, due to the finite accuracy of the computer's floating point arithmetic. However, we see errors more like 10^{-1}: 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. In the next lecture, we shall obtain improved results by expanding the interpolating polynomial in a basis that is better conditioned.

Lecture 10: Superior Bases for Polynomial Interpolants

2.3. Constructing interpolants in the Newton 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: it is typically best to work in a well-conditioned 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 form

$$p_0(x) = c_0$$

for all x. To satisfy the interpolation condition at x_0, set $c_0 = f(x_0)$. (Note that this c_0, and the c_j below, will be different from the c_j's obtained in the previous section with 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 will work for arbitrary 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

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

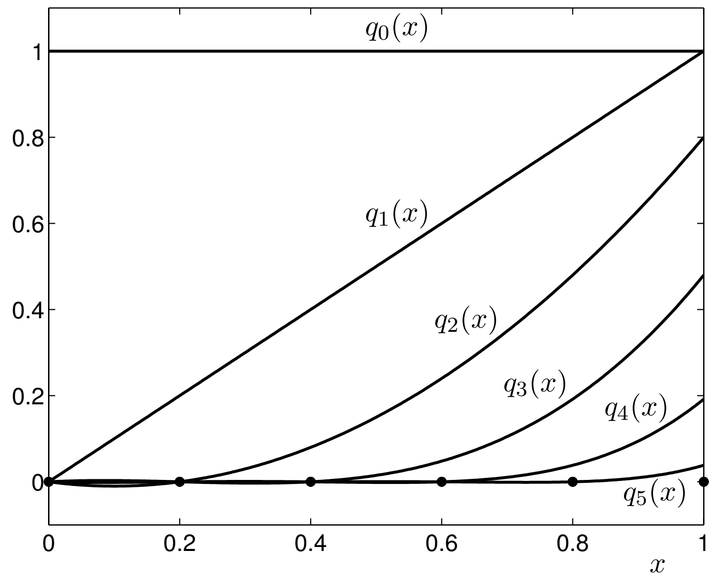
and, setting $q_0(x) = 1$, we have

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

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. The plot below 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 vectors illustrated in the last lecture.



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:

$$\begin{bmatrix} 1 & & & & \\ 1 & (x_1-x_0) & & & \\ 1 & (x_2-x_0) & (x_2-x_0)(x_2-x_1) & & \\ \vdots & \vdots & \vdots & \ddots & \\ 1 & (x_n-x_0) & (x_n-x_0)(x_n-x_1) & \cdots & \prod_{j=0}^{n-1}(x_n-x_j) \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} f(x_0) \\ f(x_1) \\ f(x_2) \\ \vdots \\ f(x_n) \end{bmatrix}. \qquad (10.1)$$

(The unspecified entries above the diagonal are zero.) This system involves a triangular matrix, which is simple to solve. Clearly $c_0 = f(x_0)$, and once we know c_0, we can solve $c_1 = (f(x_1) - c_0)/(x_1 - x_0)$. 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 (10.1), 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.

2.4. Constructing interpolants in the Lagrange basis.

A third alternative, called the *Lagrange form* of the interpolating polynomial, expresses p_n using more elaborate basis functions, but simpler constants $\{c_j\}$.

The basic idea is to express p_n as the linear combination of basis functions $\ell_j \in \mathcal{P}_n$, where $\ell_j(x_k) = 0$ 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, $c_j = f(x_j)$.

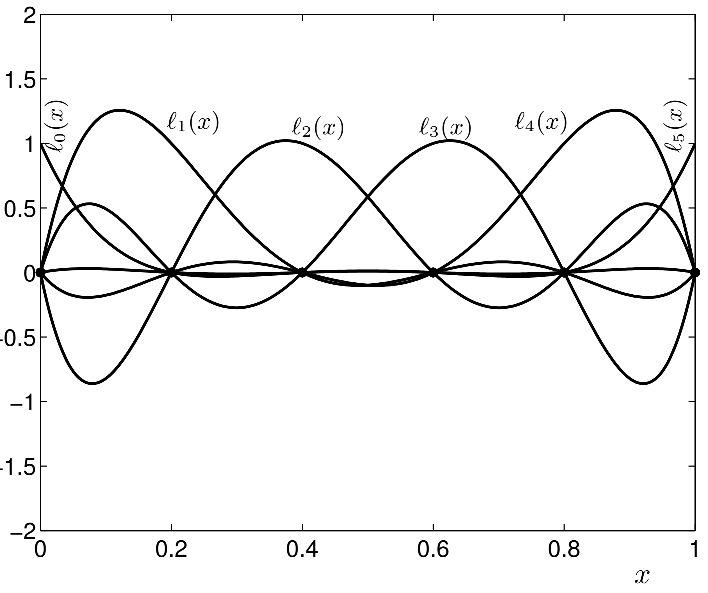
Just as we wrote the coefficients of the monomial and Newton bases as the solution of a linear system, we can do the same for the Lagrange basis:

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

Now the coefficient matrix is simply the identity.

An exercise in Problem Set 3 will investigate an important flexible and numerically stable method for constructing and evaluating Lagrange interpolants known as *barycentric interpolation.*

The plot below 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.



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 11: Interpolation Error Bounds

2.5. Convergence of interpolants.

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

Unfortunately, we do not have time to prove this in class.[†] 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).

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.

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

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

This result yields a bound for the worst error over the interval $[a, b]$:

$$\max_{x\in[a,b]} |f(x) - p_n(x)| \le \left(\max_{\xi\in[a,b]} \frac{|f^{(n+1)}(\xi)|}{(n+1)!}\right)\left(\max_{x\in[a,b]} \prod_{j=0}^n |x - x_j|\right). \tag{11.1}$$

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.

[†]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*, second ed., McGraw Hill, 1991.

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

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

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[‡] 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 $\phi^{(n+1)}(\xi) = 0$.

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:

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

[‡]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 $f'(\eta) = 0$.

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

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

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 ($f(x) = \sin(x)$). 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* $[-5, 5]$,

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

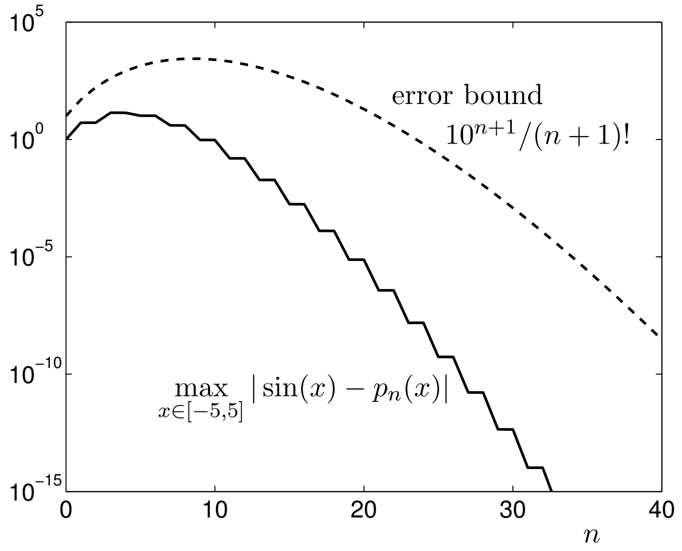
the worst case coming if all the interpolation points are clustered at an end of the interval $[-5, 5]$. Now our theorem ensures that

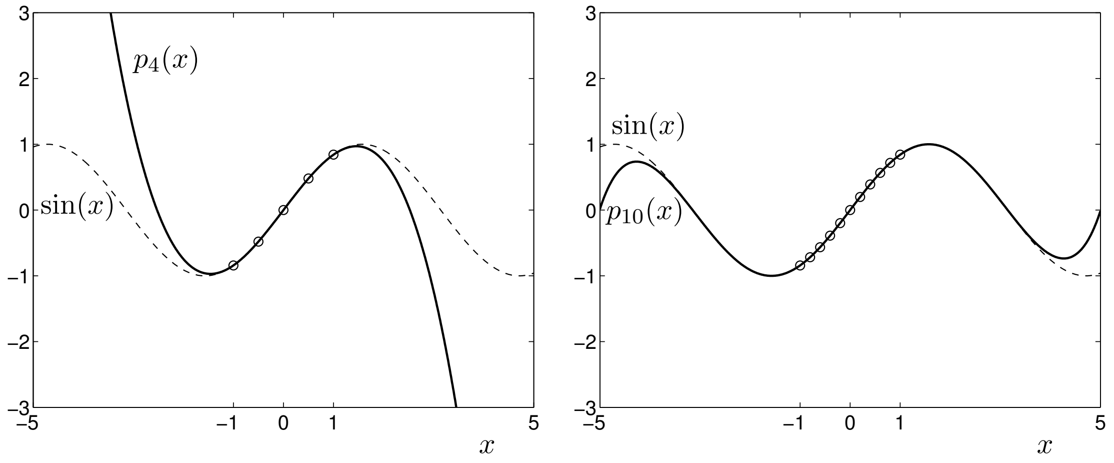
$$\max_{x\in[-5,5]} |\sin(x) - p_n(x)| \le \frac{10^{n+1}}{(n+1)!}.$$

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 (Runge's Example). The error bound (11.1) 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—but successive derivatives expose its flaw:

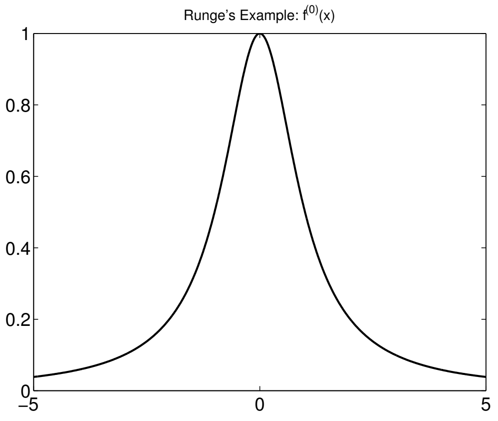
$$f'(x) = -\frac{2x}{(1+x^2)^2}, \qquad f''(x) = \frac{8x^2}{(1+x^2)^3} - \frac{2}{(1+x^2)^2}, \qquad f'''(x) = -\frac{48x^3}{(1+x^2)^4} + \frac{24x}{(1+x^2)^3}.$$

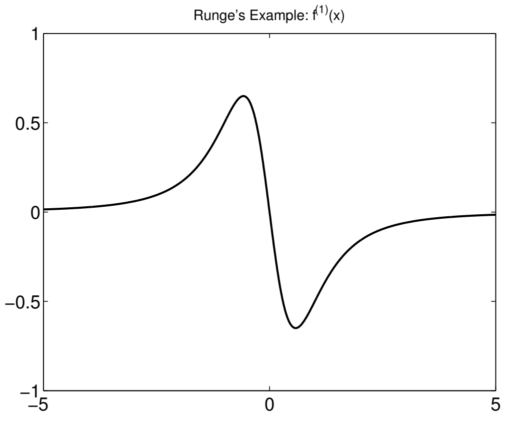
$$f^{(iv)}(x) = \frac{348x^4}{(1+x^2)^5} - \frac{288x^2}{(1+x^2)^4} + \frac{24}{(1+x^2)^3}, \qquad f^{(vi)}(x) = \frac{46080x^6}{(1+x^2)^7} - \frac{57600x^4}{(1+x^2)^6} + \frac{17280x^2}{(1+x^2)^5} - \frac{720}{(1+x^2)^4}.$$

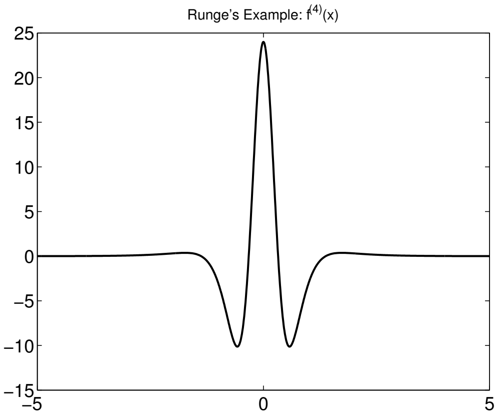
At certain points on $[-5,5]$, $f^{(n+1)}$ blows up more rapidly than $(n+1)!$, and the interpolation bound (11.1) suggests that p_n *will not* converge to f on $[-5,5]$ as n gets large. Not only does p_n fail to converge to f; the error between certain interpolation points gets enormous as n increases.

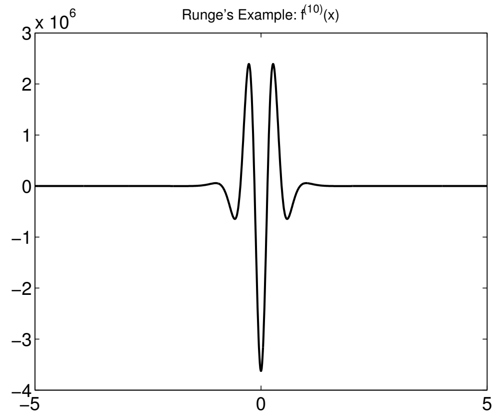
The following code uses MATLAB's Symbolic Toolbox to compute higher derivatives of the Runge function. Several of the resulting plots follow.[§] *Note how the scale on the vertical axis changes from plot to plot!*

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

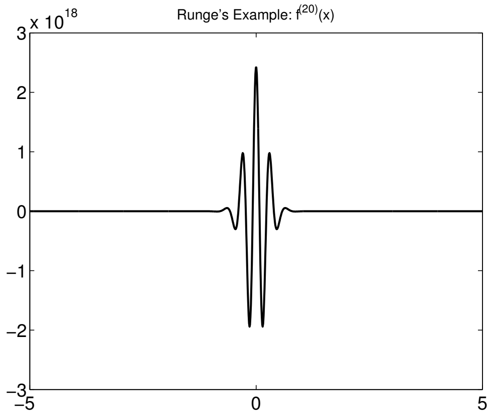


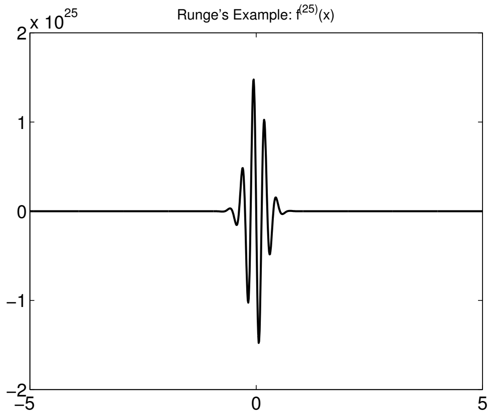






[§]Not all versions of MATLAB have the Symbolic Toolbox, but you should be able to run this code on any Student Edition, on Owlnet, and on the computers in the CAAM Department.





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 the effect were constructed by Marcinkiewicz and Grunwald in the 1930s. We close with two results of a more general nature.¶ 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* $\{\{x_j^{[n]}\}_{j=0}^n\}_{n=0}^\infty$.

Our first result is bad news.

Theorem (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 (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 $n \to \infty$.

¶An excellent exposition of these points is given in volume 3 of I. P. Natanson, *Constructive Function Theory*, Ungar, New York, 1965.

Lecture 12: Hermite Interpolation; Piecewise Polynomial Interpolation

An example in the previous lecture 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

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

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

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

which should certainly remind you of the interpolation error formula derived in the last lecture.

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 about both f and its derivatives. In this lecture we will discuss an alternative based on the best features of each of these ideas: use global information about both f and its derivatives. As a general rule, high degree polynomials are numerically fragile objects, though the degree of fragility varies with the polynomial basis used. (Barycentric interpolation, discussed on the third problem set, is ideal.) We will thus also discuss an alternative that sacrifices smoothness for accuracy, approximating f with many low degree polynomials that are accurate on small subintervals of $[a,b]$.

2.6. 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 $\{x_j\}$. A natural alternative is to 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 data points $\{(x_j, f(x_j), f'(x_j))\}_{j=0}^n$, we wish to construct a polynomial $h_n \in \mathcal{P}_{2n+1}$ such that

$$h_n(x_j) = f(x_j), \qquad h_n'(x_j) = f'(x_j). \tag{12.1}$$

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 2.4. We seek degree-$(2n+1)$ polynomials $\{A_k\}_{k=0}^n$ and $\{B_k\}_{k=0}^n$ such that

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

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

$$h_n(x) = \sum_{k=0}^n f(x_k)A_k(x) + \sum_{k=0}^n f'(x_k)B_k(x). \tag{12.2}$$

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

$$\begin{aligned} A_k(x) &:= (1 - 2(x-x_k)\ell_k'(x_k))\ell_k^2(x), \\ B_k(x) &:= (x-x_k)\ell_k^2(x). \end{aligned}$$

Note that since $\ell_k \in \mathcal{P}_n$, we have $A_k, B_k \in \mathcal{P}_{2n+1}$. Verification that these A_k and B_k, and their first derivatives, obtain the specified values at $\{x_j\}$ is straightforward, and left as an exercise on the third problem set.

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 (12.2).

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

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

Theorem. 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

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

[‡]Note that 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.

The proof of this latter result is directly analogous to the standard polynomial interpolation error given in the previous lecture. The reader is encouraged to understand how to derive this result.

2.6.1. 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 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*.

2.6.2. Hermite–Fejér interpolation. Another Hermite-like scheme initially sounds like a bad idea: Construct $h_n \in \mathcal{P}_{2n+1}$ such that

$$h_n(x_j) = f(x_j), \qquad h_n'(x_j) = 0.$$

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

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

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

For a proof of this result, see page 57 of Natanson, *Constructive Function Theory*, volume 3.

2.7. Piecewise polynomial interpolation.

We have seen through examples 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 $[a,b]$. A robust alternative is to replace the single high-degree interpolating polynomial over the entire interval with many low-degree polynomials valid between consecutive interpolation points.

2.7.1. 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,

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

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 $x \in [x_{j-1}, x_j]$.

[§]Note that all the s_j's are *linear* polynomials—the subscript j *does not* denote the polynomial degree.

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

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

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

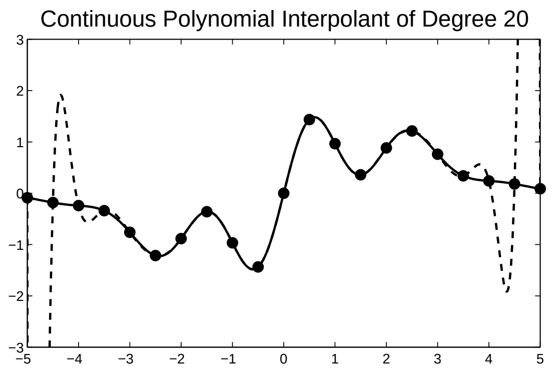
What could go wrong with this simple approach? The primary difficulty is that the interpolant is *continuous*, but generally not *continuously differentiable*. Still, these functions are easy to construct and cheap to evaluate, and can be very useful despite their simplicity.

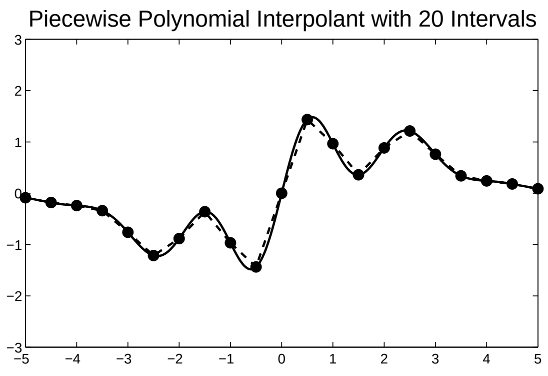
In MATLAB, linear interpolation is implemented in the commands `yy = interp1q(x,y,xx)` and `yy = interp1(x,y,xx,'linear')`.

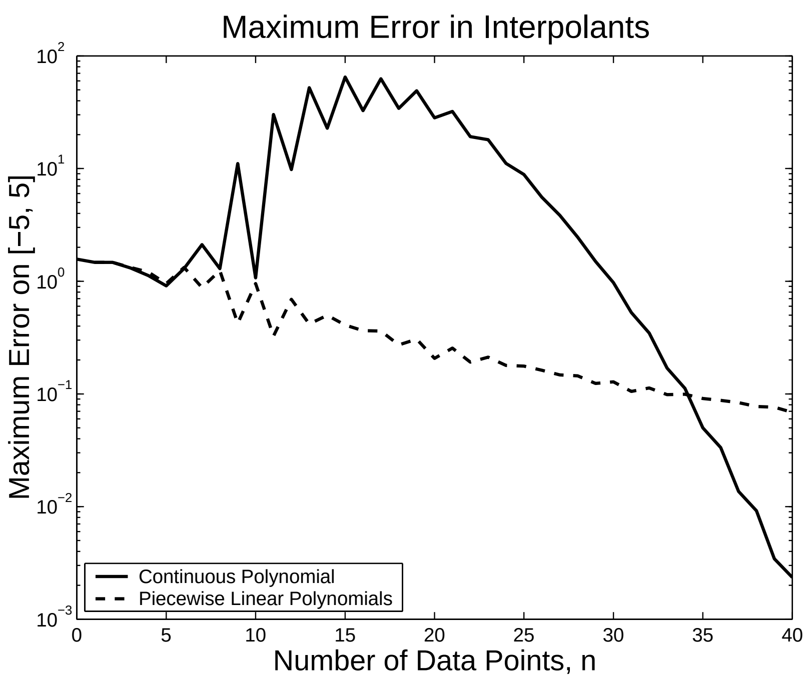
In the figures below, we compare piecewise linear interpolation with the continuous polynomial interpolation of §§2.2–2.4 over the interval $[-5, 5]$ for the function

$$f(x) = (x + \sin(3x))\mathrm{e}^{-x^2/6}.$$

(The interpolants are shown as dashed lines; the interpolation points are the solid dots.) Eventually, the continuous polynomial interpolants converge, but there is some initial period (when modest degree polynomials do not have enough 'wiggle' to capture the oscillations due to the $\sin(3x)$ term) where the continuous polynomial interpolant is poor, but piecewise linear interpolation does better.







The code that generated these plots is as follows:

```
% compare continuous polynomial interpolation with piecewise linear interpolation

 maxdeg = 40;
 xx = linspace(-5,5,1000);                  % fine grid
 fxx = (xx+sin(3*xx)).*exp(-(xx.^2)/6);     % true function value on a fine grid
 err_poly   = zeros(maxdeg+1,1); err_pwpoly = zeros(maxdeg,1);

% continuous polynomial approximation
 for n=0:maxdeg
    x = linspace(-5,5,n+1);
    fx = (x+sin(3*x)).*exp(-(x.^2)/6);
    p = polyfit(x,fx,n);              % MATLAB's polynomial interpolation routine
    err_poly(n+1) = max(abs(polyval(p,xx)-fxx));
    figure(1), clf
    plot(xx,fxx, 'b-','linewidth',2), hold on
    plot(xx,polyval(p,xx), 'r--','linewidth',2)
    plot(x,fx, 'r.','markersize',30)
    axis equal, axis([-5 5 -3 3])
    title(sprintf('Continuous Polynomial Interpolant of Degree %d',n),'fontsize',20)
    input('');
 end

% piecewise linear interpolation
 for n=1:maxdeg
    x = linspace(-5,5,n+1)';
    fx = (x+sin(3*x)).*exp(-(x.^2)/6);
    pxx = interp1(x,fx,xx,'linear');    % MATLAB's piecewise linear interpolation routine
    err_pwpoly(n) = max(abs(pxx-fxx));
    figure(2), clf
    plot(xx,fxx, 'b-','linewidth',2), hold on
    plot(xx,pxx, 'r--','linewidth',2)
    plot(x,fx, 'r.','markersize',30)
    axis equal, axis([-5 5 -3 3])
    title(sprintf('Piecewise Polynomial Interpolant with %d Intervals',n),'fontsize',20)
    input('');
 end

% compare the errors
 figure(3), clf
 semilogy([0:maxdeg], err_poly, 'b-','linewidth', 2), hold on
 semilogy([1:maxdeg], err_pwpoly, 'r--','linewidth', 2)
 legend('Continuous Polynomial', 'Piecewise Linear Polynomials',3)
 title('Maximum Error in Interpolants', 'fontsize', 20)
 xlabel('Number of Data Points, n', 'fontsize', 18)
 ylabel('Maximum Error on [-5, 5]', 'fontsize', 18)
```

Lecture 13: Splines

Spline fitting, our next topic in interpolation theory, plays an essential role in 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 2.7.1 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.

2.7.2. 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 don't have specific values for $S'(x_j) = f'(x_j)$ in mind; we just want the function $S(x)$ to be as *smooth* as possible. That motivates the main topic of this lecture, splines.

2.8. Splines.

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, and notice that we impose $2n + 2(n-1) = 4n - 2$ conditions on a total

[†]Apparently in the shipbuilding and aircraft industry where wooden splines were used in the pre-computer era, these knots went by the more colorful names of *rats*, *dogs*, or *ducks*. 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 where splines were thin pieces of wood running through 'rats,' see Gilbert Strang, *Introduction to Applied Mathematics*, Wellesley Cambridge Press, 1986.

of n cubic polynomials, which together have $4n$ degrees of freedom. Thus, there will be infinitely many choices for the function $S(x)$. There are several canonical choices for the two extra conditions that make S uniquely defined:

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

Natural cubic splines are the most common 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 'rats.'

Whatever we decide for the two additional conditions, we arrive at 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 tridiagonal coefficient matrix (zero everywhere except for the main diagonal and the first super- and sub-diagonals); we will see later in the semester that systems with this structure can be efficiently solved via Gaussian elimination. We could derive this linear system by directly enforcing the continuity conditions on the cubic polynomial that we have just described. (Try it!) However, we will prefer a more general approach that expresses the spline function $S(x)$ as the linear combination of special basis functions, which themselves are splines.

2.8.1. B-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 (cubics, discussed above, are a special case), can be developed in a convenient, numerically stable manner. 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 build our basis. Thus, add knots on either end of x_0 and x_n:

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

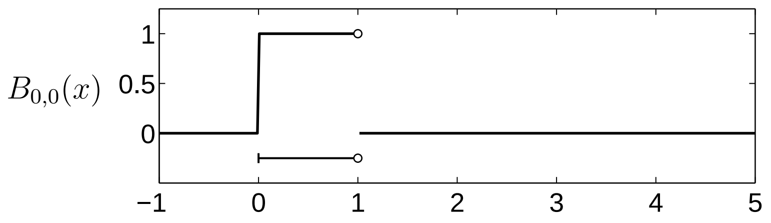
Given these knots, we can define the constant B-splines:

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

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 $B_{0,0}(x) \neq 0$.

[‡]Since the third derivative of a cubic is a constant, this requirement amounts to forcing $s_1 = s_2$ 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".

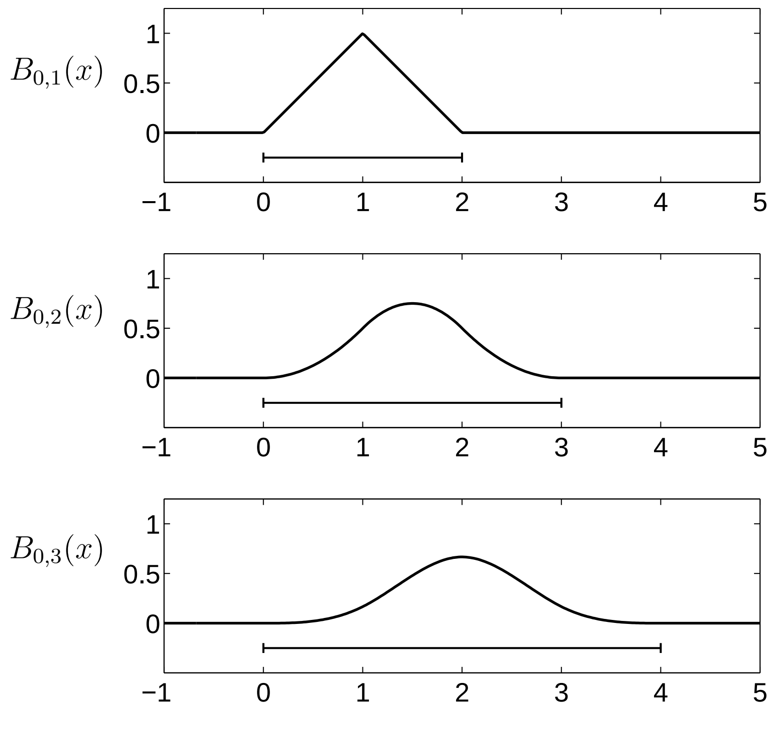
[§]See Süli and Mayers, *An Introduction to Numerical Analysis*, p. 300.



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

$$B_{j,k}(x) = \left(\frac{x - x_j}{x_{j+k} - x_j}\right) B_{j,k-1}(x) + \left(\frac{x_{j+k+1} - x}{x_{j+k+1} - x_{j+1}}\right) B_{j+1,k-1}(x). \tag{13.1}$$

While not immediately obvious from the formula, this construction ensures that $B_{j,k}$ will have one more continuous derivative than $B_{j,k-1}$ does. 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 $k \ge 1$.

The beauty 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:

$$S_k(x_j) = \sum_{j=-\infty}^{\infty} c_{j,k} B_{j,k}(x_j).$$

The compact support of the B-splines immediately suggest that we set most of the $c_{j,k}$ coefficients to zero, giving S_k as a finite sum. For $k \ge 1$, we are left with $n+k$ nontrivial $c_{j,k}$ variables to be determined from $n+1$ interpolation conditions. Thus for quadratic and higher degree splines, we require extra conditions to get a unique S_k. To illustrate how the spline S_k is ultimately arrived at, we walk through several cases slowly.

Constant splines, $k = 0$. In this simple case, the basis functions are piecewise constants, and so the spline $S_0(x)$ will itself be piecewise constant (hence discontinuous, in general). The coefficients $c_{j,0}$ in the expansion

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

are completely determined by the interpolation requirement: $S_0(x_j) = f_j$ for $j = 0, \ldots, n$. Since $B_{j,0}(x_\ell) = 0$ if $j \ne \ell$, and $B_{\ell,0}(x_\ell) = 1$ (recall the plot of $B_{0,0}(x)$ shown earlier),

$$S_0(x_\ell) = c_{\ell,0} B_{j,0}(x_\ell) = c_{\ell,0}.$$

The interpolation condition $S_0(x_\ell) = f_\ell$ implies $c_{\ell,0} = f_\ell$. Since we do not care what value the spline takes outside $[x_0, x_n]$, we set $c_{j,0} = 0$ for $j < 0$ and $j > n$:

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

Linear splines, $k = 1$. Linear splines are similarly simple to construct. The support of $B_{j,1}(x)$ is (x_j, x_{j+2}). That is, $B_{j,1}(x_\ell) = 0$ if $\ell \ne j+1$. Substituting x_j into the recursion (13.1) for $B_{j,1}(x)$ gives $B_{j,1}(x_{j+1}) = 1$, as apparent from the previous plot of $B_{0,1}(x)$. Thus

$$f_\ell = S_1(x_\ell) = \sum_{j=-\infty}^{\infty} c_{j,1} B_{j,1}(x_\ell) = c_{\ell-1,1}.$$

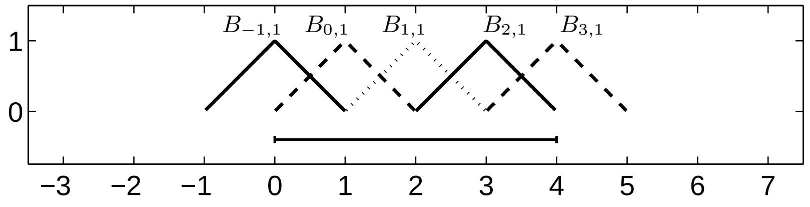
Again ignoring all splines that are zero throughout $[x_0, x_n]$, we have

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

It is instructive now to look at an example.

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

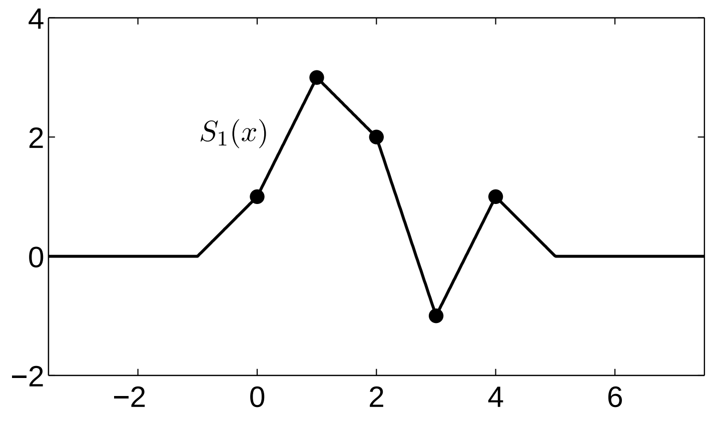
The desired linear spline S_1 is a linear combination of the five B-splines $B_{-1,1}$, $B_{0,1}$, $B_{1,1}$, $B_{2,1}$, and $B_{3,1}$, shown in the plot below.



More explicitly:

$$\begin{aligned} S_1(x) &= f_0B_{-1,1}(x) + f_1B_{0,1}(x) + f_2B_{1,1}(x) + f_3B_{2,1}(x) + f_4B_{3,1}(x) \\ &= B_{-1,1}(x) + 3B_{0,1}(x) + 2B_{1,1}(x) - B_{2,1}(x) + B_{3,1}(x). \end{aligned}$$

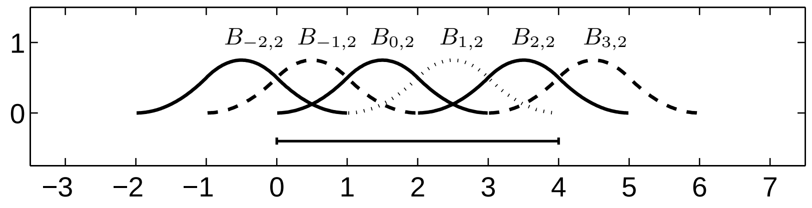
The spline S_1 is shown in the plot below. Note that linear splines are simply C^0 functions that interpolate a given data set—between the knots, they are identical to the piecewise linear functions constructed in §2.7.1. Note that $S_1(x) = 0$ for all $x \notin (x_{-1}, x_{n+1})$. This is a general 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.



So far, you might be skeptical about the use of splines: all we have done is construct an alternative basis for piecewise constant and piecewise linear interpolants. The advantage of splines will be evident as the degree of the approximating polynomials increases.

Quadratic splines, $k = 2$. The construction of quadratic B-splines from the linear splines via the recurrence (13.1) 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 plot of $B_{0,2}$ shown earlier.

To understand the implications for determining the coefficients $c_{j,2}$, it is best to return to our concrete example. Suppose we are given data at the five points $x_0, x_1, \ldots, x_4$, as defined in the previous table. Now the six splines $B_{-2,2}$, $B_{-1,2}$, $\ldots$, $B_{3,2}$ all have nontrivial values on $[x_0, x_4]$, as shown in the following figure.



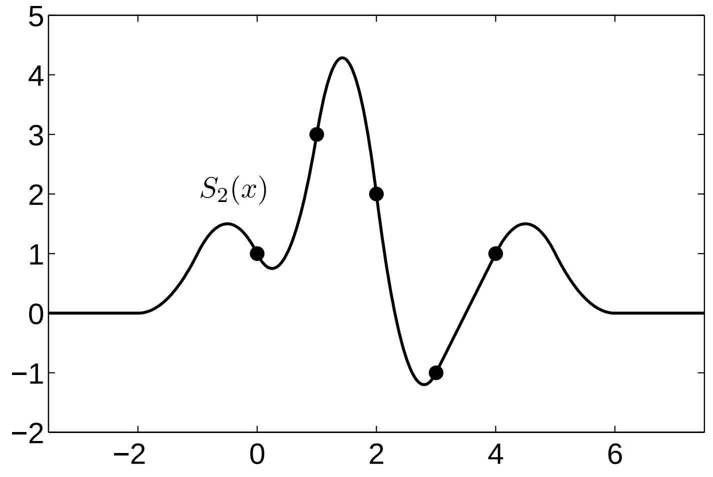
In general, S_2 will have the form

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

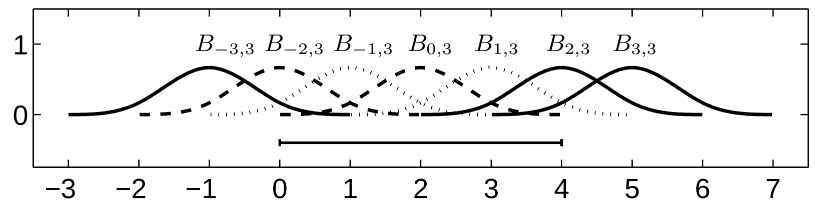
Since $n = 4$ in our specific example,

$$S_2(x) = \sum_{j=-2}^{3} c_{j,2} B_{j,2}(x).$$

This equation has $n + 2 = 6$ nontrivial coefficients, but only five requirements: $S_2(x_j) = f_j$ for $j = 0, \ldots, 4$. Thus, there are infinitely many quadratic splines that satisfy the interpolation conditions. How to choose one among them? Impose some extra condition, such as $S_2'(x_0) = 0$, or simply demand that $c_{-2,2} = 0$. In the figures below, we've arbitrarily set $c_{-2,2} = c_{n-1,2}$. Note that S_2 is supported on (x_{-2}, x_{n+2}).



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. The resulting S_3 will be a C^2 function that interpolates specified data. For the previous example with knots $x_0, \ldots, x_4$, the spline will be a linear combination of the $7 = n + k$ B-splines shown below.



To determine the coefficients of S_3 in the linear combination

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

use the $n+1$ conditions imposed by the interpolation requirement: $S_3(x_j) = f_j$. Infinitely many cubic splines satisfy these interpolation conditions; two independent requirements must be imposed to determine a unique spline. Recall the three alternatives discussed earlier: 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 introduces the following equations:

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

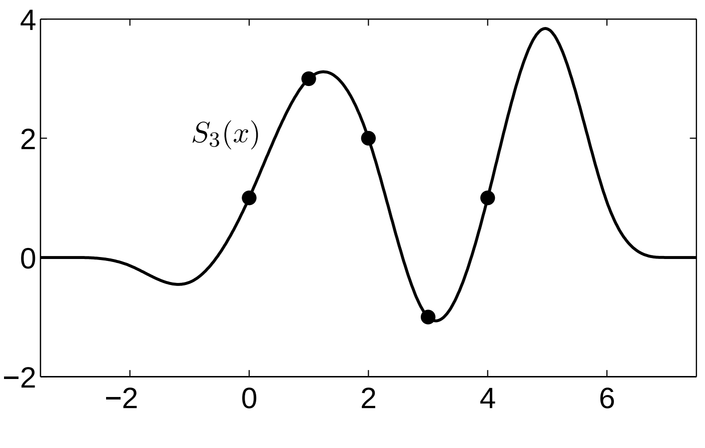
$$(x_{n+2} - x_{n-1})c_{n-3,3} - (x_{n+2} + x_{n+1} - x_{n-1} - x_{n-2})c_{n-2,3} + (x_{n+1} - x_{n-2})c_{n-1,3} \;=\; 0.$$

If the knots are equally spaced, $x_j = x_0 + jh$ for some fixed $h > 0$, these natural boundary conditions simplify:

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

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

With these in hand, the coefficients for S_3 are completely determined. In the notes for the next lecture, we will give details about construction of the associated linear system of equations. The plot below shows the cubic spline with natural boundary conditions, based on the data used in our previous example. 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 of the natural boundary conditions: by forcing the second derivative to be zero, 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}).



Of course, one can go to higher degree splines 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).

Some omissions. The great utility of B-splines in engineering has led to the development of the subject far beyond these meager 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.

Lecture 14: Matrix Formulation of Spline Interpolants

The last lecture described the construction of spline interpolants as the linear combination of basis functions called B-splines. We spent considerable time studying the construction of these B-splines. Now we will turn our attention to the ultimate goal: construction of the spline interpolants themselves.

2.8.2. Matrix formulation of spline interpolation.

Given a system of knots $\{x_j\}$, we seek the spline S_k that is a degree-k polynomial on each subinterval (x_j, x_{j+1}) with $S_k \in C^{k-1}(\mathbb{R})$ for $k \geq 1$. In terms of the degree-k B-spline basis $\{B_{j,k}\}$, we have

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

with the convention of setting $c_{j,k} = 0$ if $B_{j,k}$ is zero throughout the interval $[x_0, x_n]$. Since $B_{j,k}$ is supported (i.e., nonzero) only on the interval (x_j, x_{j+k+1}), we have

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

except for $k = 0$, where the limits on the sum go from $j = 0$ to $j = n$.

The case of $k = 0$ (piecewise constant splines) is trivial, as $c_{j,0} = f_j$ for $0 \leq j \leq n$. Hence, from now on assume $k \geq 1$, in which case we have $n + k$ unknowns $\{c_{j,k}\}$ in the above linear combination for S_k. However, the interpolation requirement $S_k(x_j) = f_j$ provides only $n + 1$ constraints.[†] Overall, $(n + k) - (n + 1) = k - 1$ additional constraints are needed to uniquely specify the spline S_k.

The freedom we have just described must manifest itself in the associated linear algebra. For $\ell = 0, \ldots, n$, the interpolation condition $S_k(x_\ell) = f_\ell$, i.e.,

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

can be written as a row of the linear system

$$\begin{bmatrix} B_{-k,k}(x_0) & B_{-k+1,k}(x_0) & \cdots & B_{n-1,k}(x_0) \\ B_{-k,k}(x_1) & B_{-k+1,k}(x_1) & \cdots & B_{n-1,k}(x_1) \\ \vdots & \vdots & \ddots & \vdots \\ B_{-k,k}(x_n) & B_{-k+1,k}(x_n) & \cdots & B_{n-1,k}(x_n) \end{bmatrix} \begin{bmatrix} c_{-k,k} \\ c_{-k+1,k} \\ \vdots \\ c_{n-1,k} \end{bmatrix} = \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_n \end{bmatrix}. \tag{14.1}$$

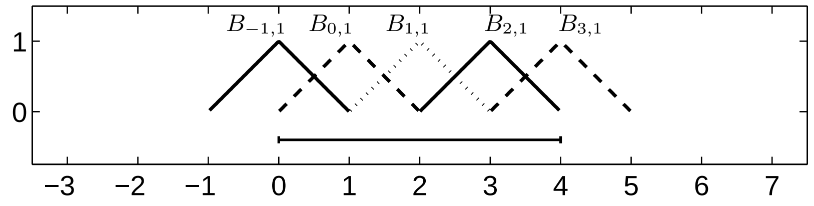
The matrix on the left has $n + 1$ rows and $n + k$ columns.

[†]What about the requirement $S_k \in C^{k-1}(\mathbb{R})$ that ensures continuity and smoothness? Does it contribute any additional equations that can be used to uniquely determine the $\{c_{j,k}\}$, as was the case with Hermite interpolation? No: By construction, each B-spline satisfies this requirement already, so there are no extra continuity constraints lurking around for us to impose.

Linear splines. To construct the matrix in the linear system (14.1), we must know the value of B-splines at the various knots. When $k = 1$, one can easily see that

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

as confirmed in the following plot of several $B_{j,1}$ for knots $x_j = j$.



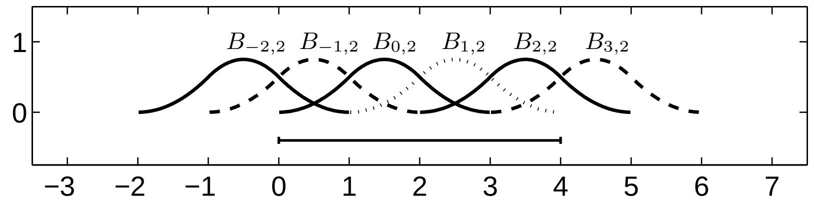
The linear system (14.1) will involve a $(n+1) \times (n+1)$ square matrix; in fact, it takes the trivial form

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

so the coefficients are simply

$$c_{j-1,1} = f_j, \qquad j = 0, \ldots, n.$$

Quadratic splines. When $k = 2$, the situation becomes more interesting, for now the matrix in equation (14.1) has dimension $(n+1) \times (n+2)$: there are more variables than constraints, reflecting the fact that there are infinitely many quadratic splines that interpolate the data. To determine the matrix entries, first consider the following plot of several quadratic B-splines based again on the knots $x_j = j$.



For simplicity, we shall work out explicit entries in the case that the knots are uniformly spaced ($x_j = x_0 + jh$ for fixed $h > 0$). The recursion that defines quadratic B-splines then takes the form

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

We know by construction that $B_{j,2}(x_\ell) = 0$ unless $\ell = j+1$ or $\ell = j+2$, and in those latter cases

$$B_{j,2}(x_{j+1}) = \frac{1}{2h}(x_{j+1}-x_j)B_{j,1}(x_{j+1}) + \frac{1}{2h}(x_{j+3}-x_{j+1})B_{j+1,1}(x_{j+1}) = \frac{1}{2h}h\cdot 1 + \frac{1}{2h}2h\cdot 0 = 1/2;$$

$$B_{j,2}(x_{j+2}) = \frac{1}{2h}(x_{j+2}-x_j)B_{j,1}(x_{j+2}) + \frac{1}{2h}(x_{j+3}-x_{j+2})B_{j+1,1}(x_{j+2}) = \frac{1}{2h}2h\cdot 0 + \frac{1}{2h}h\cdot 1 = 1/2.$$

In summary,

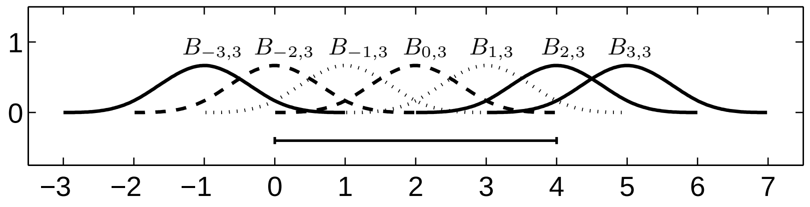
$$B_{j,2}(x_\ell) = \begin{cases} 1/2, & \ell = j+1 \\ 1/2, & \ell = j+2 \\ 0, & \ell \notin \{j+1, j+2\}. \end{cases}$$

Thus, the linear system (14.1) takes the form

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

where the blank entries are zero. To make this a well-determined system, one needs to add one more constraint. Effectively, this appends a new row to the matrix and a new entry to the right hand side vector. The operation is very similar to the more important case of cubic splines, which we will now work through in detail.

Cubic splines. This famous case with $k = 3$ is a little more intricate. In this case, the matrix in (14.1) has $n+1$ rows but $n+3$ columns, so we need to impose two additional constraints. Recall that the cubic B-splines take the following form, where once again $x_j = j$.



Assuming again that the knots are uniformly spaced, $x_j = x_0 + jh$, then with a little calculation one can confirm that

$$B_{j,3}(x_\ell) = \begin{cases} \alpha, & \ell = j+1 \\ \beta, & \ell = j+2 \\ \alpha, & \ell = j+3 \\ 0, & \ell \notin \{j+1, j+2, j+3\}. \end{cases}$$

where α and β are simple constants whose computation is left as an exercise for the reader.[‡]

[‡]Use the recurrence that defines the B-splines, together with the known values of $B_{j,2}(x_\ell)$ given above. More labor is required to compute the value of $B_{j,3}(x)$ when x is not a knot, but remember that you do not need to know these intermediate values to set up the linear system.

The linear system (14.1) now takes the form

$$\begin{bmatrix} \alpha & \beta & \alpha & & & \\ & \alpha & \beta & \alpha & & \\ & & \ddots & \ddots & \ddots & \\ & & & \alpha & \beta & \alpha \end{bmatrix} \begin{bmatrix} c_{-3,3} \\ c_{-2,3} \\ c_{-1,3} \\ \vdots \\ c_{n-1,3} \end{bmatrix} = \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_n \end{bmatrix}, \tag{14.2}$$

where the blank entries are zero.

To uniquely determine the spline coefficients, we impose the *natural spline* conditions, $S_3''(x_0) = S_3''(x_n) = 0$. With considerable tedious labor, one can verify that these conditions reduce to

$$\begin{aligned} (x_2 - x_{-1})c_{-3,3} - (x_2 + x_1 - x_{-1} - x_{-2})c_{-2,3} + (x_1 - x_{-2})c_{-1,3} &= 0 \\ (x_{n+2} - x_{n-1})c_{n-3,3} - (x_{n+2} + x_{n+1} - x_{n-1} - x_{n-2})c_{n-2,3} + (x_{n+1} - x_{n-2})c_{n-1,3} &= 0. \end{aligned}$$

If the knots are equally spaced ($x_j = x_0 + jh$) these conditions simplify to

$$\begin{aligned} 3hc_{-3,3} - 6hc_{-2,3} + 3hc_{-1,3} &= 0 \\ 3hc_{n-3,3} - 6hc_{n-2,3} + 3hc_{n-1,3} &= 0. \end{aligned}$$

Dividing these equations by $h > 0$,

$$\begin{aligned} 3c_{-3,3} - 6c_{-2,3} + 3c_{-1,3} &= 0 \\ 3c_{n-3,3} - 6c_{n-2,3} + 3c_{n-1,3} &= 0. \end{aligned}$$

We wish to augment the linear system (14.2) with these two equations. It is conventional to insert the natural spline conditions at x_0 and x_n in the first and last rows of the new system, respectively. This gives

$$\begin{bmatrix} 3 & -6 & 3 & & & \\ \alpha & \beta & \alpha & & & \\ & \alpha & \beta & \alpha & & \\ & & \ddots & \ddots & \ddots & \\ & & & \alpha & \beta & \alpha \\ & & & 3 & -6 & 3 \end{bmatrix} \begin{bmatrix} c_{-3,3} \\ c_{-2,3} \\ c_{-1,3} \\ \vdots \\ c_{n-2,3} \\ c_{n-1,3} \end{bmatrix} = \begin{bmatrix} 0 \\ f_0 \\ f_1 \\ \vdots \\ f_n \\ 0 \end{bmatrix},$$

which is now a square $(n+3) \times (n+3)$ linear system that one can solve to obtain the unique coefficients $\{c_{j,3}\}$.

Lecture 15: Trigonometric Interpolation

2.9 Trigonometric Interpolation.

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 $n = 5$. The interpolant will be built as a linear combination of the 2π-periodic functions

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

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, $f(x_0) = f(x_n)$, 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

$$t_n(x) = \sum_{k=0}^{n-1} c_k b_k(x)$$

such that

$$t_n(x_j) = f(x_j), \quad j = 0, \ldots, n-1.$$

To compute the unknown coefficients $c_0, \ldots, c_5$, we 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. As a practical matter, one might wonder how accurately this system can be solved. What can be said of the *conditioning* of the matrix?

Rather than investigating this question directly, we shall first transform to a slightly more convenient basis. Recall Euler's formula,

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

which also implies that

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

From this it follows that

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

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

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

[†]This means that f is continuous throughout $\mathbb{R}$ and $f(x) = f(x + 2\pi)$ for all $x \in \mathbb{R}$.

We shall thus write the interpolant t_n in the form

$$t_n(x) = \sum_{k=-2}^{2} \gamma_k \mathrm{e}^{\mathrm{i}kx} = \sum_{k=-2}^{2} \gamma_k (\mathrm{e}^{\mathrm{i}x})^k.$$

This last sum is written in a manner that emphasizes that t_n is *a polynomial in the variable* $\mathrm{e}^{\mathrm{i}x}$, and hence we call t_n a *trigonometric polynomial*. In this basis, the interpolation conditions give the linear system

$$\begin{bmatrix} \mathrm{e}^{-2\mathrm{i}x_0} & \mathrm{e}^{-\mathrm{i}x_0} & \mathrm{e}^{0\mathrm{i}x_0} & \mathrm{e}^{\mathrm{i}x_0} & \mathrm{e}^{\mathrm{i}2x_0} \\ \mathrm{e}^{-2\mathrm{i}x_1} & \mathrm{e}^{-\mathrm{i}x_1} & \mathrm{e}^{0\mathrm{i}x_1} & \mathrm{e}^{\mathrm{i}x_1} & \mathrm{e}^{\mathrm{i}2x_1} \\ \mathrm{e}^{-2\mathrm{i}x_2} & \mathrm{e}^{-\mathrm{i}x_2} & \mathrm{e}^{0\mathrm{i}x_2} & \mathrm{e}^{\mathrm{i}x_2} & \mathrm{e}^{\mathrm{i}2x_2} \\ \mathrm{e}^{-2\mathrm{i}x_3} & \mathrm{e}^{-\mathrm{i}x_3} & \mathrm{e}^{0\mathrm{i}x_3} & \mathrm{e}^{\mathrm{i}x_3} & \mathrm{e}^{\mathrm{i}2x_3} \\ \mathrm{e}^{-2\mathrm{i}x_4} & \mathrm{e}^{-\mathrm{i}x_4} & \mathrm{e}^{0\mathrm{i}x_4} & \mathrm{e}^{\mathrm{i}x_4} & \mathrm{e}^{\mathrm{i}2x_4} \end{bmatrix} \begin{bmatrix} \gamma_{-2} \\ \gamma_{-1} \\ \gamma_0 \\ \gamma_1 \\ \gamma_2 \end{bmatrix} = \begin{bmatrix} f(x_0) \\ f(x_1) \\ f(x_2) \\ f(x_3) \\ f(x_4) \end{bmatrix},$$

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 there is fascinating structure here. Notice that a generic entry of this matrix has the form $\mathrm{e}^{\ell \mathrm{i}x_k}$ for $\ell = -(n-1)/2, \ldots, (n-1)/2$ and $k = 0, \ldots, n-1$. Since $x_k = 2\pi k/n$, we can rewrite this entry as

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

where $\omega = \mathrm{e}^{2\pi\mathrm{i}/n}$ is an *nth root of unity* (since $\omega^n = 1$). In the $n = 5$ case, the linear system can thus be written as

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

Denote this system by $\mathbf{Fg} = \mathbf{f}$. 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 has Vandermonde structure*! From our past experience, we might well expect such a matrix to be highly ill-conditioned. Before jumping to this conclusion, we shall examine $\mathbf{F}^*\mathbf{F}$. To form $\mathbf{F}^*$ (the conjugate-transpose of $\mathbf{F}$), we note that $\overline{\omega^{-\ell}} = \omega^\ell$, so

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

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

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

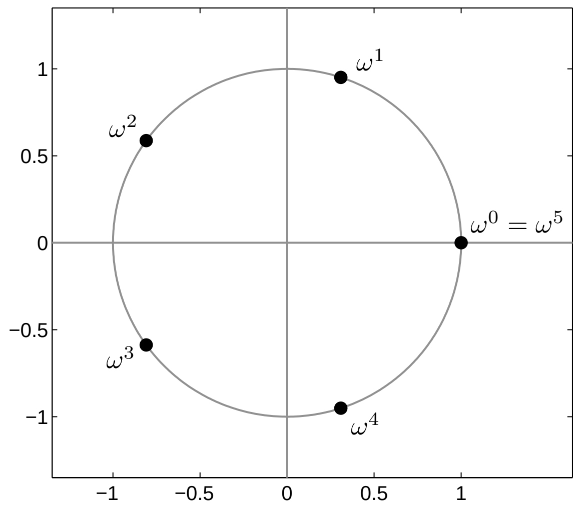
On the diagonal, when $\ell = k$, we simply have

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

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 below.



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 have arrived at the conclusion 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 $\mathbf{F}\mathbf{g} = \mathbf{f}$ can be immediately solved without the need for any factorization of $\mathbf{F}$:

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

The ready formula for $\mathbf{F}^{-1}$ is reminiscent of a unitary matrix. (Recall that $\mathbf{Q} \in \mathbb{C}^{n\times n}$ is unitary if and only if $\mathbf{Q}^{-1} = \mathbf{Q}^*$.) In fact, we see that the matrices

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

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! 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 of the real line. This distinction makes all the difference between an unstable system 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 $\mathbf{g} = n^{-1}\mathbf{F}^*\mathbf{f}$ are the *discrete Fourier coefficients* of the data in $\mathbf{f}$. 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 $\mathbf{g} = n^{-1}\mathbf{F}^*\mathbf{f}$ in only $O(n \log n)$ operations – a procedure (apparently known earlier to Gauss) that we now famously call the *Fast Fourier Transform* (FFT).

We can thus summarize this lecture as follows: the FFT of a vector of uniform samples of a 2π-periodic function f is simply the set of coefficients for the trigonometric interpolant to f at those sample points.

Lecture 16: Discrete Linear Least Squares

3. Approximation Theory.

Interpolation with high-degree polynomials is not always the best way to approximate a function: we have seen an example where the polynomial diverges from the function it is meant to approximate between nodes as the polynomial degree grows. Numerical errors add further complexity. There is also a more fundamental objection: if the function or data you are modeling cannot be approximated well with low-degree polynomials, perhaps you should be using another class of functions (rational functions, trigonometric functions, or e^x and e^{-x}, etc.). Piecewise polynomial interpolation provides an alternative, provided smoothness is not a concern.

In this lecture, we consider an alternative to interpolation, *approximation by polynomials*:

Given $f \in C[a,b]$ and $m+1$ points $\{x_j\}_{j=0}^{n}$ satisfying $a \le x_0 < x_1 < \cdots < x_m \le b$, determine some $p \in \mathcal{P}_n$ $(n \le m)$ such that

$$p(x_j) \approx f(x_j) \qquad \text{for } j = 0, \ldots, m.$$

Notice that this is essentially just the standard interpolation problem when $m = n$, in which case we have seen that there exists a unique $p \in \mathcal{P}_n$ such that $p(x_j) = f(x_j)$ for $0 \le j \le n$. However, when $m > n$, there generally will be no $p \in \mathcal{P}_n$ that delivers equality $p(x_j) = f(x_j)$ for all $j = 0, \ldots, m$. We must settle for approximation, $p(x_j) \approx f(x_j)$, together with a method for quantifying this approximation. For example, we could choose p to minimize the maximum error at any grid point:

$$\min_{p \in \mathcal{P}_n} \max_{0 \le j \le m} |f(x_j) - p(x_j)|,$$

or the sum of the squares of the errors:

$$\min_{p \in \mathcal{P}_n} \sum_{j=0}^{m} |f(x_j) - p(x_j)|^2. \tag{16.1}$$

Other alternatives include ignoring the specific points $x_0, \ldots, x_n$, and generalizing the above two measures to the entire interval of interest, $[a,b]$:

$$\min_{p \in \mathcal{P}_n} \max_{x \in [a,b]} |f(x) - p(x)| \qquad \text{or} \qquad \min_{p \in \mathcal{P}_n} \int_a^b |f(x) - p(x)|^2.$$

Each of these different error metrics gives rise different 'optimal' approximations, and inspire distinct algorithms for their construction. In this lecture, we study minimization of the second error, (16.1), the sum of the square of the errors at the grid points. Suppose we seek p in the monomial basis, $p(x) = c_0 + c_1 x + c_2 x^2 + \cdots c_n x^n$. Define

$$\mathbf{A} = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ 1 & x_2 & x_2^2 & \cdots & x_2^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_m & x_m^2 & \cdots & x_m^n \end{bmatrix}, \qquad \mathbf{c} = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix}, \qquad \mathbf{f} = \begin{bmatrix} f(x_0) \\ f(x_1) \\ f(x_2) \\ \vdots \\ f(x_m) \end{bmatrix}.$$

In this notation, problem (16.1) is equivalent to the matrix optimization problem

$$\min_{\mathbf{c}\in\mathbb{C}^n} \|\mathbf{f}-\mathbf{A}\mathbf{c}\|_2.$$

Because we are minimizing a sum of squares in (16.1), this is called a *least squares problem.* (Sometimes we say that the linear system $\mathbf{A}\mathbf{x}=\mathbf{b}$ is *overdetermined*, meaning that an excessive number of constraints prevents the system from having an exact solution $\mathbf{x}$. Some authors will thus describe the approximation procedure we are about to study as 'solving $\mathbf{A}\mathbf{x}=\mathbf{b}$ in the least-squares sense.')

3.1. Discrete least squares problems.

We focus our attention to the general least squares problem

$$\min_{\mathbf{x}\in\mathbb{C}^n} \|\mathbf{b}-\mathbf{A}\mathbf{x}\|_2,$$

where $\mathbf{A}\in\mathbb{C}^{m\times n}$, $\mathbf{b}\in\mathbb{C}^m$ with $m\ge n$.

Before tackling this problem, recall the Fundamental Theorem of Linear Algebra, stated in Lecture 2. The range of $\mathbf{A}$ is the subspace

$$\mathrm{Ran}(\mathbf{A}) = \{\mathbf{A}\mathbf{x} : \mathbf{x}\in\mathbb{C}^n\},$$

and the left null space is

$$\mathrm{Ker}(\mathbf{A}^*) = \{\mathbf{y}\in\mathbb{C}^m : \mathbf{A}^*\mathbf{y}=0\}.$$

The Fundamental Theorem of Linear Algebra states that

$$\mathbb{C}^m = \mathrm{Ran}(\mathbf{A}) \oplus \mathrm{Ker}(\mathbf{A}^*),$$

and also that $\mathrm{Ran}(\mathbf{A}) \perp \mathrm{Ker}(\mathbf{A}^*)$. This immediately implies that any $\mathbf{b}\in\mathbb{C}^m$ can be written uniquely as $\mathbf{b}=\mathbf{b}_R+\mathbf{b}_N$, where $\mathbf{b}_R\in\mathrm{Ran}(\mathbf{A})$ and $\mathbf{b}_N\in\mathrm{Ker}(\mathbf{A}^*)$ with $\mathbf{b}_R\perp\mathbf{b}_N$.

For any $\mathbf{x}\in\mathbb{C}^n$, define the *residual* vector

$$\mathbf{r}=\mathbf{b}-\mathbf{A}\mathbf{x}.$$

Decomposing $\mathbf{b}=\mathbf{b}_R+\mathbf{b}_N$ as described above, we have

$$\begin{aligned}\mathbf{r} &= \mathbf{b}-\mathbf{A}\mathbf{x}\\ &= \mathbf{b}_R-\mathbf{A}\mathbf{x}+\mathbf{b}_N.\end{aligned}$$

One can immediately see from the definition of $\mathrm{Ran}(\mathbf{A})$ that $\mathbf{A}\mathbf{x}\in\mathrm{Ran}(\mathbf{A})$. Since $\mathbf{b}_R\in\mathrm{Ran}(\mathbf{A})$, too, and $\mathrm{Ran}(\mathbf{A})$ is a subspace, it must be that $\mathbf{b}_R-\mathbf{A}\mathbf{x}\in\mathrm{Ran}(\mathbf{A})$. The Fundamental Theorem of Linear Algebra ensures that

$$\begin{aligned}\|\mathbf{r}\|_2^2 &= \|\mathbf{b}-\mathbf{A}\mathbf{x}\|_2^2\\ &= \|(\mathbf{b}_R-\mathbf{A}\mathbf{x})+\mathbf{b}_N\|_2^2\\ &= \|\mathbf{b}_R-\mathbf{A}\mathbf{x}\|_2^2+\|\mathbf{b}_N\|_2^2,\end{aligned}$$

since $\mathbf{b}_R-\mathbf{A}\mathbf{x}\perp\mathbf{b}_N$.[†] Thus,

$$\min_{\mathbf{x}\in\mathbb{C}^n}\|\mathbf{b}-\mathbf{A}\mathbf{x}\|_2^2 = \min_{\mathbf{x}\in\mathbb{C}^n}\|\mathbf{b}_R-\mathbf{A}\mathbf{x}\|_2^2+\|\mathbf{b}_N\|_2^2.$$

[†]The last equality is simply the Pythagorean Theorem: The vectors $\mathbf{b}_R-\mathbf{A}\mathbf{x}$ and $\mathbf{b}_N$ are orthogonal, so they meet at a right angle; $\mathbf{r}$ is the hypotenuse of the right-triangle whose legs are $\mathbf{b}_R-\mathbf{A}\mathbf{x}$ and $\mathbf{b}_N$. This result can be easily verified by directly computing $\|\mathbf{r}\|^2=\mathbf{r}^*\mathbf{r}=((\mathbf{b}_R-\mathbf{A}\mathbf{x})+\mathbf{b}_N)^*((\mathbf{b}_R-\mathbf{A}\mathbf{x})+\mathbf{b}_N)$.

Note that $\mathbf{x}$ appears nowhere in the last term of this sum: the $\|\mathbf{b}_N\|_2^2$ component is inaccessible regardless of the choice of $\mathbf{x}$. Thus, the best hope for minimizing $\|\mathbf{b}-\mathbf{A}\mathbf{x}\|_2$ is to minimize $\|\mathbf{b}_R-\mathbf{A}\mathbf{x}\|_2$. As this term is always non-negative, the optimal solution is some $\mathbf{x}$ that makes $\mathbf{b}_R-\mathbf{A}\mathbf{x}=\mathbf{0}$. Since $\mathbf{b}_R \in \operatorname{Ran}(\mathbf{A})$, the definition of $\operatorname{Ran}(\mathbf{A})$ guarantees there must be some $\mathbf{x}\in\mathbb{C}^n$ such that $\mathbf{b}_R=\mathbf{A}\mathbf{x}$.

3.1.1. Solving least squares problems via the normal equations.

There are several ways to obtain this solution, $\mathbf{x}$. The first relies on a simple trick. If $\mathbf{b}_R=\mathbf{A}\mathbf{x}$, then

$$\mathbf{b}-\mathbf{A}\mathbf{x}=(\mathbf{b}_R-\mathbf{A}\mathbf{x})+\mathbf{b}_N=\mathbf{b}_N,$$

and since $\mathbf{b}_N\in\operatorname{Ker}(\mathbf{A}^*)$, we immediately have

$$\mathbf{A}^*(\mathbf{b}-\mathbf{A}\mathbf{x})=\mathbf{A}^*\mathbf{b}_N=\mathbf{0}.$$

We rearrange this equation into

$$\mathbf{A}^*\mathbf{A}\mathbf{x}=\mathbf{A}^*\mathbf{b}. \tag{16.2}$$

If $\mathbf{A}$ is full rank (i.e., $\dim(\operatorname{Ran}(\mathbf{A}))=n$), then $\mathbf{A}^*\mathbf{A}$ will be invertible,[‡] so we can write

$$\mathbf{x}=(\mathbf{A}^*\mathbf{A})^{-1}\mathbf{A}^*\mathbf{b}.$$

The formulation (16.2) arises sufficiently often to have its own name: the *normal equations.*

Note that $\mathbf{A}^*\mathbf{A}\in\mathbb{C}^{n\times n}$, which is often a small matrix. (Recall that $m\ge n$; in many applications, $m\gg n$.) Thus, $O(n^3)$ floating point operations are needed to solve the system $(\mathbf{A}^*\mathbf{A})\mathbf{x}=(\mathbf{A}^*\mathbf{b})$. However, it will be more costly to form the matrix $\mathbf{A}^*\mathbf{A}$: this matrix-matrix multiplication requires roughly mn^2 operations. (Why not $2mn^2$?) Moreover, this process is prone to magnify rounding errors, and hence is not favored by numerical analysts. Still, for 'well conditioned problems,' the normal equations approach can perform well. (We know about the condition number of a square matrix $\mathbf{A}$ with respect to the solution of $\mathbf{A}\mathbf{x}=\mathbf{b}$. We shall investigate the condition number of $\mathbf{A}$ with respect to the least squares problem on the fourth problem set.)

3.1.2. Solving least squares problems via QR factorization.

We next describe a technique for solving least squares problems that is more robust to rounding errors. Recall that any matrix $\mathbf{A}\in\mathbb{C}^{m\times n}$ can be written as $\mathbf{A}=\mathbf{Q}\mathbf{R}$ for a unitary matrix $\mathbf{Q}\in\mathbb{C}^{m\times m}$ and an upper triangular matrix $\mathbf{R}\in\mathbb{C}^{m\times n}$. Substitute this factorization into the least squares objective function:

$$\|\mathbf{b}-\mathbf{A}\mathbf{x}\|_2=\|\mathbf{b}-\mathbf{Q}\mathbf{R}\mathbf{x}\|_2.$$

If we could remove $\mathbf{Q}$ from the right-hand side, we would be left with a very simple upper triangular problem. Recall that the induced matrix 2-norm is invariant to unitary transformations. Since $\mathbf{Q}$ is unitary, $\mathbf{Q}^*\mathbf{Q}=\mathbf{I}$; Moreover, since $\mathbf{Q}$ is square, this means that $\mathbf{Q}^*$ must be the unique inverse of $\mathbf{Q}$: $\mathbf{Q}^{-1}=\mathbf{Q}^*$. Thus,

$$\begin{aligned}\|\mathbf{b}-\mathbf{Q}\mathbf{R}\mathbf{x}\|_2 &= \|\mathbf{Q}\mathbf{Q}^*\mathbf{b}-\mathbf{Q}\mathbf{R}\mathbf{x}\|_2\\ &= \|\mathbf{Q}(\mathbf{Q}^*\mathbf{b}-\mathbf{R}\mathbf{x})\|_2\\ &= \|\mathbf{Q}^*\mathbf{b}-\mathbf{R}\mathbf{x}\|_2.\end{aligned}$$

[‡]When $\mathbf{A}$ is not full rank, there are infinitely many choices for $\mathbf{x}$ that yield the same residual norm. The singular value decomposition, the subject of the next lecture, provides a mechanism for describing this set and selecting one distinguished $\mathbf{x}$ from the infinitely many candidates.

Now partition $\mathbf{Q}^*\mathbf{b} \in \mathbb{C}^m$ into two sections:

$$\mathbf{Q}^*\mathbf{b} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \end{bmatrix},$$

where $\mathbf{b}_1 \in \mathbb{C}^n$ and $\mathbf{b}_2 \in \mathbb{C}^{m-n}$. The rectangular upper triangular matrix $\mathbf{R}$ can be similarly partitioned:

$$\mathbf{R} = \begin{bmatrix} \mathbf{R}_1 \\ \mathbf{0} \end{bmatrix},$$

where $\mathbf{R}_1 \in \mathbb{C}^{n\times n}$, and the zero block has dimension $(m-n)$-by-n. Thus,

$$\mathbf{Q}^*\mathbf{b} - \mathbf{R}\mathbf{x} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \end{bmatrix} - \begin{bmatrix} \mathbf{R}_1 \\ \mathbf{0} \end{bmatrix}\mathbf{x} = \begin{bmatrix} \mathbf{b}_1 - \mathbf{R}_1\mathbf{x} \\ \mathbf{b}_2 \end{bmatrix}.$$

Just as in the derivation of the normal equations, we observe that an optimal choice for $\mathbf{x}$ will completely annihilate part of the residual, while leaving another component untouched. (In moving from $\|\mathbf{b} - \mathbf{Q}\mathbf{R}\mathbf{x}\|_2$ to the equivalent quantity $\|\mathbf{Q}^*\mathbf{b} - \mathbf{R}\mathbf{x}\|_2$, we have effectively transformed into a coordinate system in which $\mathrm{Ran}(\mathbf{A})$ has been mapped to vectors that are zero in their final $m-n$ components, while $\mathrm{Ker}(\mathbf{A}^*)$ now corresponds to vectors that are zero in their first n components, for full rank $\mathbf{A}$.) In particular,

$$\|\mathbf{b} - \mathbf{A}\mathbf{x}\|_2^2 = \left\| \begin{bmatrix} \mathbf{b}_1 - \mathbf{R}_1\mathbf{x} \\ \mathbf{b}_2 \end{bmatrix} \right\|_2^2 = \|\mathbf{b}_1 - \mathbf{R}_1\mathbf{x}\|_2^2 + \|\mathbf{b}_2\|_2^2.$$

Thus, the optimal choice for $\mathbf{x}$ is simply

$$\mathbf{x} = \mathbf{R}_1^{-1}\mathbf{b}_1 = \mathbf{R}_1^{-1}\mathbf{Q}_1^*\mathbf{b},$$

where $\mathbf{Q}_1 \in \mathbb{C}^{m\times n}$ consists of the first n columns of $\mathbf{Q}$.[§]

3.1.3. Example from polynomial approximation.

We revisit the problem posed at the beginning of this lecture: given the set of distinct points $\{x_0, x_1, \ldots, x_m\} \subset [a, b]$, find the polynomial $p \in \mathcal{P}_n$ that minimizes

$$\min_{p\in\mathcal{P}_n} \sum_{j=0}^{m} |f(x_j) - p(x_j)|^2$$

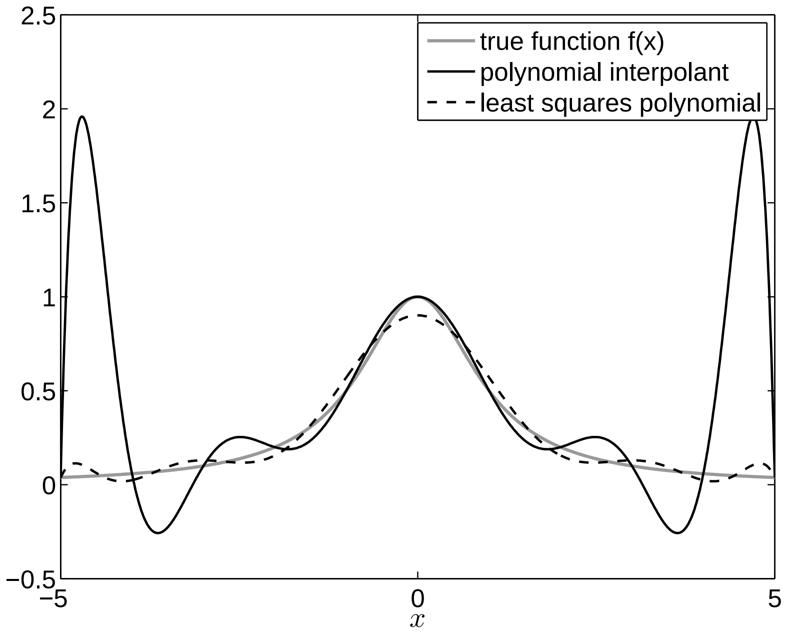
for some $f \in C[a, b]$. When $m > n$, this problem leads to an overdetermined linear system for the polynomial coefficients, one solved by MATLAB's `polyfit` command.

The coefficient matrix $\mathbf{A}$ is full-rank provided the points $\{x_j\}$ are distinct, and the resulting least squares problem can be readily solved using the techniques just described. To see this approach in action, we revisit the troublesome Runge function,

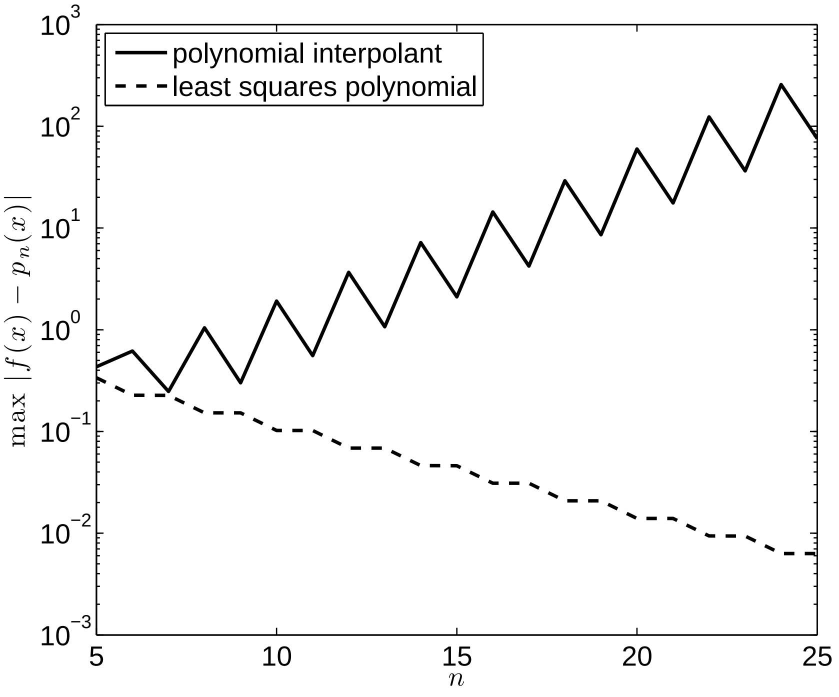
$$f(x) = \frac{1}{1+x^2}$$

for $x \in [-5, 5]$. Recall that, for this example, the polynomial interpolants at uniformly spaced points did not converge as the degree of the polynomial increased. The figure below compares the $n = 10$ interpolant at uniformly spaced points with the least squares polynomial of degree $n = 10$ that approximates f at 21 uniformly spaced points ($m = 20$).

[§]Note that $\mathbf{R}_1$ is invertible if and only if $\mathbf{A}$ is full rank. The QR factorization of a rank-deficient $\mathbf{A} \in \mathbb{C}^{m\times n}$ with $m \geq n$ must have a zero on the main diagonal. Can you explain why?



Now compare the overall error for the degree-n interpolant at uniformly spaced points with the error for the degree-n least squares polynomial based on uniformly spaced points with $m = 10n$. (This error is estimated by sampling $|f(x) - p_n(x)|$ at many points.) The least squares approach has a clear advantage here. Though simple and effective, the choice of $m = 10n$ approximation points is ad hoc. In upcoming lectures we study a more elegant approach that approximates f throughout the entire interval $[a, b]$ without the need to distinguish certain approximation points.



Lecture 17: The Singular Value Decomposition: Theory

3.2. The singular value decomposition.

Both the normal equation and QR approaches to solving the discrete linear least squares problem assume that the matrix $\mathbf{A} \in \mathbb{C}^{m\times n}$ has full column rank, i.e., its columns are linearly independent, implying that both $\mathbf{A}^*\mathbf{A}$ and $\mathbf{R}_1$ are invertible. What if this is not the case? The discrete least squares problem still makes sense, but we need a more robust computational approach to determine the solution. What if the columns of $\mathbf{A}$ are close to being linearly dependent? What does it even mean to be 'close' to linear dependence?

To answer these questions, we shall investigate one of the most important matrix factorizations, the *singular value decomposition* (SVD). This factorization writes a matrix as the product of a unitary matrix times a diagonal matrix times another unitary matrix. It is an incredibly useful tool for proving a variety of results in matrix theory, but it also has essential computational applications: from the SVD we immediately obtain bases for the four fundamental subspaces, $\mathrm{Ran}(\mathbf{A})$, $\mathrm{Ker}(\mathbf{A})$, $\mathrm{Ran}(\mathbf{A}^*)$, and $\mathrm{Ker}(\mathbf{A}^*)$. Furthermore, the SVD facilitates the robust solution of a variety of approximation problems, including not only least squares problems with rank-deficient $\mathbf{A}$, but also other low-rank matrix approximation problems that arise throughout engineering, statistics, the physical sciences, and social science.

There are several ways to derive the singular value decomposition. We shall constructively prove the SVD based on analysis of $\mathbf{A}^*\mathbf{A}$; Trefethen and Bau follow an alternative approach somewhat different from the one we describe; see their Theorem 4.1. Before beginning, we must recall some fundamental results from linear algebra.

3.2.1. Hermitian positive definite matrices.

Theorem (Spectral Theorem). Suppose $\mathbf{H} \in \mathbb{C}^{n\times n}$ is Hermitian. Then there exist n (not necessarily distinct) eigenvalues $\lambda_1, \ldots, \lambda_n$ and corresponding unit eigenvectors $\mathbf{v}_1, \ldots, \mathbf{v}_n$ such that

$$\mathbf{H}\mathbf{v}_j = \lambda_j \mathbf{v}_j$$

and the eigenvectors form an *orthonormal* basis for $\mathbb{C}^n$.

Theorem. All eigenvalues of a Hermitian matrix are real.

Proof. Let $(\lambda_j, \mathbf{v}_j)$ be an arbitrary eigenpair of the Hermitian matrix $\mathbf{H}$, so that $\mathbf{H}\mathbf{v}_j = \lambda_j\mathbf{v}_j$. Without loss of generality, we can assume that $\mathbf{v}_j$ is scaled so that $\|\mathbf{v}_j\|_2 = 1$. Thus

$$\lambda_j = \lambda_j(\mathbf{v}_j^*\mathbf{v}_j) = \mathbf{v}_j^*(\lambda_j\mathbf{v}_j) = \mathbf{v}_j^*(\mathbf{H}\mathbf{v}_j) = \mathbf{v}_j^*\mathbf{H}^*\mathbf{v}_j = (\mathbf{H}\mathbf{v}_j)^*\mathbf{v}_j = (\lambda_j\mathbf{v}_j)^*\mathbf{v}_j = \overline{\lambda}_j\mathbf{v}_j^*\mathbf{v}_j = \overline{\lambda}_j.$$

Thus $\lambda_j = \overline{\lambda}_j$, which is only possible if λ_j is real. ■

Definition. A Hermitian matrix $\mathbf{H} \in \mathbb{C}^{n\times n}$ is *positive definite* provided $\mathbf{x}^*\mathbf{H}\mathbf{x} > 0$ for all nonzero $\mathbf{x} \in \mathbb{C}^n$; if $\mathbf{x}^*\mathbf{H}\mathbf{x} \geq 0$ for all $\mathbf{x} \in \mathbb{C}^n$, we say $\mathbf{H}$ is *positive semidefinite.*

Theorem. A Hermitian positive semidefinite matrix has nonnegative real eigenvalues.

Proof. Let $(\lambda_j, \mathbf{v}_j)$ denote an eigenpair of the Hermitian positive semidefinite matrix $\mathbf{H} \in \mathbb{C}^{n\times n}$ with $\|\mathbf{v}_j\|_2^2 = \mathbf{v}_j^*\mathbf{v}_j = 1$. Since $\mathbf{H}$ is Hermitian, λ_j must be real. We conclude that

$$\lambda_j = \lambda_j\mathbf{v}_j^*\mathbf{v}_j = \mathbf{v}_j^*(\lambda_j\mathbf{v}_j) = \mathbf{v}_j^*\mathbf{H}\mathbf{v}_j \geq 0$$

since $\mathbf{H}$ is positive semidefinite. ■

3.2.2. Derivation of the singular value decomposition.

Suppose $\mathbf{A} \in \mathbb{C}^{m\times n}$ with $m \geq n$. The n-by-n matrix $\mathbf{A}^*\mathbf{A}$ is always Hermitian positive semidefinite. (Clearly $(\mathbf{A}^*\mathbf{A})^* = \mathbf{A}^*(\mathbf{A}^*)^* = \mathbf{A}^*\mathbf{A}$, so $\mathbf{A}^*\mathbf{A}$ is Hermitian. For any $\mathbf{x} \in \mathbb{C}^n$, note that $\mathbf{x}^*\mathbf{A}^*\mathbf{A}\mathbf{x} = (\mathbf{A}\mathbf{x})^*(\mathbf{A}\mathbf{x}) = \|\mathbf{A}\mathbf{x}\|_2^2 \geq 0$, so $\mathbf{A}^*\mathbf{A}$ is positive semidefinite.)

Step 1. As a consequence of results presented in §3.2.1, we can construct n eigenpairs $\{(\lambda_j, \mathbf{v}_j)\}_{j=1}^n$ of $\mathbf{A}^*\mathbf{A}$ with unit eigenvectors ($\mathbf{v}_j^*\mathbf{v}_j = 1$) that are orthogonal to one another ($\mathbf{v}_j^*\mathbf{v}_k = 0$ when $j \neq k$). We are free to pick any convenient indexing for these eigenpairs; it will be convenient to label them so that the eigenvalues are decreasing in magnitude, $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n \geq 0$.

Step 2. Define $\sigma_j = \|\mathbf{A}\mathbf{v}_j\|_2$.
Note that $\sigma_j^2 = \|\mathbf{A}\mathbf{v}_j\|_2^2 = \mathbf{v}_j^*\mathbf{A}^*\mathbf{A}\mathbf{v}_j = \lambda_j$. Since the eigenvalues $\lambda_1, \dots, \lambda_n$ are decreasing in magnitude, so are the σ_j values: $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n \geq 0$.

Step 3. Next, we will build a set of related orthonormal vectors in $\mathbb{C}^m$. Suppose we have already constructed such vectors $\mathbf{u}_1, \dots, \mathbf{u}_{j-1}$.
If $\sigma_j \neq 0$, then define $\mathbf{u}_j = \sigma_j^{-1}\mathbf{A}\mathbf{v}_j$, so that $\|\mathbf{u}_j\|_2 = \sigma_j^{-1}\|\mathbf{A}\mathbf{v}_j\|_2 = 1$.
If $\sigma_j = 0$, then pick $\mathbf{u}_j$ to be any unit vector such that

$$\mathbf{u}_j \in \operatorname{span}\{\mathbf{u}_1, \dots, \mathbf{u}_{j-1}\}^\perp;$$

i.e., ensure $\mathbf{u}_j^*\mathbf{u}_k = 0$ for all $k < j$.[†]

By construction, $\mathbf{u}_j^*\mathbf{u}_k = 0$ for $j \neq k$ if σ_j or σ_k is zero. If both σ_j and σ_k are nonzero, then

$$\mathbf{u}_j^*\mathbf{u}_k = \frac{1}{\sigma_j\sigma_k}(\mathbf{A}\mathbf{v}_j)^*(\mathbf{A}\mathbf{v}_k) = \frac{1}{\sigma_j\sigma_k}\mathbf{v}_j^*\mathbf{A}^*\mathbf{A}\mathbf{v}_k = \frac{\lambda_k}{\sigma_j\sigma_k}\mathbf{v}_j^*\mathbf{v}_k,$$

where we used the fact that $\mathbf{v}_j$ is an eigenvector of $\mathbf{A}^*\mathbf{A}$. Now if $j \neq k$, then $\mathbf{v}_j^*\mathbf{v}_k = 0$, and hence $\mathbf{u}_j^*\mathbf{u}_k = 0$. On the other hand, $j = k$ implies that $\mathbf{v}_j^*\mathbf{v}_k = 1$, so $\mathbf{u}_j^*\mathbf{u}_k = \lambda_j/\sigma_j^2 = 1$.

In conclusion, we have constructed a set of orthonormal vectors $\{\mathbf{u}_j\}_{j=1}^n$ with $\mathbf{u}_j \in \mathbb{C}^m$.

Step 4. For all $j = 1, \dots, n$,

$$\mathbf{A}\mathbf{v}_j = \sigma_j\mathbf{u}_j,$$

regardless of whether $\sigma_j = 0$ or not. We can stack these n vector equations as columns of a single matrix equation,

$$\left[\begin{array}{cccc} | & | & & | \\ \mathbf{A}\mathbf{v}_1 & \mathbf{A}\mathbf{v}_2 & \cdots & \mathbf{A}\mathbf{v}_n \\ | & | & & | \end{array}\right] = \left[\begin{array}{cccc} | & | & & | \\ \sigma_1\mathbf{u}_1 & \sigma_2\mathbf{u}_2 & \cdots & \sigma_n\mathbf{u}_n \\ | & | & & | \end{array}\right].$$

Note that both matrices in this equation can be factored into the product of simpler matrices:

$$\mathbf{A}\left[\begin{array}{cccc} | & | & & | \\ \mathbf{v}_1 & \mathbf{v}_2 & \cdots & \mathbf{v}_n \\ | & | & & | \end{array}\right] = \left[\begin{array}{cccc} | & | & & | \\ \mathbf{u}_1 & \mathbf{u}_2 & \cdots & \mathbf{u}_n \\ | & | & & | \end{array}\right]\left[\begin{array}{cccc} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_n \end{array}\right].$$

[†]Note that $\sigma_j = 0$ implies that $\lambda_j = 0$, and so $\mathbf{A}^*\mathbf{A}$ has a zero eigenvalue; i.e., this matrix is singular. Recall from the last lecture that this case can only occur when $\mathbf{A}$ is rank-deficient: $\dim(\operatorname{Ran}(\mathbf{A})) < n$.

Denote these matrices as $\mathbf{AV} = \widehat{\mathbf{U}}\widehat{\mathbf{\Sigma}}$, where $\mathbf{A} \in \mathbb{C}^{m\times n}$, $\mathbf{V} \in \mathbb{C}^{n\times n}$, $\widehat{\mathbf{U}} \in \mathbb{C}^{m\times n}$, and $\widehat{\mathbf{\Sigma}} \in \mathbb{C}^{n\times n}$.

The (j,k) entry of $\mathbf{V}^*\mathbf{V}$ is simply $\mathbf{v}_j^*\mathbf{v}_k$, and so $\mathbf{V}^*\mathbf{V} = \mathbf{I}$. Since $\mathbf{V}$ is a square matrix, we have just proved that it is unitary. Hence, $\mathbf{VV}^* = \mathbf{I}$ as well, and we conclude that

$$\mathbf{A} = \widehat{\mathbf{U}}\widehat{\mathbf{\Sigma}}\mathbf{V}^*.$$

This matrix factorization is known as the *reduced singular value decomposition.* It can be obtained via the MATLAB command

```
[Uhat, Sighat, V] = svd(A,0);
```

While the matrix $\widehat{\mathbf{U}}$ has orthonormal columns, *it is not a unitary matrix.* In particular, we have $\widehat{\mathbf{U}}^*\widehat{\mathbf{U}} = \mathbf{I} \in \mathbb{C}^{n\times n}$, but

$$\widehat{\mathbf{U}}\widehat{\mathbf{U}}^* \in \mathbb{C}^{m\times m}$$

cannot be the identity unless $m = n$. (To see this, note that $\widehat{\mathbf{U}}\widehat{\mathbf{U}}^*$ is an orthogonal projection onto $\mathrm{Ran}(\widehat{\mathbf{U}}) = \mathrm{span}\{\mathbf{u}_1, \ldots, \mathbf{u}_n\}$. Since $\dim(\mathrm{Ran}(\widehat{\mathbf{U}})) = n$, this projection cannot equal the m-by-m identity matrix when $m > n$.)

Though $\widehat{\mathbf{U}}$ is not unitary, we might call it *subunitary.*[‡] We can construct $m-n$ additional column vectors to append to $\widehat{\mathbf{U}}$ to make it unitary. Here is the recipe: For $j = n+1, \ldots, m$, pick

$$\mathbf{u}_j \in \mathrm{span}\{\mathbf{u}_1, \ldots, \mathbf{u}_{j-1}\}^\perp$$

with $\mathbf{u}_j^*\mathbf{u}_j = 1$. Then define

$$\mathbf{U} = \left[\begin{array}{cccc} | & | & & | \\ \mathbf{u}_1 & \mathbf{u}_2 & \cdots & \mathbf{u}_m \\ | & | & & | \end{array}\right].$$

It is simple to confirm that $\mathbf{U}^*\mathbf{U} = \mathbf{UU}^* = \mathbf{I} \in \mathbb{C}^{m\times m}$, so $\mathbf{U}$ is unitary.

We wish to replace the $\widehat{\mathbf{U}}$ in the reduced SVD with the unitary matrix $\mathbf{U}$. To do so, we also need to replace $\widehat{\mathbf{\Sigma}}$ by some $\mathbf{\Sigma}$ in such a way that $\widehat{\mathbf{U}}\widehat{\mathbf{\Sigma}} = \mathbf{U\Sigma}$. The simplest approach is to obtain $\mathbf{\Sigma}$ by appending zeros to the end of $\widehat{\mathbf{\Sigma}}$, thus ensuring there is no contribution when the new entries of $\mathbf{U}$ multiply against the new entries of $\mathbf{\Sigma}$:

$$\mathbf{\Sigma} = \left[\begin{array}{c} \widehat{\mathbf{\Sigma}} \\ \mathbf{0} \end{array}\right] \in \mathbb{C}^{m\times n}.$$

Finally, we are prepared to state our main result, the *full singular value decomposition.*

Theorem (Singular value decomposition). Any matrix $\mathbf{A} \in \mathbb{C}^{m\times n}$ can be written in the form

$$\mathbf{A} = \mathbf{U\Sigma V}^*,$$

where $\mathbf{U} \in \mathbb{C}^{m\times m}$ and $\mathbf{V} \in \mathbb{C}^{n\times n}$ are unitary matrices and $\mathbf{\Sigma} \in \mathbb{C}^{m\times n}$ is zero everywhere except for entries on the main diagonal, where the (j,j) entry is σ_j for $j = 1, \ldots, \min(m,n)$, and

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_{\min(m,n)} \geq 0.$$

[‡]There is no universally accepted term for such a matrix; Gilbert Strang suggests the descriptive term *subunitary.*

We have only proved this result for $m \geq n$. The proof for $m < n$ is obtained by applying the same arguments above to $\mathbf{A}^*$ in place of $\mathbf{A}$.

The full SVD is obtained via the MATLAB command

```
[U,S,V] = svd(A).
```

Lecture 18: The SVD: Examples, Norms, Fundamental Subspaces, Compression

3.2.3. Example of the singular value decomposition.

The standard algorithm for computing the singular value decomposition differs a bit from the algorithm described in the last lecture. We know from our experiences with the normal equations for least squares problems that significant errors can be introduced when $\mathbf{A}^*\mathbf{A}$ is constructed. For practical SVD computations, one can sidestep this by using Householder transformations to create unitary matrices $\mathbf{U}$ and $\mathbf{V}$ such that $\mathbf{B} := \mathbf{UAV}^*$ is *bidiagonal*, i.e., $b_{jk} = 0$ unless $j = k$ or $j-1 = k$ One then applies specialized eigenvalue algorithms for computing the SVD of a bidiagonal matrix; see Trefethen & Bau (Lecture 31) for details.

While this approach has numerical advantages over the method used in our constructive proof of the SVD, it is still instructive to follow through that construction for a simple matrix, say

$$\mathbf{A} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}.$$

Step 1. First, form $\mathbf{A}^*\mathbf{A}$:

$$\mathbf{A}^*\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$$

and compute its eigenvalues and (normalized) eigenvectors:

$$\lambda_1 = 3, \quad \mathbf{v}_1 = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ 1 \end{bmatrix}, \qquad \lambda_2 = 1, \quad \mathbf{v}_2 = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ -1 \end{bmatrix}.$$

Step 2. Set

$$\begin{aligned} \sigma_1 &= \|\mathbf{Av}_1\|_2 = \sqrt{\lambda_1} = \sqrt{3}; \\ \sigma_2 &= \|\mathbf{Av}_2\|_2 = \sqrt{\lambda_2} = 1. \end{aligned}$$

Step 3. Since $\sigma_1, \sigma_2 \neq 0$, we can immediately form $\mathbf{u}_1$ and $\mathbf{u}_2$:

$$\mathbf{u}_1 = \frac{1}{\sigma_1}\mathbf{Av}_1 = \frac{1}{\sqrt{6}}\begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix}, \qquad \mathbf{u}_2 = \frac{1}{\sigma_2}\mathbf{Av}_2 = \frac{1}{\sqrt{2}}\begin{bmatrix} -1 \\ 1 \\ 0 \end{bmatrix}.$$

The $1/\sigma_j$ scaling ensures that both $\mathbf{u}_1$ and $\mathbf{u}_2$ are unit vectors. We can verify that they are orthogonal:

$$\mathbf{u}_1^*\mathbf{u}_2 = \frac{1}{\sqrt{12}}\,[1 \;\; 1 \;\; 2]\begin{bmatrix} -1 \\ 1 \\ 0 \end{bmatrix} = 0.$$

Step 4. At this point, we have all the ingredients to build the reduced singular value decomposition:

$$\mathbf{A} = \widehat{\mathbf{U}}\widehat{\mathbf{\Sigma}}\mathbf{V}^* = \begin{bmatrix} 1/\sqrt{6} & -1/\sqrt{2} \\ 1/\sqrt{6} & 1/\sqrt{2} \\ 2/\sqrt{6} & 0 \end{bmatrix}\begin{bmatrix} \sqrt{3} & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{bmatrix}.$$

The only additional information required to build the full SVD is the unit vector $\mathbf{u}_3$ that is orthogonal to $\mathbf{u}_1$ and $\mathbf{u}_2$. One can find such a vector by inspection:

$$\mathbf{u}_3 = \frac{1}{\sqrt{3}}\begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}.$$

If you are naturally able to eyeball this orthogonal vector, there are any number of mechanical ways to compute $\mathbf{u}_3$, e.g., by finding a vector $\mathbf{u}_3 = [\alpha, \beta, \gamma]^T$ that satisfies the orthogonality conditions $\mathbf{u}_1^*\mathbf{u}_3 = \mathbf{u}_2^*\mathbf{u}_3 = 0$ and normalization $\mathbf{u}_3^*\mathbf{u}_3 = 1$, or using the Gram–Schmidt process. A related method is just to read $\mathbf{u}_3$ off as the third column of the $\mathbf{Q}$ factor in the full QR decomposition of $[\mathbf{u}_1\ \mathbf{u}_2]$. (Why is this so? Recall that the first n columns of the $\mathbf{Q}$ factor form a basis for the range of the factored matrix; the remaining $m-n$ columns are unit vectors that must be orthogonal to those previous columns, since $\mathbf{Q}$ is unitary.) For example:

```
>> u1 = [1;1;2]/sqrt(6);   u2 = [-1;1;0]/sqrt(2);
>> [Q,R] = qr([u1 u2])
Q =
   -0.4082    0.7071   -0.5774
   -0.4082   -0.7071   -0.5774
   -0.8165   -0.0000    0.5774
R =
   -1.0000         0
         0   -1.0000
         0         0
```

Note that the third vector in the `Q` matrix is simply $-\mathbf{u}_3$. (We could just as well replace $\mathbf{u}_3$ by $-\mathbf{u}_3$ without changing the SVD. Why?)

In conclusion, a full SVD of $\mathbf{A}$ is:

$$\mathbf{A} = \mathbf{U\Sigma V}^* = \begin{bmatrix} 1/\sqrt{6} & -1/\sqrt{2} & 1/\sqrt{3} \\ 1/\sqrt{6} & 1/\sqrt{2} & 1/\sqrt{3} \\ 2/\sqrt{6} & 0 & -1/\sqrt{3} \end{bmatrix} \begin{bmatrix} \sqrt{3} & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{bmatrix}.$$

3.2.4. Singular values and the matrix 2-norm.

In Lecture 2 we defined the induced matrix 2-norm

$$\|\mathbf{A}\|_2 = \max_{\|\mathbf{x}\|_2=1} \|\mathbf{Ax}\|_2,$$

but did not provide a simple formula for this norm in terms of the entries of $\mathbf{A}$, as we did for the induced matrix 1- and ∞-norms. With the SVD at hand, we can now derive such a formula.

Recall that the vector 2-norm (and hence the matrix 2-norm) is invariant to premultiplication by a unitary matrix, as proved in Lecture 2. Let $\mathbf{A} = \mathbf{U\Sigma V}^*$ be a singular value decomposition of $\mathbf{A}$. Thus

$$\|\mathbf{A}\|_2 = \|\mathbf{U\Sigma V}^*\|_2 = \|\mathbf{\Sigma V}^*\|_2.$$

The matrix 2-norm is also immune to a unitary matrix on the right:

$$\|\mathbf{\Sigma V}^*\|_2 = \max_{\|\mathbf{x}\|_2=1} \|\mathbf{\Sigma V}^*\mathbf{x}\|_2 = \max_{\|\mathbf{y}\|_2=1} \|\mathbf{\Sigma y}\|_2 = \|\mathbf{\Sigma}\|_2,$$

where we have set $\mathbf{y} = \mathbf{V}^*\mathbf{x}$ and noted that $\|\mathbf{y}\|_2 = \|\mathbf{V}^*\mathbf{x}\|_2 = \|\mathbf{x}\|_2$ since $\mathbf{V}^*$ is a unitary matrix. Let $p = \min\{m, n\}$. Then

$$\|\mathbf{\Sigma y}\|_2^2 = \sum_{j=1}^{p} \sigma_j^2 y_j^2,$$

which is maximized over $\|\mathbf{y}\|_2 = 1$ by $\mathbf{y} = [1, 0, \ldots, 0]^T$, giving

$$\|\mathbf{A}\|_2 = \|\mathbf{\Sigma}\|_2 = \sigma_1.$$

Thus the matrix 2-norm is simply the first singular value. The 2-norm is often the 'natural' norm to use in applications, but if the matrix $\mathbf{A}$ is large, its computation is costly ($O(mn^2)$ floating point operations). For quick estimates that only require $O(mn)$ operations and are accurate to a factor of $\sqrt{m}$ or $\sqrt{n}$, use the matrix 1- or ∞-norms.

The SVD has many other important uses. For example, if $\mathbf{A} \in \mathbb{C}^{n\times n}$ is invertible, we have $\mathbf{A}^{-1} = \mathbf{V\Sigma}^{-1}\mathbf{U}^*$, and so $\|\mathbf{A}^{-1}\|_2 = 1/\sigma_n$. This illustrates that a square matrix is singular if and only if $\sigma_n = 0$. We shall explore this in more depth later when we use the SVD to construct low-rank approximations to $\mathbf{A}$.

Like the 2-norm, the Frobenius norm,

$$\|\mathbf{A}\|_F = \Big(\sum_{j=1}^{m}\sum_{k=1}^{n} |a_{jk}|^2\Big)^{1/2}$$

is unitarily invariant. What are $\|\mathbf{A}\|_F$ and $\|\mathbf{A}^{-1}\|_F$ in terms of the singular values of $\mathbf{A}$?

3.2.5. The SVD and the four fundamental subspaces.

For simplicity, assume $m \ge n$. Then $\mathbf{A} = \mathbf{U\Sigma V}^*$ can be written as the linear combination of m-by-n outer product matrices:

$$\mathbf{A} = \mathbf{U\Sigma V}^* = \begin{bmatrix}\sigma_1\mathbf{u}_1 & \sigma_2\mathbf{u}_2 & \cdots & \sigma_n\mathbf{u}_n\end{bmatrix}\begin{bmatrix}\mathbf{v}_1^* \\ \mathbf{v}_2^* \\ \vdots \\ \mathbf{v}_n^*\end{bmatrix} = \sum_{j=1}^{n} \sigma_j \mathbf{u}_j \mathbf{v}_j^*.$$

Hence for any $\mathbf{x} \in \mathbb{C}^n$,

$$\mathbf{Ax} = [\sum_{j=1}^{n} \sigma_j \mathbf{u}_j \mathbf{v}_j^*]\mathbf{x} = \sum_{j=1}^{n} (\sigma_j \mathbf{v}_j^*\mathbf{x})\mathbf{u}_j,$$

since $\mathbf{v}_j^*\mathbf{x}$ is just a scalar. We see that $\mathbf{Ax}$ is a linear combination of the left singular vectors $\{\mathbf{u}_j\}$, and have nearly uncovered a basis for $\mathrm{Ran}(\mathbf{A})$. The only catch is that $\mathbf{u}_j$ will not contribute to the above linear combination if $\sigma_j = 0$. If all the singular values are nonzero, set $r = n$; otherwise, define r such that $\sigma_r \ne 0$ but $\sigma_{r+1} = 0$. Then we have

$$\mathbf{Ax} = \sum_{j=1}^{r} (\sigma_j \mathbf{v}_j^*\mathbf{x})\mathbf{u}_j,$$

and so

$$\mathrm{Ran}(\mathbf{A}) = \Big\{\sum_{j=1}^{r} \gamma_j \mathbf{u}_j : \gamma_1, \ldots, \gamma_r \in \mathbb{C}\Big\}.$$

Since the vectors $\mathbf{u}_1, \ldots, \mathbf{u}_r$ are orthogonal by construction, they are linearly independent, and thus give a basis for $\mathrm{Ran}(\mathbf{A})$:

$$\mathrm{Ran}(\mathbf{A}) = \mathrm{span}\{\mathbf{u}_1, \ldots, \mathbf{u}_r\}.$$

Moreover, r is the dimension of $\mathrm{Ran}(\mathbf{A})$, i.e., $\mathrm{rank}(\mathbf{A}) = r$.

Immediately we have a basis for $\mathrm{Ker}(\mathbf{A}^*)$, too: The Fundamental Theorem of Linear Algebra guarantees that $\mathrm{Ran}(\mathbf{A}) \oplus \mathrm{Ker}(\mathbf{A}^*) = \mathbb{C}^m$ and $\mathrm{Ran}(\mathbf{A}) \perp \mathrm{Ker}(\mathbf{A}^*)$. Together these facts, with the orthogonality of the left singular vectors, gives

$$\mathrm{Ker}(\mathbf{A}^*) = \mathrm{span}\{\mathbf{u}_{r+1}, \ldots, \mathbf{u}_m\}.$$

Applying the same arguments to $\mathbf{A}^*$ yields bases for the two remaining fundamental subspaces:

$$\mathrm{Ran}(\mathbf{A}^*) = \mathrm{span}\{\mathbf{v}_1, \ldots, \mathbf{v}_r\}, \qquad \mathrm{Ker}(\mathbf{A}) = \mathrm{span}\{\mathbf{v}_{r+1}, \ldots, \mathbf{v}_n\},$$

where $\mathrm{Ran}(\mathbf{A}^*) \oplus \mathrm{Ker}(\mathbf{A}) = \mathbb{C}^n$ and $\mathrm{Ran}(\mathbf{A}^*) \perp \mathrm{Ker}(\mathbf{A})$. Hence, the SVD is a beautiful tool for revealing the fundamental subspaces.

3.2.6. Low-rank matrix approximation.

One of the key applications of the singular value decomposition is the construction of *low-rank approximations* to a matrix. Recall that the SVD of $\mathbf{A}$ can be written as

$$\mathbf{A} = \sum_{j=1}^{r} \sigma_j \mathbf{u}_j \mathbf{v}_j^*,$$

where $r = \mathrm{rank}(\mathbf{A})$. We can approximate $\mathbf{A}$ by taking only a partial sum here:

$$\mathbf{A}_k = \sum_{j=1}^{k} \sigma_j \mathbf{u}_j \mathbf{v}_j^*$$

for $k \le r$. The linear independence of $\{\mathbf{u}_1, \ldots, \mathbf{u}_k\}$ guarantees that $\mathrm{rank}(\mathbf{A}_k) = k$. But how well does this partial sum approximate $\mathbf{A}$? This question is answered by the following result, due variously to Schmidt, Mirsky, Eckart, and Young, that has wide-ranging consequences in applications.

Theorem. For all $1 \le k < \mathrm{rank}(\mathbf{A})$,

$$\min_{\mathrm{rank}(\mathbf{X})=k} \|\mathbf{A} - \mathbf{X}\| = \sigma_{k+1},$$

with the minimum attained by

$$\mathbf{A}_k = \sum_{j=1}^{k} \sigma_j \mathbf{u}_j \mathbf{v}_j^*.$$

Proof. [See, e.g., J. W. Demmel, *Applied Numerical Linear Algebra*, §3.2.3] Let $\mathbf{X} \in \mathbb{C}^{m\times n}$ be any rank-k matrix. The Fundamental Theorem of Linear Algebra guarantees that $\mathbb{C}^n = \mathrm{Ran}(\mathbf{X}^*) \oplus \mathrm{Ker}(\mathbf{X})$. Since $\mathrm{rank}(\mathbf{X}^*) = \mathrm{rank}(\mathbf{X}) = k$, we conclude that $\dim(\mathrm{Ker}(\mathbf{X})) = n - k$.

From the singular value decomposition

$$\mathbf{A} = \sum_{j=1}^{r} \sigma_j \mathbf{u}_j \mathbf{v}_j^*,$$

extract the vectors $\{\mathbf{v}_1, \ldots, \mathbf{v}_{k+1}\}$, which form a basis for a $k+1$ dimensional subspace of $\mathbb{C}^n$. Since $\mathrm{Ker}(\mathbf{X}) \subseteq \mathbb{C}^n$ has dimension $n-k$, it must be that the intersection

$$\mathrm{Ker}(\mathbf{X}) \cap \mathrm{span}\{\mathbf{v}_1, \ldots, \mathbf{v}_{k+1}\}$$

is nontrivial, i.e., is at least one-dimensional.[†] Let $\mathbf{z}$ be some unit vector in that intersection:

$$\mathbf{z} \in \mathrm{Ker}(\mathbf{X}) \cap \mathrm{span}\{\mathbf{v}_1, \ldots, \mathbf{v}_{k+1}\}, \qquad \|\mathbf{z}\|_2 = 1.$$

Expand $\mathbf{z} = \gamma_1\mathbf{v}_1 + \cdots + \gamma_{k+1}\mathbf{v}_{k+1}$, so that $\|\mathbf{z}\|_2 = 1$ implies

$$1 = \mathbf{z}^*\mathbf{z} = \Big(\sum_{j=1}^{k+1} \gamma_j \mathbf{v}_j\Big)^* \Big(\sum_{j=1}^{k+1} \gamma_j \mathbf{v}_j\Big) = \sum_{j=1}^{k+1} |\gamma_j|^2.$$

Since $\mathbf{z} \in \mathrm{Ker}(\mathbf{X})$, we have

$$\|\mathbf{A}-\mathbf{X}\|_2 \;\geq\; \|(\mathbf{A}-\mathbf{X})\mathbf{z}\|_2 \;=\; \|\mathbf{A}\mathbf{z}\|_2 \;=\; \Big\|\sum_{j=1}^{k+1} \sigma_j \mathbf{u}_j \mathbf{v}_j^* \mathbf{z}\Big\|_2 \;=\; \Big\|\sum_{j=1}^{k+1} \sigma_j \gamma_j \mathbf{u}_j\Big\|_2.$$

Since $\sigma_{k+1} \leq \sigma_k \leq \cdots \leq \sigma_1$ and the $\mathbf{u}_j$ vectors are orthogonal,

$$\Big\|\sum_{j=1}^{k+1} \sigma_j \gamma_j \mathbf{u}_j\Big\|_2 \geq \sigma_{k+1} \Big\|\sum_{j=1}^{k+1} \gamma_j \mathbf{u}_j\Big\|_2.$$

But notice that

$$\Big\|\sum_{j=1}^{k+1} \gamma_j \mathbf{u}_j\Big\|_2^2 = \Big(\sum_{j=1}^{k+1} \gamma_j \mathbf{u}_j\Big)^* \Big(\sum_{j=1}^{k+1} \gamma_j \mathbf{u}_j\Big) = \sum_{j=1}^{k+1} |\gamma_j|^2 = 1,$$

where the last equality was derived above from the fact that $\|\mathbf{z}\|_2 = 1$. In conclusion,

$$\|\mathbf{A}-\mathbf{X}\|_2 \;\geq\; \sigma_{k+1} \Big\|\sum_{j=1}^{k+1} \gamma_j \mathbf{u}_j\Big\|_2 \;=\; \sigma_{k+1}$$

for any rank-k matrix $\mathbf{X}$.

All that remains is to show that this bound is attained by $\mathbf{A}_k$, the kth partial sum of the singular value decomposition. We have

$$\mathbf{A} - \mathbf{A}_k \;=\; \sum_{j=1}^{r} \sigma_j \mathbf{u}_j \mathbf{v}_j - \sum_{j=1}^{k} \sigma_j \mathbf{u}_j \mathbf{v}_j \;=\; \sum_{j=k+1}^{r} \sigma_j \mathbf{u}_j \mathbf{v}_j.$$

But this last expression is essentially a singular value decomposition for $\mathbf{A}-\mathbf{X}$, with largest singular value σ_{k+1}. Hence $\|\mathbf{A}-\mathbf{A}_k\|_2 = \sigma_{k+1}$ as claimed, and we see that $\mathbf{A}_k$ is a best rank-k approximation to $\mathbf{A}$ in the two-norm. ■

Notice that we do not claim that the best rank-k approximation given in the theorem is *unique*. Can you think of how you might find other rank-k matrices $\widehat{\mathbf{A}}_k$ such that $\|\mathbf{A}-\widehat{\mathbf{A}}_k\|_2 = \|\mathbf{A}-\mathbf{A}_k\|_2$?

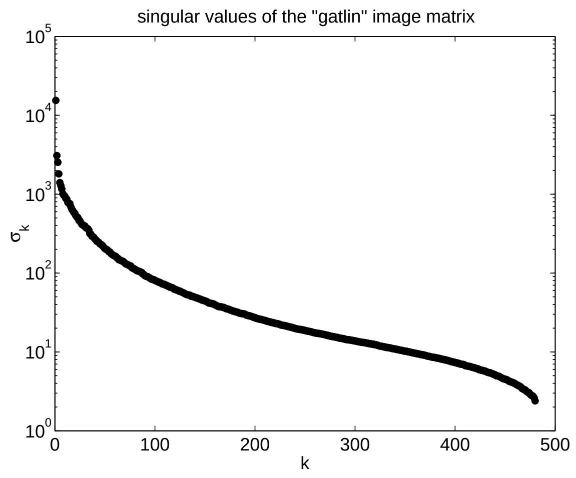
[†]Otherwise, $\mathrm{Ker}(\mathbf{X}) \oplus \mathrm{span}\{\mathbf{v}_1, \ldots, \mathbf{v}_{k+1}\}$ would be an $n+1$ dimensional subspace of $\mathbb{C}^n$: impossible!

Application: image compression. As an illustration of the utility of low-rank matrix approximations, consider the compression of digital images. On a computer, an image is simply a matrix denoting pixel colors. For example, a grayscale image can be represented as a matrix whose entries are integers between 0 and 255 (for 256 shades of gray), denoting the shade of each pixel. Typically, such matrices can be well-approximated by low-rank matrices. Instead of storing the mn entries of the matrix $\mathbf{A}$, one need only store the $k(m+n)+k$ numbers that make up the various σ_j, $\mathbf{u}_j$, and $\mathbf{v}_j$ values in the sum

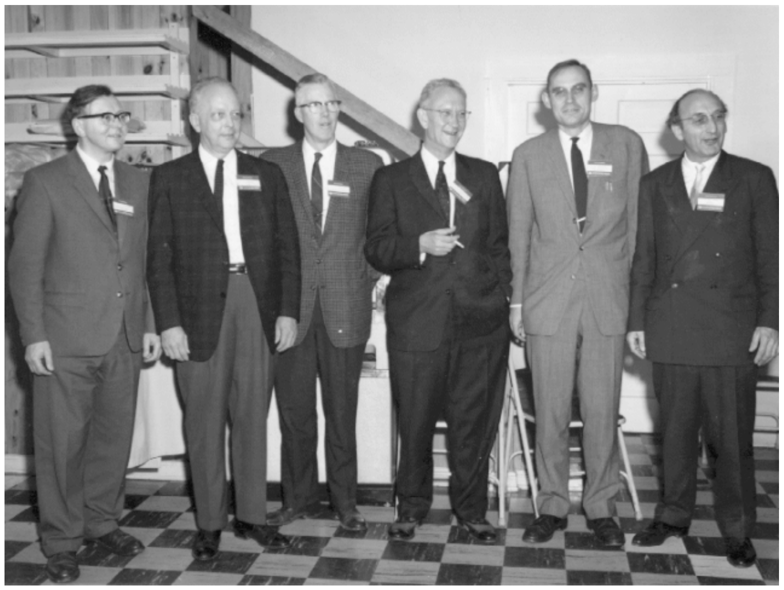
$$\mathbf{A}_k = \sum_{j=1}^{k} \sigma_j \mathbf{u}_j \mathbf{v}_j^*.$$

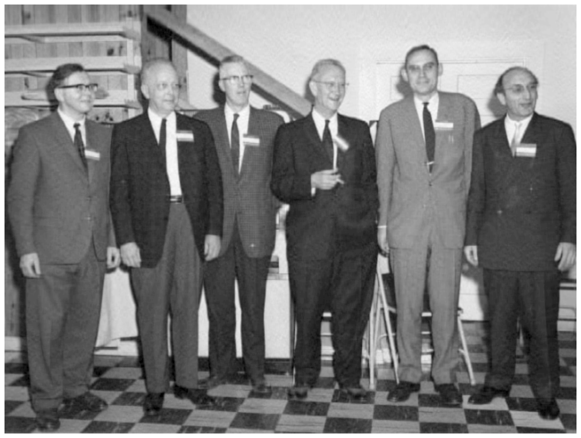
When $k \ll \min(m,n)$, this can make for a significant improvement (though modern image compression protocols use more sophisticated approaches).

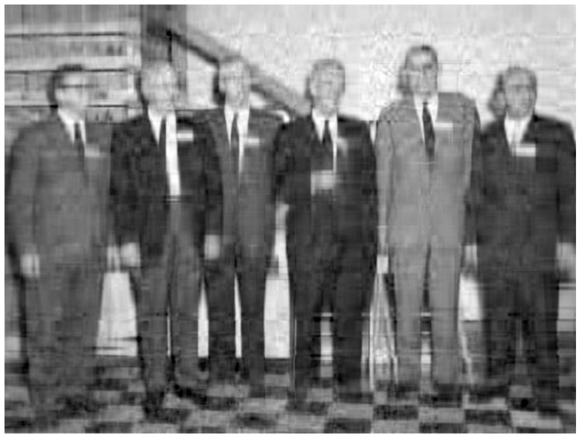
Next we show the singular values for one image matrix, a photograph of many of the patriarchs of modern matrix computations taken at the 1964 Gatlinburg Conference on Numerical Algebra: from left to right, we have Jim Wilkinson, Wallace Givens, George Forsythe, Alston Householder, Peter Henrici, and Fritz Bauer. The matrix is of dimension 480-by-640, reflecting the fact that the picture is wider than it is tall. Though the singular values are large, $\sigma_{480} > 1$, there is a relative difference of four orders of magnitude between the smallest and largest singular value. If all the singular values were roughly the same, we would not expect accurate low-rank approximations.)

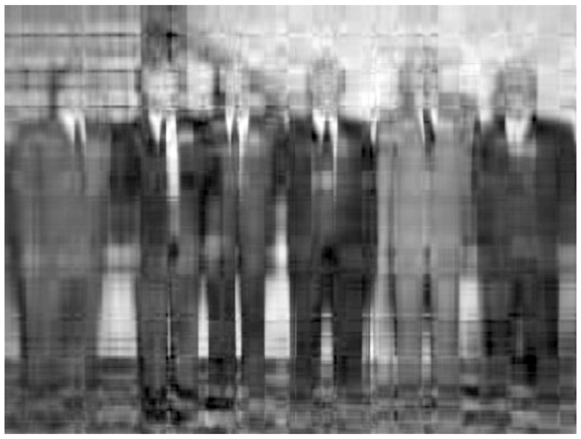


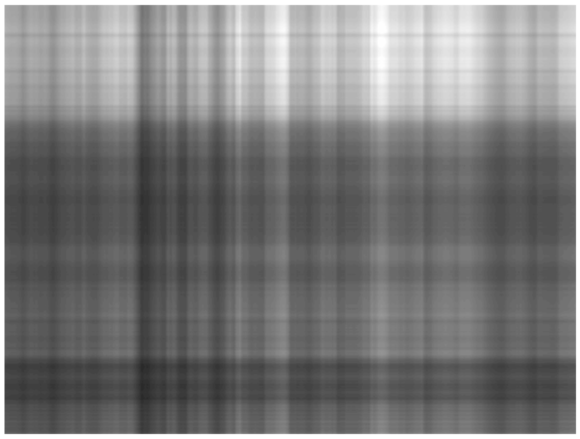
true image (rank 480)




best rank−100 approximation


best rank−25 approximation


best rank−10 approximation


best rank−1 approximation

Below is a sample of that MATLAB code that generated these images, so you can experiment with this example further if you like. For further examples and more theory, see J. W. Demmel, *Applied Numerical Linear Algebra*, §3.2.3.

```
load gatlin                         % load the "gatlin" image data, built-in to MATLAB
[U,S,V] = svd(X);                   % "gatlin" stores the image as the variable "X"

figure(1),clf                       % plot the singular values
semilogy(diag(S),'b.','markersize',20)
set(gca,'fontsize',16)
title('singular values of the "gatlin" image matrix')
xlabel('k'), ylabel('\sigma_k')

figure(2),clf                       % plot the original image
image(X), colormap(map)             % image: MATLAB command to display a matrix as image
axis equal, axis off
title('true image (rank 480)','fontsize',16)

figure(3),clf                       % plot the optimal rank-k approximation
k = 100;
Xk = U(:,1:k)*S(1:k,1:k)*V(:,1:k)';
image(Xk), colormap(map)
axis equal, axis off
title(sprintf('best rank-%d approximation',k),'fontsize',16)
```

Lecture 19: Continuous Least Squares Approximation

3.3. Continuous least squares approximation.

We began §3.1 with the problem of approximating some $f \in C[a,b]$ with a polynomial $p \in \mathcal{P}_n$ at the discrete points $x_0, x_1, \ldots, x_m$ for some $m \geq n$. This example motivated our study of discrete least squares problems (a subject with many other diverse applications), but the choice of the m points is somewhat arbitrary. Suppose we simply wish for the approximating polynomial to represent f throughout all of $[a,b]$. What value should m take? How should one pick the points $\{x_k\}$? Suppose we uniformly distribute these approximation points over $[a,b]$: set $h_m := (b-a)/m$ and let $x_k = a + kh_m$. The least squares error formula, 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, in the limit of infinitely many uniformly spaced approximation points, we are actually minimizing an integral, rather than a sum. In this lecture, we will see how to pose such problems as a matrix problem of dimension $(n+1)$-by-$(n+1)$, instead of a discrete least squares problem with matrix of dimension '∞-by-$(n+1)$'.

3.3.1. Inner products for function spaces.

To facilitate the development of continuous 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. The *inner product* of the functions $f, g \in C[a,b]$ is given by

$$\langle f, g\rangle = \int_a^b f(x)g(x)\,\mathrm{d}x.$$

This 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 $f \in C[a,b]$.

Just as the vector 2-norm naturally follows from the vector inner product ($\|\mathbf{x}\|_2 = \sqrt{\mathbf{x}^*\mathbf{x}}$), so we have

$$\|f\|_{L^2} := \langle f, f\rangle^{1/2} = \Big(\int_a^b f(x)^2\,\mathrm{d}x\Big)^{1/2}.$$

Here 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

[†]If we wanted to consider complex-valued functions f and g, the inner product would be generalized to $\langle f, g\rangle = \int_a^b f(x)\overline{g(x)}\,\mathrm{d}x$, giving $\langle f, g\rangle = \overline{\langle g, f\rangle}$.

$C[a,b]$ 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. (The Lebesgue theory gives a more robust definition of the integral than the conventional Riemann approach; for details, consult MATH 425.)

3.3.2. Least squares minimization via calculus. Given some $f \in C[a,b]$, the basic L^2 approximation problem seeks the polynomial $p \in \mathcal{P}_n$ that minimizes the error $f-p$ in the L^2 norm. In symbols:

$$\min_{p\in\mathcal{P}_n} \|f-p\|_{L^2}.$$

We shall denote the polynomial that attains this minimum by p_*.

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 $p(x) = c_0 + c_1 x$. Note that $\|f-p\|_{L^2}$ will be minimized by the same polynomial as $\|f-p\|_{L^2}^2$. Thus for any given $p \in \mathcal{P}_1$, the error function is given by

$$\begin{aligned} E(c_0,c_1) := \|f(x)-(c_0+c_1x)\|_{L^2}^2 &= \int_a^b (f(x)-c_0-c_1x)^2\,\mathrm{d}x \\ &= \int_a^b \Big(f(x)^2 - 2f(x)(c_0+c_1x) + (c_0^2 + 2c_0c_1x + c_1^2x^2)\Big)\,\mathrm{d}x \\ &= \int_a^b f(x)^2\,\mathrm{d}x - 2c_0\int_a^b f(x)\,\mathrm{d}x - 2c_1\int_a^b xf(x)\,\mathrm{d}x \\ &\qquad + c_0^2(b-a) + c_0c_1(b^2-a^2) + \tfrac{1}{3}c_1^2(b^3-a^3). \end{aligned}$$

To find the optimal polynomial, p_*, we need to optimize E over c_0 and c_1, i.e., we must 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

$$\begin{aligned} \frac{\partial E}{\partial c_0} &= -2\int_a^b f(x)\,\mathrm{d}x + 2c_0(b-a) + c_1(b^2-a^2) \\ \frac{\partial E}{\partial c_1} &= -2\int_a^b xf(x)\,\mathrm{d}x + c_0(b^2-a^2) + \tfrac{2}{3}c_1(b^3-a^3). \end{aligned}$$

Setting these partial derivatives equal to zero yields

$$\begin{aligned} 2c_0(b-a) + c_1(b^2-a^2) &= 2\int_a^b f(x)\,\mathrm{d}x \\ c_0(b^2-a^2) + \tfrac{2}{3}c_1(b^3-a^3) &= 2\int_a^b xf(x)\,\mathrm{d}x. \end{aligned}$$

These equations, linear in the unknowns c_0 and c_1, can be written in the matrix form

$$\begin{bmatrix} 2(b-a) & b^2-a^2 \\ b^2-a^2 & \frac{2}{3}(b^3-a^3) \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} = \begin{bmatrix} 2\int_a^b f(x)\,\mathrm{d}x \\ 2\int_a^b xf(x)\,\mathrm{d}x \end{bmatrix}.$$

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: $f(x) = e^x$. We apply this result to the function $f(x) = e^x$ for $x \in [0,1]$. Since

$$\int_0^1 e^x \, dx = e - 1, \qquad \int_0^1 xe^x \, dx = [e^x(x-1)]_{x=0}^1 = 1,$$

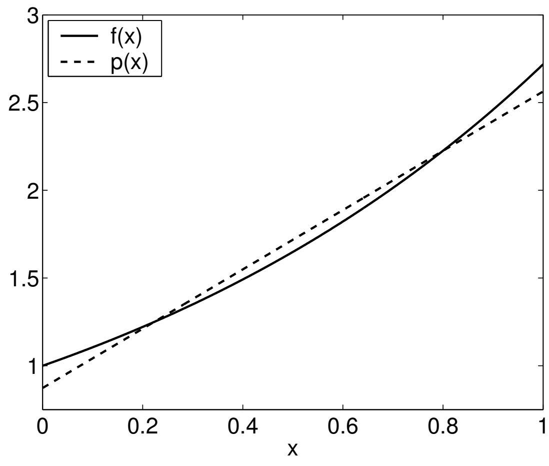
we must solve the system

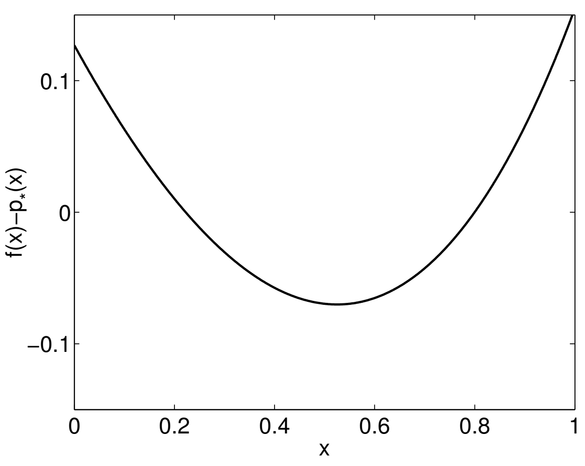
$$\begin{bmatrix} 2 & 1 \\ 1 & \frac{2}{3} \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} = \begin{bmatrix} 2e-2 \\ 2 \end{bmatrix}.$$

The desired solution is

$$c_0 = 4e - 10, \qquad c_1 = 18 - 6e.$$

Below we show a plot of this approximation (left), and the error $f(x) - p_*(x)$.





We can see from these pictures that the approximation looks decent to the eye, but the error is not terribly small. (In fact, $\|f - p_*\|_{L^2} = 0.06277\ldots$.) 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 L^2-optimal degree-n approximation.

3.3.3. 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 $\mathcal{P}_n$. Then 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

$$E(c_0,\ldots,c_n) := \|f(x)-p(x)\|_{L^2}^2 = \int_a^b \Big(f(x)-\sum_{k=0}^n c_k\phi_k(x)\Big)^2\,\mathrm{d}x$$

$$= \langle f,f\rangle - 2\sum_{k=0}^n c_k\langle f,\phi_k\rangle + \sum_{k=0}^n\sum_{\ell=0}^n c_kc_\ell\langle\phi_k,\phi_\ell\rangle.$$

As before, compute $\partial E/\partial c_j$ for $j=0,\ldots,n$:

$$\frac{\partial E}{\partial c_j} = -2\langle f,\phi_j\rangle + \sum_{k=0}^n 2c_k\langle\phi_k,\phi_j\rangle.$$

Setting $\partial E/\partial c_j = 0$ gives the $n+1$ equations

$$\langle f,\phi_j\rangle = \sum_{k=0}^n c_k\langle\phi_k,\phi_j\rangle.$$

This is simply a system of linear algebraic equations, which can be written in the matrix form

$$\begin{bmatrix} \langle\phi_0,\phi_0\rangle & \langle\phi_0,\phi_1\rangle & \cdots & \langle\phi_0,\phi_n\rangle \\ \langle\phi_1,\phi_0\rangle & \langle\phi_1,\phi_1\rangle & & \vdots \\ \vdots & & \ddots & \vdots \\ \langle\phi_n,\phi_0\rangle & \langle\phi_n,\phi_1\rangle & \cdots & \langle\phi_n,\phi_n\rangle \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} \langle f,\phi_0\rangle \\ \langle f,\phi_1\rangle \\ \vdots \\ \langle f,\phi_n\rangle \end{bmatrix},$$

which we shall denote as $\mathbf{Hc}=\mathbf{b}$.

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 $\mathbf{H}$ denote the $n+1$-dimensional Hilbert matrix, and suppose $\mathbf{b}$ is constructed so that the exact solution to the system $\mathbf{Hc}=\mathbf{b}$ 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 system are small). Unfortunately, this is not the case – entirely consistent with our analysis of the sensitivity of linear systems, studied in Section 1.4.2.

n	$\kappa(\mathbf{H})$	$\|\mathbf{c}-\widehat{\mathbf{c}}\|_2$
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

[‡]See M.-D. Choi, 'Tricks or treats with the Hilbert matrix,' *American Math. Monthly* 90 (1983) 301–312.

Clearly these errors are not acceptable!

The last few 2-norm condition numbers are in fact smaller than they ought to be, a consequence of the fact that MATLAB is not computing the singular value decomposition of the Hilbert matrix exactly. (MATLAB computes the condition number as the ratio of the maximum and minimum singular values.) The standard algorithm for computing singular values obtains answers with small *absolute* accuracy, but not small *relative* accuracy. Thus we expect that singular values smaller than about $10^{-16}\|\mathbf{H}\|_2$ may not even be computed to the correct order of magnitude.

In the next lecture, we will see how better-conditioned bases for $\mathcal{P}_n$ yield matrices $\mathbf{H}$ for which we can solve $\mathbf{Hx} = \mathbf{b}$ much more accurately.

3.3.4. Connection to discrete least squares.

Why did the continuous least squares approximation problem studied above directly lead to a square $(n+1) \times (n+1)$ linear system, while the discrete least squares problem introduced in Lecture 16 led to an $(m+1) \times (n+1)$ least squares problem?

In the discrete case, we seek to minimize $\|\mathbf{c} - \mathbf{Af}\|_2$, where (using the monomial basis)

$$\mathbf{A} = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ 1 & x_2 & x_2^2 & \cdots & x_2^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_m & x_m^2 & \cdots & x_m^n \end{bmatrix}, \qquad \mathbf{c} = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix}, \qquad \mathbf{f} = \begin{bmatrix} f(x_0) \\ f(x_1) \\ f(x_2) \\ \vdots \\ f(x_m) \end{bmatrix}.$$

We have seen that this discrete problem can be solved via the *normal equations*

$$\mathbf{A}^*\mathbf{Ac} = \mathbf{A}^*\mathbf{f}.$$

Now compute

$$\mathbf{A}^*\mathbf{f} = \begin{bmatrix} \sum_{k=0}^n f(x_k) \\ \sum_{k=0}^n x_k f(x_k) \\ \sum_{k=0}^n x_k^2 f(x_k) \\ \vdots \\ \sum_{k=0}^n x_k^n f(x_k) \end{bmatrix} \in \mathbb{C}^{n+1}.$$

Notice that if $m+1$ approximation points are uniformly spaced over $[a,b]$, $x_k = a + kh_m$ for $h_m = (b-a)/m$, we have

$$\lim_{m\to\infty} h_m \mathbf{A}^*\mathbf{f} = \begin{bmatrix} \int_a^b f(x)\,\mathrm{d}x \\ \int_a^b x f(x)\,\mathrm{d}x \\ \int_a^b x^2 f(x)\,\mathrm{d}x \\ \vdots \\ \int_a^b x^n f(x)\,\mathrm{d}x \end{bmatrix} = \begin{bmatrix} \langle f, 1\rangle \\ \langle f, x\rangle \\ \langle f, x^2\rangle \\ \vdots \\ \langle f, x^n\rangle \end{bmatrix},$$

which is precisely the right hand side vector $\mathbf{b} \in \mathbb{C}^{n+1}$ obtained for the *continuous least squares problem.* Similarly, the $(j+1, k+1)$ entry of the matrix $\mathbf{A}^*\mathbf{A} \in \mathbb{C}^{(n+1)\times(n+1)}$ 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 uniform grids, we have in the limit that

$$\lim_{m\to\infty} h_m (\mathbf{A}^*\mathbf{A})_{j+1,k+1} = \int_a^b x^{j+k}\,\mathrm{d}x = \langle x^j, x^k\rangle.$$

Thus in aggregate we have

$$\lim_{m\to\infty} h_m \mathbf{A}^*\mathbf{A} = \mathbf{H},$$

where $\mathbf{H}$ is the matrix that arose in the continuous least squares problem.

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* $\mathbf{H}\mathbf{c} = \mathbf{b}$ as we independently derived for polynomial approximation by *continuous least squares.* In the latter case, calculus led us directly to the normal equation form of the solution.

Lecture 20: Orthogonal Polynomials for Continuous Least Squares Problems

In the last lecture we saw how to reduce continuous least squares problems to systems of linear algebraic equations. 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

$$\begin{bmatrix} \langle\phi_0,\phi_0\rangle & \langle\phi_0,\phi_1\rangle & \cdots & \langle\phi_0,\phi_n\rangle \\ \langle\phi_1,\phi_0\rangle & \langle\phi_1,\phi_1\rangle & & \vdots \\ \vdots & & \ddots & \vdots \\ \langle\phi_n,\phi_0\rangle & \langle\phi_n,\phi_1\rangle & \cdots & \langle\phi_n,\phi_n\rangle \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} \langle f,\phi_0\rangle \\ \langle f,\phi_1\rangle \\ \vdots \\ \langle f,\phi_n\rangle \end{bmatrix}.$$

The monomial basis $\phi_k(x) = x^k$ can give poor numerical approximations even for fairly small values of n due to the fragility of the Hilbert matrix. Here we show how to construct a basis for $\mathcal{P}_n$ that proves to be more robust.

3.3.5. Orthogonal polynomials.

We say two vectors are orthogonal if their inner product is zero. The same idea leads to the notion of orthogonality of functions in $C[a,b]$. It will prove useful for us to generalize the notion of inner product introduced in §3.3.1. For any function $w \in C[a,b]$ with $w(x) > 0$ (actually, we can allow $w(x) = 0$ only on a set of measure zero), we define

$$\langle f,g\rangle = \int_a^b f(x)g(x)w(x)\,\mathrm{d}x.$$

One can confirm that this definition is consistent with the axioms required of an inner product that were enumerated in the last lecture. This inner product thus motivates the following definition.

Definition. Two functions f and g are *orthogonal* if $\langle f,g\rangle = 0$.

Definition. 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.

Proposition. The system of orthogonal polynomials $\{\phi_k\}_{k=0}^{\ell}$ is a basis for $\mathcal{P}_\ell$, for all $\ell = 0,\ldots,n$.

This leads immediately to our first key theorem, one we will use repeatedly.

Theorem. 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. Our previous proposition implies that $\{\phi_k\}_{k=0}^{n-1}$ is a basis for $\mathcal{P}_{n-1}$. Thus for any $p \in \mathcal{P}_{n-1}$,

$$p = \sum_{k=0}^{n-1} c_k \phi_k$$

for some constants $\{c_k\}_{k=0}^{n-1}$. The linearity of the inner product and orthogonality of $\{\phi_k\}_{k=0}^{n}$ imply that

$$\langle p, \phi_n \rangle = \Big\langle \sum_{k=0}^{n-1} c_k \phi_k, \phi_n \Big\rangle = \sum_{k=0}^{n-1} c_k \langle \phi_k, \phi_n \rangle = \sum_{k=0}^{n-1} 0 = 0. \qquad \blacksquare$$

We need a mechanism for constructing orthogonal polynomials. The Gram–Schmidt process used to orthogonalize vectors in $\mathbb{C}^n$ can easily 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.

Gram–Schmidt orthogonalization. Given a basis $\{p_0, \ldots, p_n\}$ for some subspace $\mathcal{S}$, the following algorithm will construct 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, it requires a nontrivial amount of computation. As k gets larger, the work required in the sum at step k grows: the work grows with every step. (Recall that when dealing with functions on $C[a, b]$, each inner product evaluation requires the computation of an integral, potentially a expensive operation.)

To construct a set of orthogonal polynomials, we take some a basis $\{p_k\}_{k=0}^{n}$ for $\mathcal{P}_n$, and perform Gram–Schmidt orthogonalization. If p_k has exact degree k for $k = 0, \ldots, n$, then ϕ_k will have exact degree k as well, as required for a system of orthogonal polynomials. The simplest basis for $\mathcal{P}_n$ is the monomial basis, $\{x^k\}_{k=0}^{n}$. One could perform Gram–Schmidt orthogonalization directly on this basis to obtain orthogonal polynomials, but there is a slicker alternative for which most of the terms in the sum for ϕ_k turn out to be zero.

Suppose one has a set of orthogonal polynomials, $\{\phi_k\}_{k=0}^{n}$, and seeks the next orthogonal polynomial, ϕ_{n+1}. Since ϕ_n has exact degree n, the polynomial $x\phi_n(x)$ has exact degree $n + 1$. Thus, we could apply Gram–Schmidt orthogonalization on $\{\phi_0(x), \phi_1(x), \ldots, \phi_n(x), x\phi_n(x)\}$, which forms a basis for $\mathcal{P}_{n+1}$. This will allow us to make an essential simplification to the customary Gram–Schmidt recurrence

$$\phi_{n+1}(x) = x\phi_n(x) - \sum_{j=0}^{n} \frac{\langle x\phi_n(x), \phi_j(x) \rangle}{\langle \phi_j, \phi_j \rangle} \phi_j(x).$$

First notice that

$$\begin{aligned} \langle x\phi_n(x), \phi_k(x) \rangle &= \int_a^b \Big(x\phi_n(x) \Big) \phi_k(x) w(x) \, \mathrm{d}x \\ &= \int_a^b \phi_n(x) \Big(x\phi_k(x) \Big) w(x) \, \mathrm{d}x \\ &= \langle \phi_n(x), x\phi_k(x) \rangle. \end{aligned}$$

Since $x\phi_k(x) \in \mathcal{P}_{k+1}$,

$$\langle x\phi_n(x), \phi_k(x) \rangle = \langle \phi_n(x), x\phi_k(x) \rangle = 0$$

for all $j < n-1$. This eliminates the bulk of the terms from Gram–Schmidt sum:

$$\sum_{k=0}^{n} \frac{\langle x\phi_n(x), \phi_k(x)\rangle}{\langle \phi_k, \phi_k\rangle}\phi_k = \sum_{k=n-1}^{n} \frac{\langle x\phi_n(x), \phi_k(x)\rangle}{\langle \phi_k, \phi_k\rangle}\phi_k.$$

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.

Theorem (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:

$$\begin{aligned}
\phi_0(x) &= 1,\\
\phi_1(x) &= x - \frac{\langle x, 1\rangle}{\langle 1, 1\rangle},\\
\phi_k(x) &= x\phi_{k-1}(x) - \frac{\langle x\phi_{k-1}(x), \phi_{k-1}(x)\rangle}{\langle \phi_{k-1}(x), \phi_{k-1}(x)\rangle}\phi_{k-1}(x) - \frac{\langle x\phi_{k-1}(x), \phi_{k-2}(x)\rangle}{\langle \phi_{k-2}(x), \phi_{k-2}(x)\rangle}\phi_{k-2}(x) \qquad \text{for } k \geq 2.
\end{aligned}$$

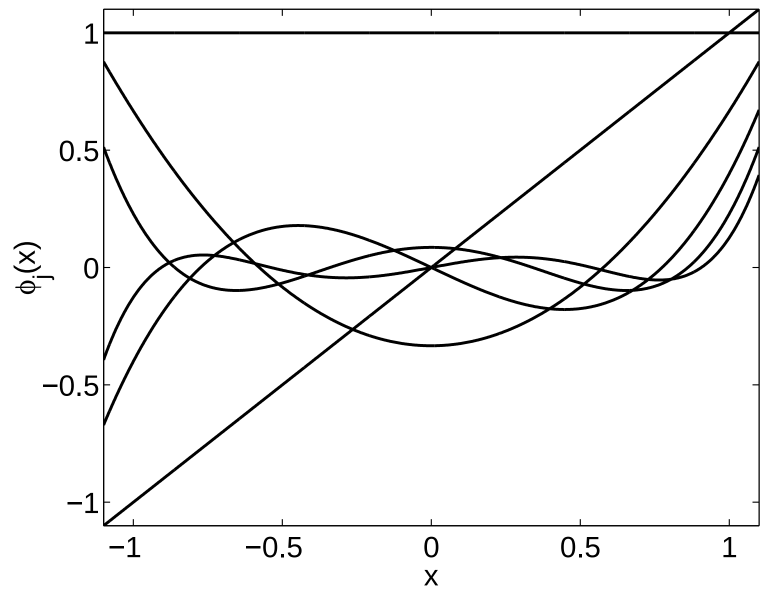
Our definition of orthogonal polynomials made no stipulation about normalization. It is often convenient to work with monic polynomials, i.e., $\phi_k(x) = x^k + \cdots$, as constructed by the three-term recurrence above. Some applications make other normalizations more convenient, e.g., $\langle \phi_k, \phi_k\rangle = 1$ or $\phi(0) = 1$. It is a simple exercise to adapt the three term recurrence to generate such alternative normalizations.

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*:

$$\begin{aligned}
\phi_0(x) &= 1\\
\phi_1(x) &= x\\
\phi_2(x) &= x^2 - \tfrac{1}{3}\\
\phi_3(x) &= x^3 - \tfrac{3}{5}x\\
\phi_4(x) &= x^4 - \tfrac{6}{7}x^2 + \tfrac{3}{35}\\
\phi_5(x) &= x^5 - \tfrac{10}{9}x^3 + \tfrac{5}{21}x\\
\phi_6(x) &= x^6 - \tfrac{15}{11}x^4 + \tfrac{5}{11}x^2 - \tfrac{5}{231}.
\end{aligned}$$

[†] The Gram–Schmidt process will not reduce 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 $[a,b]$.

Below we show a plot of $\phi_0, \phi_1, \ldots, \phi_5$. Note how distinct these polynomials are from one another, somewhat reminiscent of the Lagrange basis functions for polynomial interpolation.



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.

3.3.6. Continuous least squares with orthogonal polynomials.

Definition. A system of orthogonal polynomials $\{\psi_k\}_{k=0}^n$ is *orthonormal* provided that $\langle \psi_k, \psi_k \rangle = 1$ for all $k = 0, \ldots, 1$.

Given a any set of orthogonal polynomials $\{\phi_k\}_{k=0}^n$, we obtain orthonormal polynomials by setting

$$\psi_k := \frac{\phi_k}{\langle \phi_k, \phi_k \rangle^{1/2}},$$

giving

$$\langle \psi_k, \psi_k \rangle = \frac{\langle \phi_k, \phi_k \rangle}{\langle \phi_k, \phi_k \rangle} = 1.$$

We seek an expression for the least squares approximation to f as a linear combination of orthonormal polynomials. That is, determine the coefficients $\{c_k\}_{k=0}^n$ in the expansion

$$p(x) = \sum_{k=0}^{n} c_k \psi_k(x)$$

to minimize $\|f - p\|_{L^2}$. The optimal choice of coefficients follows immediately from the linear system derived in the last lecture,

$$\begin{bmatrix} \langle\psi_0,\psi_0\rangle & \langle\psi_0,\psi_1\rangle & \cdots & \langle\psi_0,\psi_n\rangle \\ \langle\psi_1,\psi_0\rangle & \langle\psi_1,\psi_1\rangle & & \vdots \\ \vdots & & \ddots & \vdots \\ \langle\psi_n,\psi_0\rangle & \langle\psi_n,\psi_1\rangle & \cdots & \langle\psi_n,\psi_n\rangle \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} \langle f,\psi_0\rangle \\ \langle f,\psi_1\rangle \\ \vdots \\ \langle f,\psi_n\rangle \end{bmatrix}.$$

Since $\{\psi_k\}_{k=0}^n$ is a system of orthonormal polynomials, this matrix equation reduces to

$$\begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & & \vdots \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} \langle f,\psi_0\rangle \\ \langle f,\psi_1\rangle \\ \vdots \\ \langle f,\psi_n\rangle \end{bmatrix},$$

with the trivial solution $c_k = \langle f,\psi_k\rangle$. As this linear system clearly has a unique solution, the optimal polynomial must be unique.

Theorem. The unique optimal L^2 approximation to $f \in C[a,b]$ on $[a,b]$ is given by

$$p_* = \sum_{k=0}^n \langle f,\psi_k\rangle\psi_k,$$

where $\{\psi_k\}_{k=0}^n$ forms a system of orthonormal polynomials on $[a,b]$.

From this expression for the optimal polynomial immediately follows a fundamental property of all least squares approximations.

Theorem (Orthogonality of the optimal L^2 error). Let $p_* \in \mathcal{P}_n$ be the optimal L^2 approximation to $f \in C[a,b]$. Then $f - p_*$ is orthogonal to all $q \in \mathcal{P}_n$, i.e., $\langle f - p_*, q\rangle = 0$.

Proof. Given any $q \in \mathcal{P}_n$, express this polynomial in the basis of orthonormal polynomials,

$$q = \sum_{k=0}^n \gamma_k\psi_k.$$

We have just shown that p_* takes the form

$$p_* = \sum_{k=0}^n \langle f,\psi_k\rangle\psi_k.$$

Since $\{\psi_k\}_{k=0}^n$ forms a basis for $\mathcal{P}_n$, it suffices to show that $f - p_*$ is orthogonal to each ψ_k. In particular, for $k = 0,\ldots,n$, we have

$$\begin{aligned} \langle f - p_*, \psi_k\rangle &= \langle f - \textstyle\sum_{j=0}^n \langle f,\psi_j\rangle\psi_j, \psi_k\rangle \\ &= \langle f,\psi_k\rangle - \textstyle\sum_{j=0}^n \langle f,\psi_j\rangle\langle\psi_j,\psi_k\rangle \\ &= \langle f,\psi_k\rangle - \langle f,\psi_k\rangle\langle\psi_k,\psi_k\rangle \\ &= \langle f,\psi_k\rangle - \langle f,\psi_k\rangle \\ &= 0. \end{aligned}$$

Since $f - p_*$ is orthogonal to all members of a basis for $\mathcal{P}_n$, it is orthogonal to any member of $\mathcal{P}_n$:

$$\langle f - p_*, q\rangle = \sum_{k=0}^{n} \gamma_k \langle f - p_*, \psi_k\rangle = \sum_{k=0}^{n} 0 = 0. \qquad \blacksquare$$

Example: $f(x) = e^x$. We repeat our previous example: approximating $f(x) = e^x$ on $[0, 1]$ with a linear polynomial. First, we need to construct orthonormal polynomials for this interval. It is easy to see that $\psi_0(x) = 1$, and a straightforward computation gives $\psi_1(x) = \sqrt{3}(1 - 2x)$. We then compute

$$\begin{aligned} \langle e^x, \psi_0(x)\rangle &= \int_0^1 e^x \, dx = e - 1 \\ \langle e^x, \psi_1(x)\rangle &= \sqrt{3}\int_0^1 e^x (1-2x)\, dx = \sqrt{3}(e-3), \end{aligned}$$

giving a formula for p_*:

$$\begin{aligned} p_* &= (e-1)\psi_0 + \sqrt{3}(e-3)\psi_1 \\ &= (e-1)1 + \sqrt{3}(e-3)[\sqrt{3}(1-2x)] \\ &= 4e - 10 + x(18 - 6e). \end{aligned}$$

This is exactly the polynomial we obtained using basic calculus techniques.

Note that with this procedure, one can easily to increase the degree of the approximating polynomial. To increase the degree by one, simply add

$$\langle f, \psi_{n+1}\rangle \psi_{n+1}$$

to the old approximation. True, this requires computation of an integral, but the general method we discussed in the last lecture would also require a new integral evaluation to include in the right hand side of the $(n+2)$-by-$(n+2)$ linear system, which then must be solved to get the new 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.

```
% Code to demonstrate computation of continuous least squares approximation.
% Uses MATLAB's built-in codes to compute inner products.

% Use the weight function w(x) = 1 on the interval [-1,1].
% Construct the orthogonal polynomials for this weight, interval.
% These are the Legendre polynomials; one can look up their coefficients
% in mathematical tables.  We input them in MATLAB's standard format
% for polynomials.  (We have normalized the standard Legendre polynomials.)

Leg = [[     0     0       0      0      0      0      1]*sqrt(1/2);  % psi_0(x)
       [     0     0       0      0      0      1      0]*sqrt(3/2);  % psi_1(x)
       [     0     0       0      0    3/2      0   -1/2]*sqrt(5/2);  % psi_2(x)
```

[‡]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.

```
       [     0     0       0    5/2      0    -3/2     0]*sqrt(7/2);  % psi_3(x)
       [     0     0    35/8      0  -15/4      0   3/8]*sqrt(9/2);  % psi_4(x)
       [     0  63/8       0  -35/4      0    15/8     0]*sqrt(11/2); % psi_5(x)
       [231/16     0 -315/16      0 105/16      0 -5/16]*sqrt(13/2)];% psi_6(x)

% All the necessary integrals have integrands that are the product of our
% target function f(x) = exp(x)*sin(5*x) and some polynomial.
% The following inline function defines this general form of integrand.
 f = inline('sin(pi*x) + 3*exp(-(50*(x-.5)).^2)');
 integrand = inline('feval(f,x).*polyval(p,x)','x','f','p');

% We also include a function to evaluate the 2-norm of the error
 errintegrand = inline('(feval(f,x)-polyval(p,x)).*polyval(q,x)','x','f','p','q');

% compute the expansion coefficients for the optimal polynomial approximation
 x = linspace(-1.1,1.1,1000)';
 figure(1),clf
 plot(x,f(x),'b-','linewidth',3), hold on
 axis([-1.1 1.1 -2 5])
 set(gca,'fontsize',20)
 drawnow
 px = zeros(1,size(Leg,1));

 clear pxplt
 for j=1:size(Leg,1)
     input('press return to continue')
     c(j) = quad(integrand,-1,1,1e-10,[],f,Leg(j,:));
     px = px + c(j)*Leg(j,:);
     fprintf(' c_%d = %10.7f \n', j-1, c(j))
     if exist('pxplt','var'), set(pxplt,'linewidth',1); end
     pxplt = plot(x,polyval(px,x),'r-','linewidth',3);
     quad(errintegrand,-1,1,1e-10,[],f,px,[1])
     title(sprintf('Degree %d Least-Squares Approximation',j),'fontsize',20)
 end
```

Appendix.

We derived the formula

$$p_* = \sum_{k=0}^{n} \langle f, \psi_k \rangle \psi_k$$

based on a simple calculus result from the previous lecture. Here is an alternative derivative-free exposition that mirrors the construction of the discrete least squares solution in §3.1.

For a general $p \in \mathcal{P}_n$, write

$$p = \sum_{k=0}^{n} c_k \psi_k.$$

Using the linearity of the inner product, and the fact that $\{\psi_k\}_{k=0}^{n}$ is a system of orthonormal polynomials, we have

$$\|f-p\|_{L^2}^2 = \langle f-p, f-p \rangle \;=\; \langle f, f \rangle - 2\langle f, p \rangle + \langle p, p \rangle$$

$$\begin{aligned}
&= \|f\|_{L^2}^2 - 2\langle f, \textstyle\sum_{k=0}^n c_k\psi_k\rangle + \langle \textstyle\sum_{k=0}^n c_k\psi_k, \textstyle\sum_{j=0}^n c_j\psi_j\rangle \\
&= \|f\|_{L^2}^2 - 2\textstyle\sum_{k=0}^n c_k\langle f, \psi_k\rangle + \textstyle\sum_{k=0}^n \textstyle\sum_{j=0}^n c_k c_j\langle \psi_k, \psi_j\rangle \\
&= \|f\|_{L^2}^2 - 2\textstyle\sum_{k=0}^n c_k\langle f, \psi_k\rangle + \textstyle\sum_{k=0}^n c_k^2\langle \psi_k, \psi_k\rangle \\
&= \|f\|_{L^2}^2 - 2\textstyle\sum_{k=0}^n c_k\langle f, \psi_k\rangle + \textstyle\sum_{k=0}^n c_k^2.
\end{aligned}$$

Despite these manipulations, it is still not clear how we should choose the c_j to give the least squares approximation. Toward this end, note that

$$(c_k - \langle f, \psi_k\rangle)^2 = c_k^2 - 2c_k\langle f, \psi_k\rangle + \langle f, \psi_k\rangle^2.$$

Rearranging this expression and summing over k, we have

$$-2\textstyle\sum_{k=0}^n c_k\langle f, \psi_k\rangle + \textstyle\sum_{k=0}^n c_k^2 = \textstyle\sum_{k=0}^n \Big[(c_k - \langle f, \psi_k\rangle)^2 - \langle f, \psi_k\rangle^2\Big].$$

Substituting this formula into our expression for the error, we obtain

$$\|f-p\|_{L^2}^2 \;=\; \|f\|_{L^2}^2 + \sum_{k=0}^n (c_k - \langle f, \psi_k\rangle)^2 - \sum_{k=0}^n \langle f, \psi_k\rangle^2.$$

The first term in this key expression, $\|f\|_{L^2}^2$, is independent of our choice of the c_k, as is the last term, $-\sum_{k=0}^n \langle f, \psi_k\rangle^2$. Thus, to minimize $\|f\|_{L^2}^2$, minimize the middle term

$$\sum_{k=0}^n (c_k - \langle f, \psi_k\rangle)^2.$$

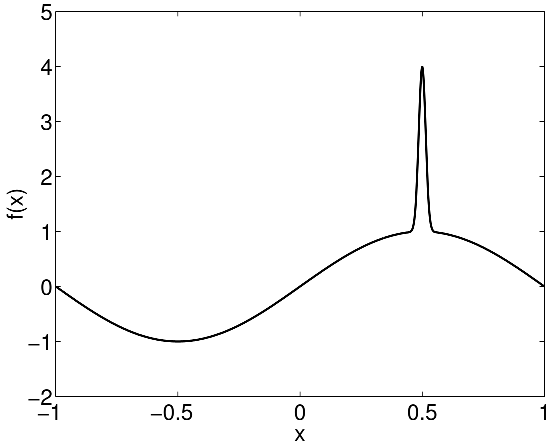
As this term is nonnegative, our best hope is to find coefficients c_k that zero out this expression. That is easy:

$$c_k = \langle f, \psi_k\rangle.$$

Moreover, this is the only choice for the c_k that will zero the middle term. Hence, we have constructed the optimal polynomial p_* and shown it to be unique.

Lecture 21: Minimax Approximation

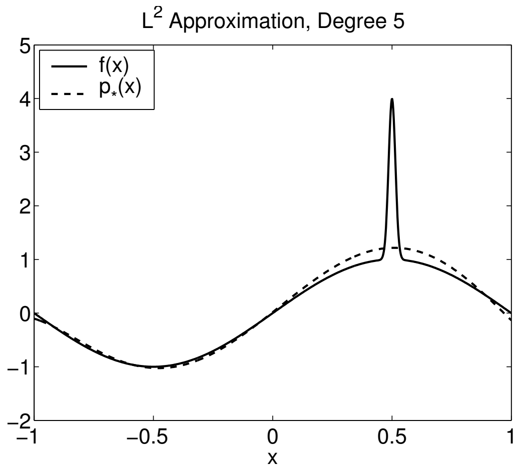
In many applications, the L^2-norm has a physical interpretation – often associated with some measure of energy – and this makes continuous least-squares approximation particularly appealing (e.g., the best approximation minimizes the energy of the error). Moreover, that optimal polynomial approximation in this norm can be computed at the expense of a few inner products (integrals). However, like discrete least-squares, this approach suffers from a potential problem: it minimizes the influence of outlying data, i.e., points where the function f varies wildly over a small portion of the interval $[a, b]$. Such an example is shown below.[†]

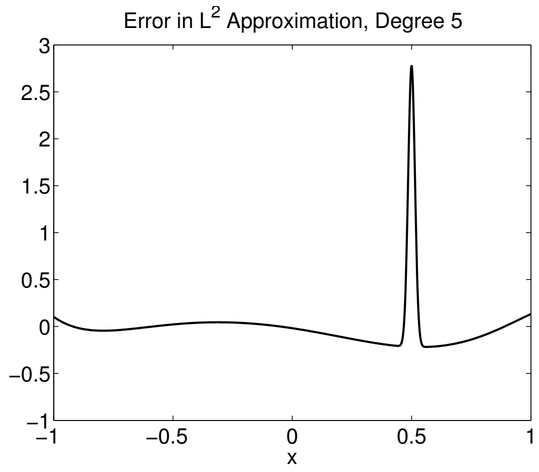


For this function the L^2-norm of the error,

$$\|f - p_*\|_{L^2} = \Big(\int_a^b (f(x) - p_*(x))^2 \,\mathrm{d}x\Big)^{1/2},$$

averages out the discrepancy $f - p_*$ over all $x \in [a, b]$, so it is possible to have a large error $f(x) - p_*(x)$ on some narrow range of x values that makes a negligible contribution to the integral. Below on the left, we compare the function shown above to its degree-5 least-squares approximation; on the right, we show the error $f - p_*$, which is small throughout $[-1, 1]$ except for a large spike.





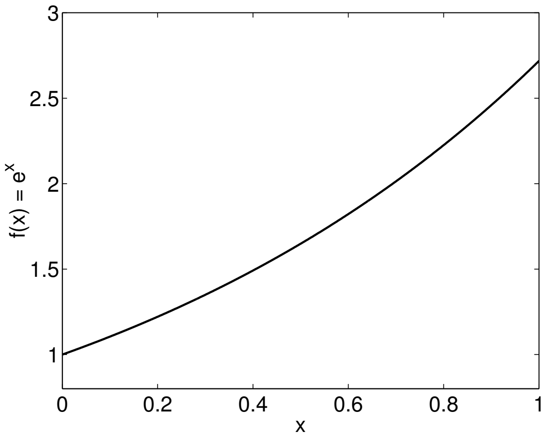
[†]The function in question is $f(x) = \sin(\pi x) + 3\exp(-50(x - \frac{1}{2})^2)$. Despite the nasty appearance of the plot, this is function is perfectly smooth: $f(x) \in C^\infty[-1, 1]$.

Functions of this sort may seem pathological, but they highlight the fact that L^2-optimization does not always generate a polynomial p_* that is close to f throughout the interval $[a, b]$. Indeed, in a number of settings the L^2-norm is the wrong way to measure error: we really want to minimize $\max_{x\in[a,b]} |f(x) - p(x)|$.

3.4 Minimax approximation. The goal of minimizing the maximum deviation of a polynomial p from our function f is called *minimax* (or *uniform*, or L^∞) approximation, since

$$\min_{p\in\mathcal{P}_k} \max_{x\in[a,b]} |f(x) - p(x)| \;=\; \min_{p\in\mathcal{P}_k} \|f - p\|_{L^\infty}.$$

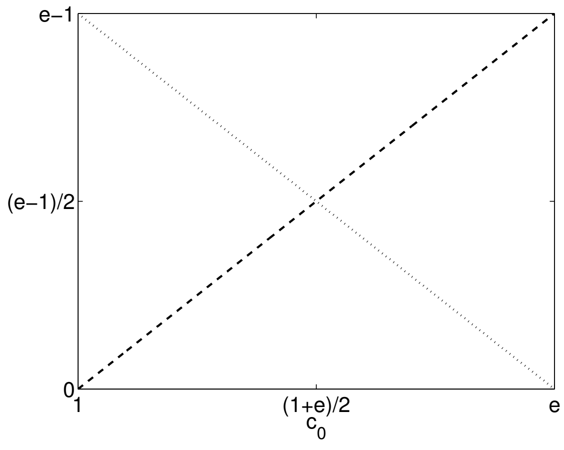
A simple example. Suppose we seek the constant that best approximates $f(x) = \mathrm{e}^x$ over the interval $[0, 1]$, shown below.



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

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

The following figure shows $|\mathrm{e}^0 - c_0|$ (broken line) and $|\mathrm{e}^1 - c_0|$ (dotted line) for $c_0 \in [1, \mathrm{e}]$.

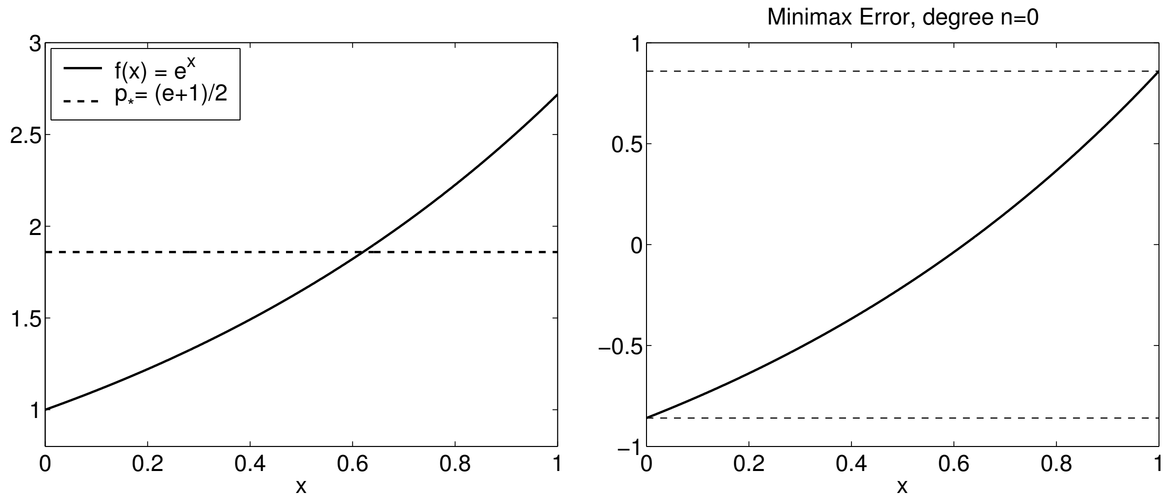


The optimal value for c_0 will be the point at which the larger of these two lines is minimal. The figure above clearly reveals that this happens when the errors are equal, at $c_0 = (1+\mathrm{e})/2$. We conclude that the optimal minimax constant polynomial approximation to e^x on $x \in [0,1]$ is $p_*(x)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

The plots below compare f to the optimal polynomial p_* (left), and show the error $f - p_*$ (right). We picked c_0 to be the point at which the error was equal in magnitude at the end points $x = 0$ and $x = 1$; in fact, it is equal in magnitude, but opposite in sign,

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

as seen in the illustration on the right below. It turns out that this property—maximal error attained at various points in the interval with alternating sign—is a key feature of minimax approximation.



3.4.1. Oscillation Theorem. As hinted in the previous example, 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 (from Süli and Mayers, §8.3) to give a general impression of how such results are established.

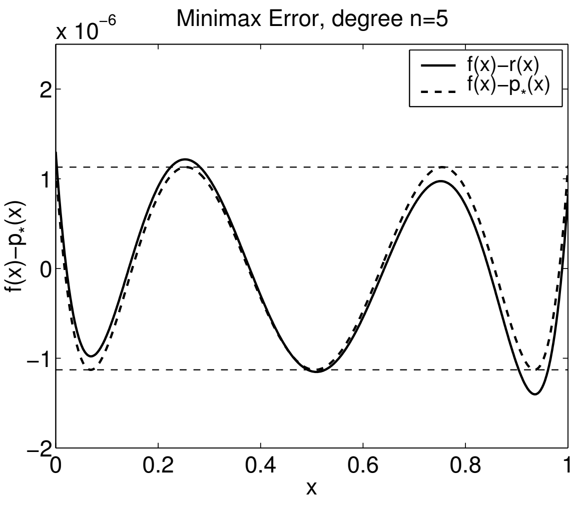
Theorem (de la Vallée Poussin Theorem). Let $f \in C[a,b]$ and suppose $r \in \mathcal{P}_n$ is some polynomial for which there exist $n+2$ points $\{x_j\}_{j=0}^{n+1}$ with $a \le x_0 < x_1 < \cdots < x_{n+1} \le b$ at which the error $f(x) - r(x)$ oscillates signs, i.e.,

$$\mathrm{sign}(f(x_j) - r(x_j)) = -\mathrm{sign}(f(x_{j+1}) - r(x_{j+1}))$$

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

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

Before proving this result, we provide a numerical illustration. Here we are approximating $f(x) = \mathrm{e}^x$ with a 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 plot below we can see that for this r, we can find $n+2 = 7$ points at which the sign of the error $f(x) - r(x)$ oscillates. The broken line shows the error curve for the optimal minimax polynomial p_* (whose computation is discussed below). Here is the point of the de la Vallée Poussin theorem: *Since the error $f(x)-r(x)$ oscillates sign $n+2$ times, there must be some $x \in [0,1]$ at which the minimax error $\pm\|f-p_*\|_{L^\infty}$ (denoted by the horizontal broken lines) exceeds $|f(x)-r(x)|$ at one of the points that give the oscillating sign.* In other words, the de la Vallée Poussin theorem provides a mechanism for developing *lower bounds* on $\|f-p_*\|_{L^\infty}$.



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,

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

We will prove the result by contradiction. Thus suppose

$$\|f-p_*\|_{L^\infty} < |f(x_j)-r(x_j)|, \qquad \text{for all } j=0,\ldots,n+1. \tag{21.1}$$

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:

$$|f(x_j)-p_*(x_j)| < |f(x_j)-r(x_j)|, \qquad \text{for all } j=0,\ldots,n+1. \tag{21.2}$$

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 (21.2) 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 $p_*(x_j)-r(x_j)$ will always have the same sign as $f(x_j)-r(x_j)$. 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 $p_*(x)-r(x)\in\mathcal{P}_n$ 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 (21.1). Hence, there must be at least one j for which

$$\|f-p_*\|_{L^\infty} \ge |f(x_j)-r(x_j)|. \qquad \blacksquare$$

The following result has the same flavor, but it is considerably more precise (with a more intricate proof, which we omit).[‡]

Theorem (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

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

and

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

In words, this means that the optimal error, $f - p_*$, attains its maximum at $n+2$ points, with the error alternating sign between consecutive points.

Note that this result is *if and only if*: the oscillation property exactly *characterizes* the minimax approximation. If you can present some polynomial $p_* \in \mathcal{P}_n$ such that $f - p_*$ satisfies the oscillation property, then this p_* must be the unique minimax approximation!

Theorem (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.

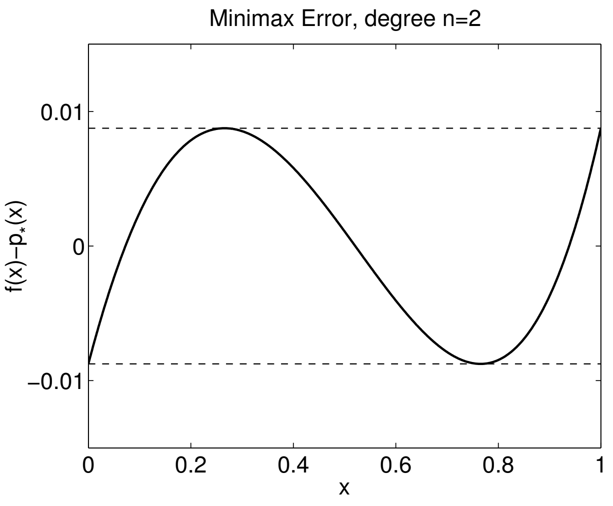
The proof is a straightforward application of the Oscillation Theorem. One can show that any two potential minimax polynomials must have the same $n+2$ critical oscillation points. Any two degree-n polynomials that agree at $n+2$ points must be identical. See Süli and Mayers, Theorem 8.5, for details.

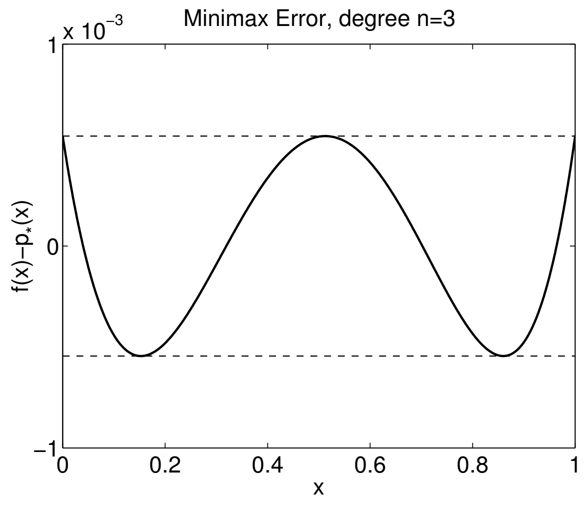
This oscillation property forms the basis of algorithms that find the minimax approximation: iteratively adjust an approximating polynomial until it satisfies the oscillation property. The most famous algorithm for computing the minimax approximation is called the *Remez exchange algorithm*, essentially a specialized linear programming procedure. In exact arithmetic, this algorithm is guaranteed to terminate with the correct answer in finitely many operations.

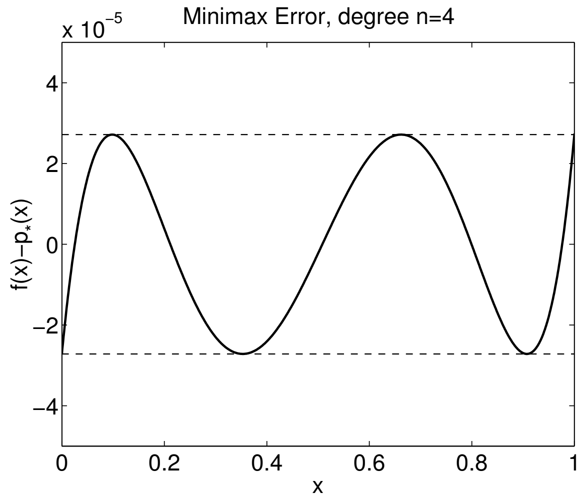
The oscillation property is demonstrated in the previous example, where we approximated $f(x) = e^x$ with a constant. Indeed, the maximum error is attained at two points (that is, $n+2$, since $n = 0$), and the error differs in sign at those points. The pictures below show the errors $f(x) - p_*(x)$ 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.

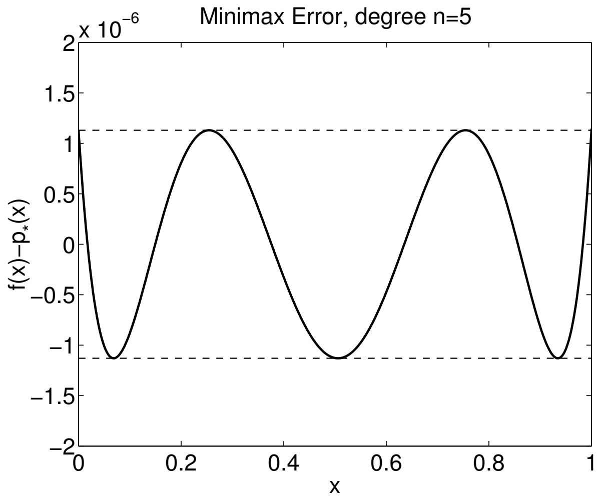
[‡]For a proof, see Süli and Mayers, §8.3. Another excellent resource is G. W. Stewart, *Afternotes Goes to Graduate School*, SIAM, 1998; see Stewart's Lecture 3.

[§]These examples were computed using the COCA package, software written by Bernd Fischer and Jan Modersitski that even solves minimax approximation problems when the interval $[a,b]$ is replaced by a region of the complex plane.









Example: $\mathbf{e}^{\boldsymbol{x}}$ revisited. Now we shall use the Oscillation Theorem to compute the optimal linear minimax approximation to $f(x) = \mathrm{e}^x$ 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 by the Oscillation Theorem, we are guaranteed 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 our values for x_0, x_2, and $p_*(x) = \alpha + \beta x$, these equations become

$$\begin{aligned} 1-\alpha &= \delta \\ e^{x_1}-\alpha-\beta x_1 &= -\delta \\ e-\alpha-\beta &= \delta. \end{aligned}$$

The first and third equation together imply $\beta = e-1$. We also deduce that $2\alpha = e^{x_1} - x_1(e-1)+1$. There are a variety of choices for x_1 that will satisfy these conditions, but in those cases δ will not be the *maximal error*. It is key that

$$|\delta| = \max_{x\in[a,b]} |f(x)-p_*(x)|.$$

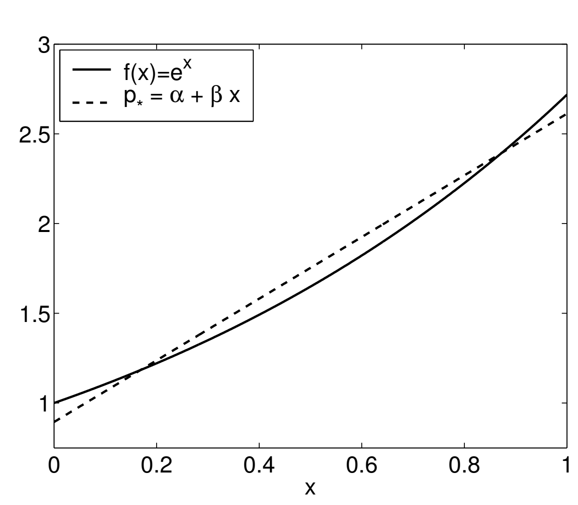
To make sure this happens, we can require that *the derivative of error* be zero at x_1, reflecting that the error $f-p_*$ attains a local minimum/maximum at x_1.The pictures on the previous page confirm that this is reasonable.[¶] Imposing the condition that $f'(x_1)-p_*'(x_1)=0$ yields

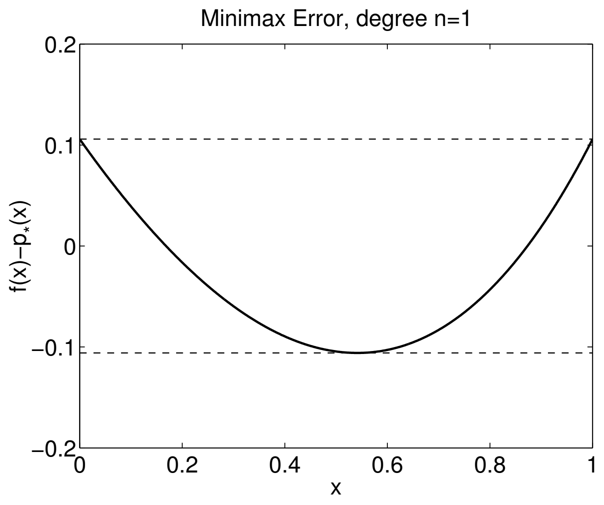
$$e^{x_1}-\beta = 0.$$

Now we can explicitly solve the equations to obtain

$$\begin{aligned} \alpha &= \tfrac{1}{2}(e-(e-1)\log(e-1)) = 0.89406\ldots \\ \beta &= e-1 = 1.71828\ldots \\ x_1 &= \log(e-1) = 0.54132\ldots \\ \delta &= \tfrac{1}{2}(2-e+(e-1)\log(e-1)) = 0.10593\ldots. \end{aligned}$$

An illustration of the optimal linear approximation we have just computed, along with the associated error, is shown below. Compare this approximation to the L^2-optimal linear polynomial computed at the beginning of our study of continuous least-squares minimization.





[¶]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, $x_j \in (a,b)$.

Lecture 22: Minimax Approximation, Optimal Interpolation, Chebyshev Polynomials

3.4.2. Optimal interpolation points.

As an application of the minimax approximation procedure, we consider how best to choose interpolation points $\{x_j\}_{j=0}^n$ to minimize

$$\|f - p_n\|_{L^\infty},$$

where $p_n \in \mathcal{P}_n$ is the interpolant to f at the specified points.

Recall the interpolation error bound developed in Section 2.4: 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

$$\|f - p_n\|_{L^\infty} = \max_{\xi\in[a,b]} \frac{|f^{(n+1)}(\xi)|}{(n+1)!}\ \max_{x\in[a,b]}\Big|\prod_{j=0}^{n}(x - x_j)\Big|.$$

For Runge's example, $f(x) = 1/(1+x^2)$ for $x \in [-5,5]$, we observed that $\|f - p_n\|_{L^\infty} \to \infty$ as $n \to \infty$ if the interpolation points $\{x_j\}$ are uniformly spaced over $[-5,5]$. However, Marcinkiewicz's theorem (Section 2.4) guarantees there is always some scheme for assigning the interpolation points such that $\|f - p_n\|_{L^\infty} \to 0$ as $n \to \infty$. In the case of Runge's function, we observed that the choice

$$x_j = 5\cos(j\pi/n), \qquad j = 0,\ldots,n$$

is one such scheme. 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, $\|f - p_n\|_{L^\infty}$). That is, we want to solve

$$\min_{x_0,\ldots,x_n}\ \max_{x\in[a,b]}\Big|\prod_{j=0}^{n}(x - x_j)\Big|. \tag{22.1}$$

Notice that

$$\begin{aligned}\prod_{j=0}^{n}(x - x_j) &= x^{n+1} - x^n\sum_{j=0}^{n}x_j + x^{n-1}\sum_{j=0}^{n}\sum_{k=0}^{n}x_jx_k - \cdots + (-1)^{n+1}\prod_{j=0}^{n}x_j\\ &= x^{n+1} - r(x),\end{aligned}$$

where $r \in \mathcal{P}_n$ is a degree-n polynomial depending on the interpolation nodes $\{x_j\}_{j=0}^n$.

To find the optimal interpolation points according to (22.1), we should solve

$$\min_{r\in\mathcal{P}_n}\ \max_{x\in[a,b]}|x^{n+1} - r(x)| \;=\; \min_{r\in\mathcal{P}_n}\|x^{n+1} - r(x)\|_{L^\infty}.$$

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. 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

$$\begin{aligned}
n = 0:&\quad T_0(x) = \cos(0 \cos^{-1} x) = \cos(0) = 1\\
n = 1:&\quad T_1(x) = \cos(\cos^{-1} x) = x\\
n = 2:&\quad T_2(x) = \cos(2 \cos^{-1} x) = 2\cos^2(\cos^{-1} x) - 1 = 2x^2 - 1.
\end{aligned}$$

For $n = 2$, we employed the identity $\cos 2\theta = 2\cos^2\theta - 1$, substituting $\theta = \cos^{-1} x$. More generally, we have the identity

$$\cos(n+1)\theta = 2\cos\theta\cos n\theta - \cos(n-1)\theta.$$

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. (In fact, Chebyshev polynomials are orthogonal polynomials on $[-1, 1]$ with respect to the inner product $\langle f, g\rangle = \int_a^b f(x)g(x)(1-x^2)^{-1/2}$, a fact we will use when studying Gaussian quadrature in a few lectures.)

Chebyshev polynomials exhibit a wealth of interesting properties, of which we mention just three.

Proposition. 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

$$T_n(\xi_j) = \cos\left(n \cos^{-1}\cos\left(\frac{(2j-1)\pi}{2n}\right)\right) = \cos\left(\frac{(2j-1)\pi}{2}\right) = 0,$$

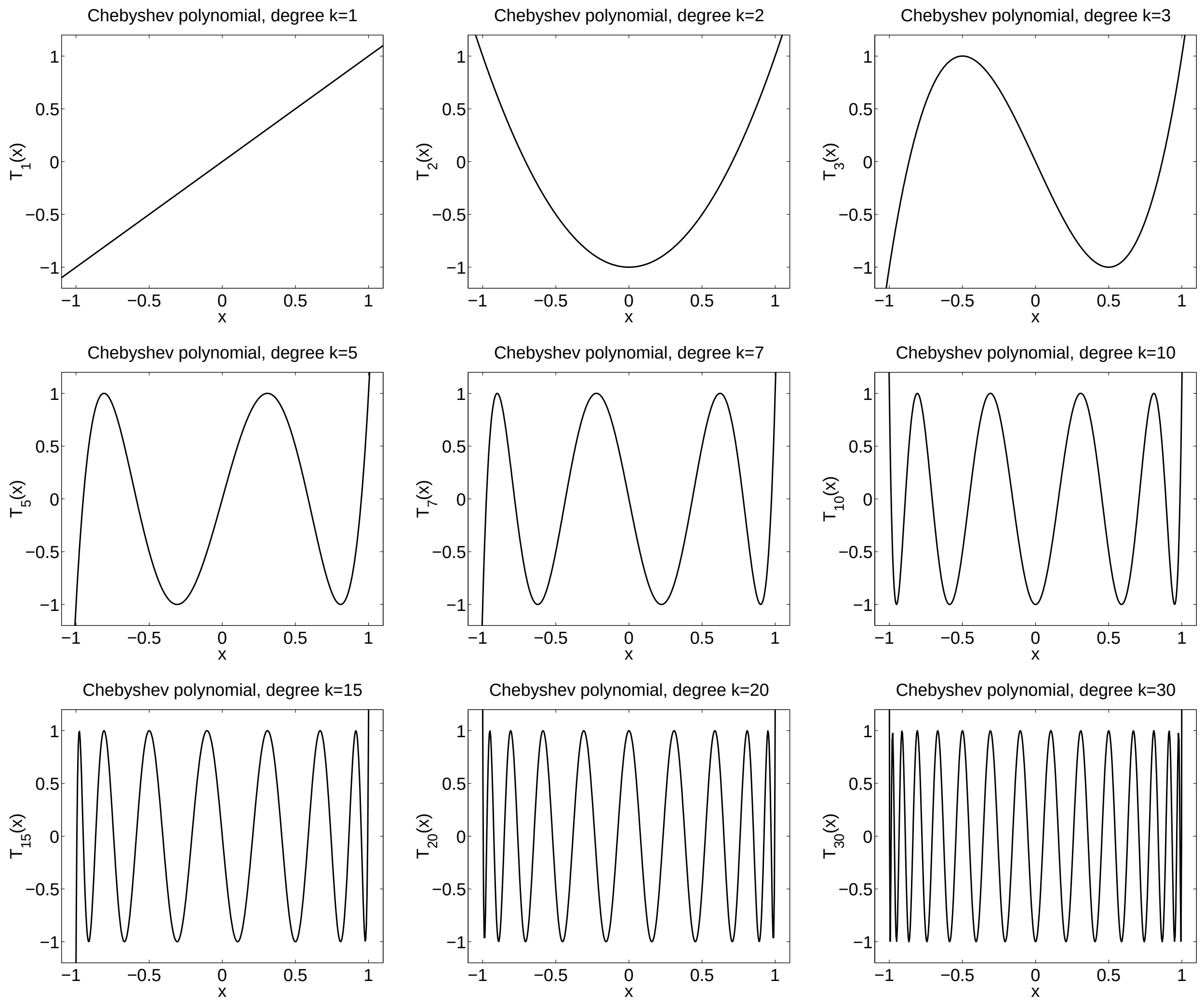
[†]Furthermore, it doesn't apply if $|x| > 1$. For such x one can define the Chebyshev polynomials using hyperbolic trigonometric functions, $T_n(x) = \cosh(n \cosh^{-1} x)$. Indeed, using hyperbolic trigonometric identities, one can show that this expression generates for $x \notin [-1, 1]$ the same polynomials we get for $x \in [-1, 1]$ from the standard trigonometric identities.

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 $[-1,1]$. $\blacksquare$

The figures below show Chebyshev polynomials T_n for nine different values of n.



The punchline. 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 $n \ge 0$. Define

$$\widehat{T}_{n+1} = 2^{-n}T_{n+1}$$

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 (Optimal approximation of x^{n+1}). The optimal approximation to x^{n+1} from $\mathcal{P}_n$ on the interval $x \in [-1, 1]$ is given by

$$r_n(x) = x^{n+1} - \widehat{T}_{k+1}(x) = x^{n+1} - 2^{-n} T_{k+1}(x) \in \mathcal{P}_n.$$

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.

Lecture 23: Interpolatory Quadrature

4. Quadrature.

The computation of continuous least squares approximations to $f \in C[a,b]$ required evaluations of the inner product $\langle f, \phi_j \rangle = \int_a^b f(x)\phi_j(x)\,\mathrm{d}x$, where ϕ_j is a polynomial (a basis function for $\mathcal{P}_n$). Often such integrals may be 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.†

4.1. Interpolatory Quadrature with Prescribed Nodes.

Given $f \in C[a,b]$, we seek approximations to the definite integral

$$\int_a^b f(x)\,\mathrm{d}x.$$

If $p_n \in \mathcal{P}_n$ interpolates f at $n+1$ points in $[a,b]$, then we might hope that

$$\int_a^b f(x)\,\mathrm{d}x \approx \int_a^b p_n(x)\,\mathrm{d}x.$$

This approach is known as *interpolatory* quadrature. Will such rules produce a reasonable estimate to the integral? Of course, that depends on properties of f and the interpolation points. When we interpolate at equally spaced points, the formulas that result are called *Newton–Cotes* quadrature rules.

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. This approach can be improved in two complementary ways: increasing the degree of the interpolating polynomial, and reducing the width of the subintervals over which each interpolating polynomial applies. The first approach leads to the *trapezoid rule* and *Simpson's rule*; the second yields *composite* rules, where the integral over $[a,b]$ is split into the sum of integrals over subintervals. In many cases, the function f may be fairly regular over part of the domain $[a,b]$, but then have some region of rapid growth or oscillation. We ultimately seek quadrature software that automatically detects such regions, ensuring that sufficient function evaluations are performed there without requiring excessive effort on areas where f is well-behaved. Such *adaptive* quadrature procedures, discussed briefly at the end of these notes, are essential for practical problems.‡

4.1.1 Trapezoid Rule.

†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 the topic of interpolatory quadrature, see Süli and Mayers, Chapter 7, which has guided many aspects of our presentation here.

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 $x = a$ and $x = b$,

$$p_1(x) = f(a) + \frac{f(b) - f(a)}{b - a}\,(x - a).$$

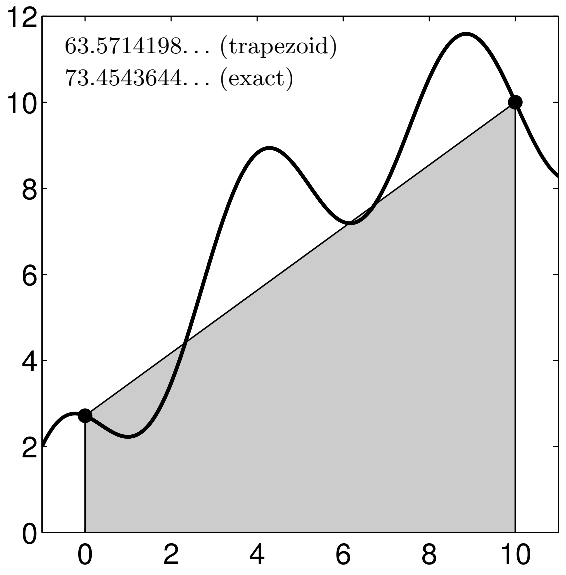
The integral of p_1 approximates the integral of f:

$$\begin{aligned}\int_a^b p_1(x)\,\mathrm{d}x &= \int_a^b \Big(f(a) + \frac{f(b)-f(a)}{b-a}\,(x-a)\Big)\,\mathrm{d}x \\ &= \Big[f(a)x\Big]_a^b + \Big(\frac{f(b)-f(a)}{b-a}\Big)\Big[\tfrac{1}{2}(x-a)^2\Big]_a^b \\ &= \frac{b-a}{2}(f(a)+f(b)).\end{aligned}$$

In summary,

$$\textbf{Trapezoid Rule:}\qquad \int_a^b f(x)\,\mathrm{d}x \approx \frac{b-a}{2}\Big(f(a)+f(b)\Big).$$

The procedure behind the trapezoid rule is illustrated in the following picture, where the area approximating the integral is colored gray.



Error bound for the trapezoid rule. To derive an error bound, we simply integrate the interpolation error bound developed in Lecture 11. That bound ensured that for each $x \in [a, b]$, there exists 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 $\xi(x)$ below. Integrating, we obtain

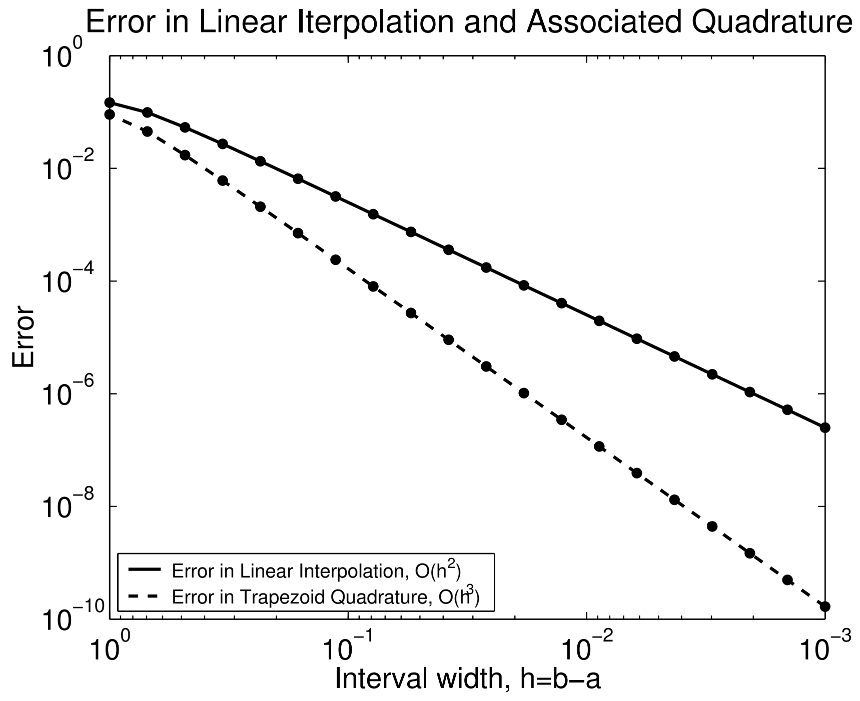
$$\int_a^b f(x)\,\mathrm{d}x - \int_a^b p_1(x)\,\mathrm{d}x = \int_a^b \tfrac{1}{2} f''(\xi(x))(x-a)(x-b)\,\mathrm{d}x$$

$$= \tfrac{1}{2}f''(\eta)\int_a^b (x-a)(x-b)\,\mathrm{d}x$$
$$= \tfrac{1}{2}f''(\eta)(\tfrac{1}{6}a^3 - \tfrac{1}{2}a^2b + \tfrac{1}{2}ab^2 - \tfrac{1}{6}b^3)$$
$$= -\tfrac{1}{12}f''(\eta)(b-a)^3$$

for some $\eta \in [a,b]$. The second step follows from the mean value theorem for integrals.[§]

(We shall develop a much more general theory from which we can derive this error bound, plus bounds for more complicated schemes, too, in Lecture 23b.)

Example: $f(x) = \mathrm{e}^x(\cos x + \sin x)$. 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. Our theory predicts that linear interpolation will have an $O(h^2)$ error as $h \to 0$, while the trapezoid rule has $O(h^3)$ error.



4.1.2 Simpson's rule. We expect that better accuracy can be attained by replacing the trapezoid rule's linear interpolant with a higher degree polynomial interpolant to f over $[a,b]$. This will increase the number of times we must evaluate 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 follows from using a quadratic approximation that interpolates f at a, b, and the midpoint $(a+b)/2$. If we use the Newton form of the interpolant with $x_0 = a$, $x_1 = b$, and

[§]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 $\eta \in [-1,1]$, $g(\eta)\int_{-1}^1 h(t)\,\mathrm{d}t = 0 \neq \int_{-1}^1 g(t)h(t)\,\mathrm{d}t = 2/3$.

$x_2 = (a+b)/2$, we obtain

$$p_2(x) = f(a) + \frac{f(b)-f(a)}{b-a}\,(x-a) + \frac{2f(a) - 4f(\frac{1}{2}(a+b)) + 2f(b)}{(b-a)^2}\,(x-a)(x-b).$$

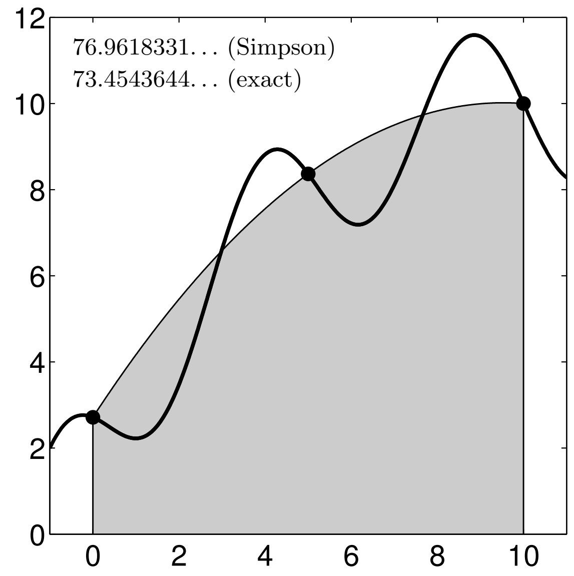
Simpson's rule then approximates the integral of f with the integral of p_2:

$$\begin{aligned}\int_a^b p_2(x)\,\mathrm{d}x &= \int_a^b p_1(x)\,\mathrm{d}x + \frac{2f(a) - 4f(\frac{1}{2}(a+b)) + 2f(b)}{(b-a)^2}\int_a^b (x-a)(x-b)\,\mathrm{d}x \\ &= \frac{b-a}{2}(f(a)+f(b)) + \frac{2f(a) - 4f(\frac{1}{2}(a+b)) + 2f(b)}{(b-a)^2}\,\frac{(b-a)^3}{6} \\ &= \frac{b-a}{6}\Big(f(a) + 4f(\tfrac{1}{2}(a+b)) + f(b)\Big),\end{aligned}$$

where we have used the fact that the first two terms of p_2 are identical to the linear approximation p_1 used above for the trapezoid rule. In summary:

Simpson's Rule: $$\int_a^b f(x)\,\mathrm{d}x \approx \frac{b-a}{6}\Big(f(a) + 4f(\tfrac{1}{2}(a+b)) + f(b)\Big).$$

The picture below shows an application of Simpson's rule on $[a,b]=[0,10]$.



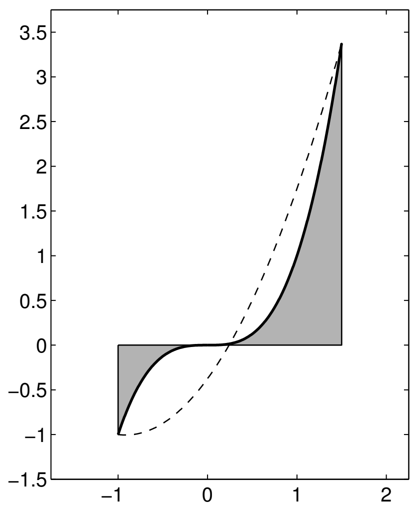
Error bound for Simpson's rule. Simpson's rule enjoys a remarkable feature: though it only approximates f by a quadratic, *it integrates any cubic polynomial exactly*! One can verify this by directly applying Simpson's rule to a generic cubic polynomial.[¶] In Lecture 23b, we shall derive the tools to compute an error bound for Simpson's rule:

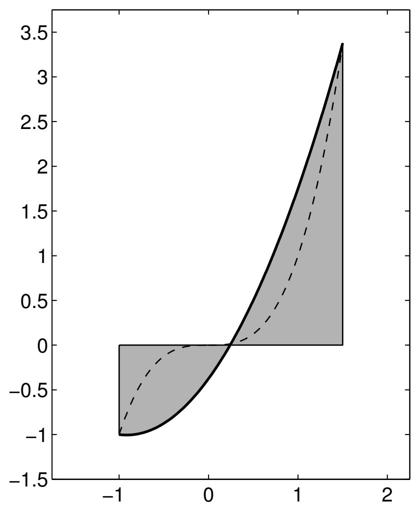
$$\int_a^b f(x)\,\mathrm{d}x - \int_a^b p_2(x)\,\mathrm{d}x = -\frac{1}{90}\,\frac{(b-a)^5}{2^5}\,f^{(4)}(\eta)$$

[¶]It turns out that Newton–Cotes formulas based on approximating f by an even-degree polynomial always exactly integrate polynomials one degree higher.

for some $\eta \in [a, b]$.

We emphasize that although Simpson's rule is exact for cubics, the interpolating polynomial we integrate really is quadratic. Though this should be clear from the discussion above, you might find it helpful to see this graphically. Both plots below show a cubic function f (solid line) and its quadratic interpolant (dashed line). On the left, the area under f is colored gray – its area is the integral we wish to compute. 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. It is remarkable that this is the case for *all* cubics.


exact integral


Simpson's rule

4.1.3 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 still further, then integrate that high-degree polynomial. As we learned in our study of polynomial interpolation, the success of such an approach depends significantly on the choice of the interpolation points. For example, we would not expect to get an accurate answer by integrating a high degree polynomial that interpolates Runge's function $f(x) = (x^2 + 1)^{-1}$ over uniformly spaced points on $[-5, 5]$.

One expects to get better results by integrating the interpolant to f at Chebyshev points. This procedure is known as *Clenshaw–Curtis quadrature.* The formulas get a bit intricate, but the results are fantastic if f is smooth (e.g., analytic in a region of the complex plane containing $[a, b]$).[‖]

4.1.4 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, and apply the trapezoid rule or Simpson's rule on each subinterval. Applying the trapezoid rule 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

[‖]See L. N. Trefethen, 'Is Gauss Quadrature Better than Clenshaw–Curtis?', *SIAM Review* 50 (2008) 67–87.

and b, $x_j = a + jh$ for $j = 0, \ldots, n$ and $h = (b-a)/n$, giving the following famous formulation:

$$\textbf{Composite Trapezoid:} \qquad \int_a^b f(x)\,\mathrm{d}x \;\approx\; \frac{h}{2}\Big(f(a) + 2\sum_{j=1}^{n-1} f(a+jh) + f(b)\Big).$$

(Of course, one can readily adjust this rule to cope with irregularly spaced points.) 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

$$f''(\eta) = \frac{1}{n}\sum_{j=1}^{n} f''(\eta_j).$$

The composite trapezoid error bound thus simplifies to

$$\int_a^b f(x)\,\mathrm{d}x - \frac{h}{2}\Big(f(a) + 2\sum_{j=1}^{n-1} f(a+jh) + f(b)\Big) = -\frac{h^2}{12}(b-a)f''(\eta).$$

This error analysis has an important consequence: *the error for the composite trapezoid rule is only* $O(h^2)$, not the $O(h^3)$ we saw for the usual trapezoid rule (in which case $b - a = h$ since $n = 1$).

Similar analysis can be performed to derive 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 formula

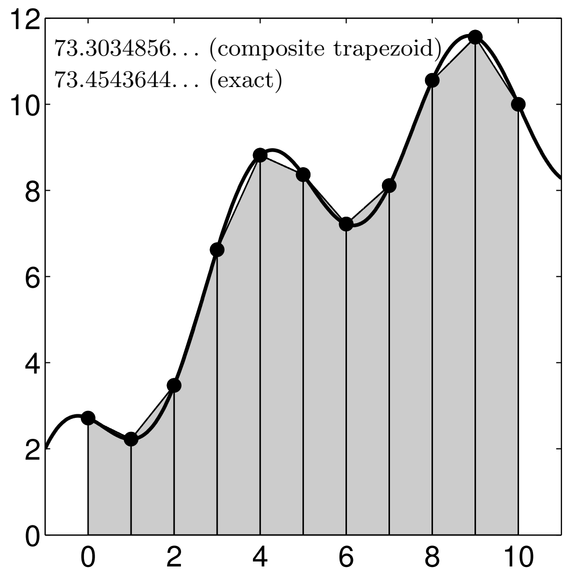
$$\textbf{Composite Simpson:} \quad \int_a^b f(x)\,\mathrm{d}x \;\approx\; \frac{h}{3}\Big(f(a) \;+\; 4\sum_{j=1}^{n/2} f(a+(2j-1)h) \;+\; 2\sum_{j=1}^{n/2-1} f(a+2jh) \;+\; f(b)\Big).$$

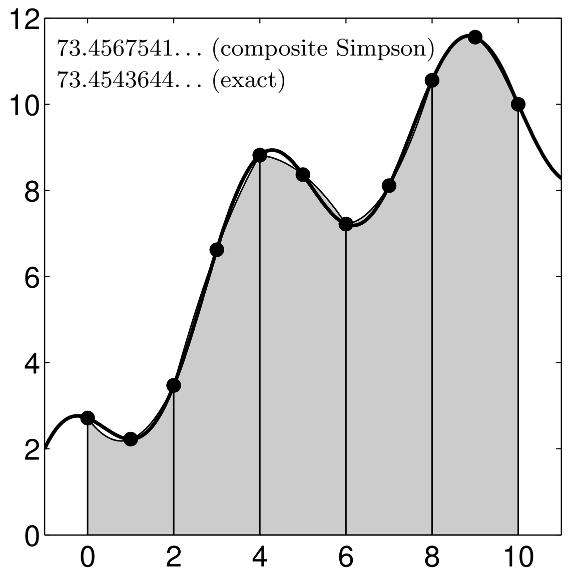
Derivation of the error formula for the composite Simpson's rule follows the same strategy as the analysis of the composite trapezoid rule. One obtains

$$\int_a^b f(x)\,\mathrm{d}x - \frac{h}{3}\Big(f(a) \;+\; 4\sum_{j=1}^{n/2} f(a+(2j-1)h) \;+\; 2\sum_{j=1}^{n/2-1} f(a+2jh) \;+\; f(b)\Big) = -\frac{h^4}{180}(b-a)f^{(4)}(\eta)$$

for some $\eta \in [a, b]$.

The illustrations below 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 the better accuracy.





Next we present a MATLAB script implementing the composite trapezoid rule; Simpson's rule is a straightforward modification that is left as an exercise. To get the standard (not composite) trapezoid rule, call `trapezoid` with `N=1`.

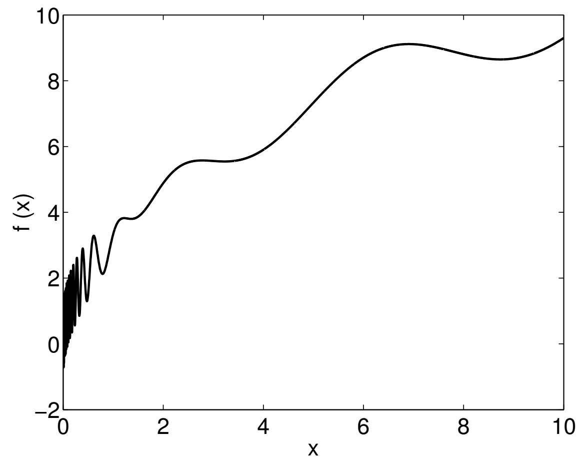
```
 function intf = trapezoid(f, a, b, N)
% Composite trapezoid rule to approximate the integral of f from a to b.
% Uses N+1 function evaluations (N>1; h=(b-a)/N).

 h = (b-a)/N; intf = 0;
 intf = (feval(f,a)+feval(f,b))/2;
 for j=1:N-1
    intf = intf + feval(f,a+j*h);
 end
 intf = intf*h;
```

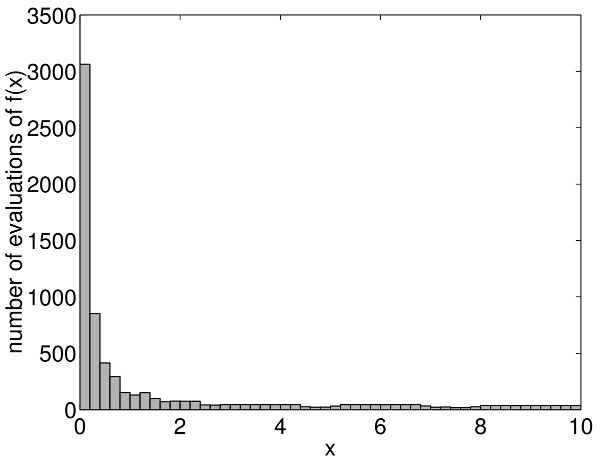
Pause a moment for reflection. Suppose you are willing to evaluate f a fixed number of times. How can you get the most bang for your buck? If f is smooth, a rule based on a high-order interpolant (such as Clenshaw–Curtis quadrature, or the Gaussian quadrature rules we will present in a few lectures) 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.*)

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.[**]

MATLAB's quadrature routines. The MATLAB quadrature routine `quad` implements an adaptive composite Simpson's rule. A different quadrature routine, `quadl`, uses Gaussian quadrature, which we shall talk about a few classes from now.

The following illustrations show MATLAB's adaptive quadrature rule at work. On the left, we have a function that varies smoothly over most of the domain, but oscillates wildly over 10% the region of interest. The plot on the right is a histogram of the number of function evaluations used by MATLAB's `quad`. Clearly, this routine uses many more function evaluations in the region where the function oscillates most rapidly (`quad` identified this region itself; the user only supplies the region of integration $[a, b]$).





[**]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 in version 6 as the routines `quad` and `quadl`.

Lecture 24: Richardson Extrapolation and 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., the trapezoid rule). 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 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.*

4.3. Richardson extrapolation.

We begin in a general setting: Suppose we wish to compute some abstract quantity, x_*, which could be an integral, a derivative, the solution to a differential equation at a certain point, or something else entirely. Further suppose we cannot compute x_* exactly; we can only access numerical approximations to it, generated by some function ϕ that depends upon a mesh parameter h. We compute $\phi(h)$ for several values of h, expecting that $\phi(h) \to \phi(0) = x_*$ 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 this: (1) often ϕ becomes increasingly expensive to evaluate as h shrinks;[†]; (2) the numerical accuracy with which we can evaluate ϕ may deteriorate as h gets small, due to rouding 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 $h \to 0$.)

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 $\phi(0) = x_*$, we can rewrite the Taylor series for $\phi(h)$ as

$$\phi(h) = x_* + c_1h + c_2h^2 + c_3h^3 + \cdots$$

for some constants $\{c_j\}_{j=1}^{\infty}$.

This expansion implies that taking $\phi(h)$ as an approximation for x_* incurs an $O(h)$ error. Halving the parameter h should roughly halve the error, according to the expansion

$$\phi(h/2) = x_* + c_1\tfrac{1}{2}h + c_2\tfrac{1}{4}h^2 + c_3\tfrac{1}{8}h^3 + \cdots.$$

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 $O(h)$ term. In particular, define

$$\psi(h) := 2\phi(h/2) - \phi(h)$$

[†]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 h^{-1} or h^{-2} or h^{-3}, etc.

$$
\begin{aligned}
&= 2\Big(x_* + c_1\tfrac{1}{2}h + c_2\tfrac{1}{4}h^2 + c_3\tfrac{1}{8}h^3 + \cdots\Big) - \Big(x_* + c_1 h + c_2 h^2 + c_3 h^3 + \cdots\Big)\\
&= x_* - c_2\tfrac{1}{2}h^2 - c_3\tfrac{3}{4}h^3 + \cdots .
\end{aligned}
$$

Thus, $\psi(h)$ also approximates $x_* = \psi(0) = \phi(0)$, but with an $O(h^2)$ error, rather than the $O(h)$ error that pollutes $\phi(h)$. For small h, this $O(h^2)$ approximation will be considerably more accurate.

Why stop with $\psi(h)$? Repeat the procedure, combining $\psi(h)$ and $\psi(h/2)$ to eliminate the $O(h^2)$ term. Since

$$\psi(h/2) = x_* - c_2\tfrac{1}{8}h^2 - c_3\tfrac{3}{32}h^3 + \cdots,$$

we have

$$\theta(h) := \frac{4\psi(h/2) - \psi(h)}{3} = x_* + c_3\tfrac{1}{8}h^3 + \cdots .$$

To compute $\theta(h)$, we need to compute both $\psi(h)$ and $\psi(h/2)$. These, in turn, require $\phi(h)$, $\phi(h/2)$, and $\phi(h/4)$. Usually, ϕ becomes increasingly expensive to compute as the mesh size 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

$$
\begin{aligned}
R(j,0) &:= \phi(h/2^j), && j \ge 0;\\
R(j,k) &:= \frac{2^k R(j,k-1) - R(j-1,k-1)}{2^k - 1}, && j \ge k > 0.
\end{aligned}
$$

Thus, for example, $R(0,0) = \phi(h)$, $R(1,0) = \phi(h/2)$, 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*:

$$
\begin{array}{ccccc}
R(0,0) & & & & \\
R(1,0) & R(1,1) & & & \\
R(2,0) & R(2,1) & R(2,2) & & \\
R(3,0) & R(3,1) & R(3,2) & R(3,3) & \\
\cdots & \cdots & \cdots & \cdots & \ddots \\
\uparrow & \uparrow & \uparrow & \uparrow & \\
O(h) & O(h^2) & O(h^3) & O(h^4) &
\end{array}
$$

To compute any given element in the table, one must first determine entries above and to the left. The only expensive computations are in the first column; the extrapolation procedure itself is simple arithmetic. We expect the bottom-right element in the table to be the most accurate approximation to x_*. (This will usually be the case, but we can run into trouble if our assumption that ϕ is infinitely continuously differentiable does not hold, e.g., where floating point roundoff errors spoil what otherwise would have been an infinitely continuously differentiable procedure. In this case, rounding errors that are barely noticeable in the first column destroy the accuracy of the bottom right entries in the extrapolation table.)

4.3.1. Example: Finite difference approximation of the first derivative.

To see the power of Richardson extrapolation, consider the finite difference approximation of the first derivative. Given some $f \in C^\infty[a,b]$, expand in Taylor series about the point $\alpha \in [a,b]$:

$$f(\alpha+h) = f(\alpha) + hf'(\alpha) + \tfrac{1}{2}h^2 f''(\alpha) + \tfrac{1}{6}h^3 f'''(\alpha) + \cdots.$$

This expansion can be rearranged to give a finite difference approximation to $f'(\alpha)$:

$$f'(\alpha) = \frac{f(\alpha+h)-f(\alpha)}{h} + O(h),$$

so we define

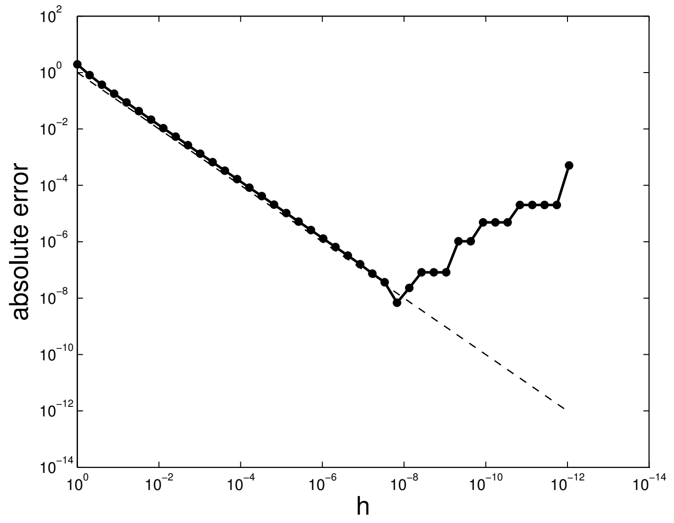
$$\phi(h) = \frac{f(\alpha+h)-f(\alpha)}{h}.$$

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}

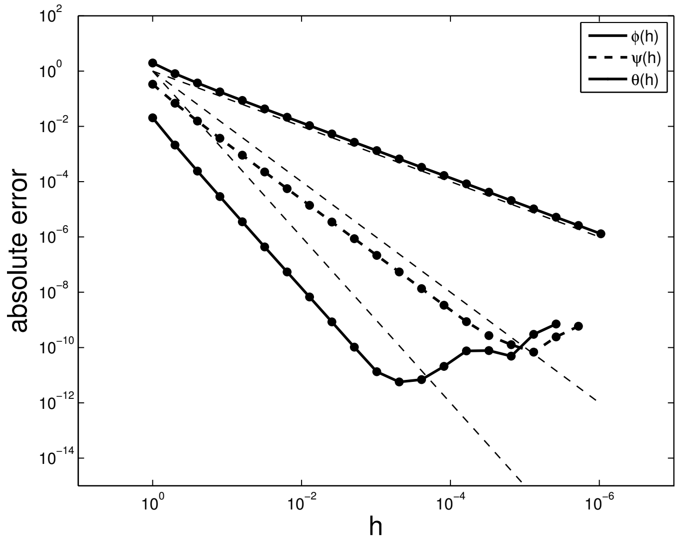
Even with $h = 1/512$ we fail to obtain three correct digits. As we take h smaller and smaller, finite precision arithmetic eventually causes unacceptable errors, as seen in the figure below showing the error in $\phi(h)$ as $h \to 0$. (The dashed line shows what perfect $O(h)$ convergence would look like.)



Yet a few steps of Richardson extrapolation on the above data reveals greatly improved solutions, five correct digits in $R(4,4)$:

$R(j,0)$	$R(j,1)$	$R(j,2)$	$R(j,3)$	$R(j,4)$
4.67077427047160				
3.52681448375804	$\underline{2}$.38285469704447			
3.08824451601118	$\underline{2}$.64967454826433	$\underline{2.7}$3861449867095		
$\underline{2}$.89548016367188	$\underline{2.7}$0271581133258	$\underline{2.7}$2039623235534	$\underline{2.71}$779362288168	
$\underline{2}$.80502585140344	$\underline{2.71}$457153913500	$\underline{2.718}$52344840247	$\underline{2.718}$25590783778	$\underline{2.71828}$672683485

The plot below illustrates how the rounding errors made in the initial data pollute the Richardson iterates before long. (For comparison, the dashed lines indicate what an exact error of $O(h)$, $O(h^2)$, and $O(h^3)$ would like.)



4.3.2. Extrapolation for higher order approximations.

In many cases, the initial algorithm $\phi(h)$ is better than $O(h)$ accurate, and in this case the formula for $R(j,k)$ should be adjusted to take advantage. Suppose that

$$\phi(h) = x_* + c_1 h^r + c_2 h^{2r} + c_3 h^{3r} + \cdots$$

for some integer $r \ge 1$. Then define

$$R(j,0) := \phi(h/2^j) \qquad \text{for } j \ge 0$$

$$R(j,k) := \frac{2^{rk}R(j,k-1) - R(j-1,k-1)}{2^{rk}-1} \qquad \text{for } j \ge k > 0.$$

In this case, the $R(:,k)$ column will be $O(h^{(k+1)r})$ accurate.

4.3.3. Extrapolating the composite trapezoid rule: Romberg integration.

Suppose $f \in C^\infty[a,b]$, and we wish to approximate $\int_a^b f(x)\,dx$ with the composite trapezoid rule,

$$T(h) = \frac{h}{2}\Big[f(a) + 2\sum_{j=1}^{n-1} f(a+jh) + f(b)\Big].$$

One can show that if $f \in C^\infty[a,b]$, then

$$T(h) = \int_a^b f(x)\,\mathrm{d}x + c_1h^2 + c_2h^4 + c_3h^6 + \cdots.$$

Now perform Richardson extrapolation on $T(h)$ with $r = 2$:

$$\begin{aligned} R(j,0) &= T(h/2^j) \qquad \text{for } j \ge 0 \\ R(j,k) &= \frac{4^kR(j,k-1) - R(j-1,k-1)}{4^k-1} \qquad \text{for } j \ge k > 0. \end{aligned}$$

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.

The significance of this procedure is best appreciated through an example. 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.

```
0.000000000000
1.570796326795   2.094395102393
1.896118897937   2.004559754984   1.998570731824
1.974231601946   2.000269169948   1.999983130946   2.000005549980
1.993570343772   2.000016591048   1.999999752455   2.000000016288   1.999999994587
1.998393360970   2.000001033369   1.999999996191   2.000000000060   1.999999999996   2.000000000001
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 the example shown here, 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.

MATLAB code implementing Romberg integration is shown on the next page.

[‡]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.

```
 function R = romberg(f, a, b, max_k)
% Compute the triangular extrapolation table for Romberg integration
% using the composite trapezoid rule, starting with h=b-a.
% f:     function name (either a name in quotes, or an inline function)
% a, b:  lower and upper limits of integration
% max_k: the number of extrapolation steps (= number of columns in R, plus one.)
%        max_k=0 will do no extrapolation.
% Example: R = romberg('sin',0,pi,1,6)

 R = zeros(max_k+1);
 for j=1:max_k+1
     R(j,1) = trapezoid(f,a,b,2^(j-1));
 end
 for k=2:max_k+1
    for j=k:max_k+1
        R(j,k) = (4^(k-1)*R(j,k-1)-R(j-1,k-1))/(4^(k-1)-1);
    end
 end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% demonstration of calling romberg.m
% based on integrating sin(x) from 0 to pi; the exact integral is 2.

 a = 0; b = pi;
 max_k = 6;
 R = romberg('sin',a,b,max_k);

% The ratio of the difference between successive entries in column k
% should be approximately 4^k.  This provides a simple test to see if
% our extrapolation is working, without requiring knowledge of the exact
% answer.  See Johnson and Riess, Numerical Analysis, 2nd ed., p. 323.

 Rat = zeros(max_k-1,max_k-1);
 for k=1:max_k-1
    Rat(k:end,k) = (R(1+k:end-1,k)-R(k:end-2,k))./(R(2+k:end,k)-R(1+k:end-1,k));
 end
```

Lecture 25: Gaussian Quadrature

4.4 Gaussian quadrature.

It is clear that the trapezoid rule,

$$I(f) = \frac{b-a}{2}(f(a)+f(b)),$$

exactly integrates linear polynomials, but not all quadratics. In fact, one can show that *no* quadrature rule of the form

$$I(f) = w_a f(a) + w_b f(b)$$

will exactly integrate all quadratics over $[a,b]$, regardless of the choice of constants w_a and w_b.

4.4.1. 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:

$$w_0 = w_1 = \tfrac{1}{2}(b-a), \qquad x_0 = \tfrac{1}{2}(b+a) - \tfrac{\sqrt{3}}{6}(b-a) \qquad x_1 = \tfrac{1}{2}(b+a) + \tfrac{\sqrt{3}}{6}(b-a).$$

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

$$I(f) = f(-1/\sqrt{3}) + f(1/\sqrt{3}).$$

4.4.2. 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

$$I(f) = \sum_{j=0}^{n} w_j f(x_j),$$

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 $I(f)$ will exactly integrate all polynomials of degree n. This assumption will play a key role in the coming development.

Orthogonal polynomials, introduced in Lecture 20, will play a prominent role in this exposition. 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. We wish to construct a quadrature rule of the form

$$I(f) = \sum_{j=0}^{n} w_j f(x_j) \approx \int_a^b f(x)w(x)\,\mathrm{d}x.$$

It is our aim to make $I(f)$ exact for all $p \in \mathcal{P}_{2n+1}$.

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

$$\begin{aligned}\int_a^b p(x)w(x)\,\mathrm{d}x &= \int_a^b \phi_{n+1}(x)q(x)w(x)\,\mathrm{d}x + \int_a^b r(x)w(x)\,\mathrm{d}x\\ &= \int_a^b r(x)w(x)\,\mathrm{d}x.\end{aligned}$$

The second step is another consequence that important basic fact, proved in Lecture 20, 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,

$$I(p) = \sum_{j=0}^{n} w_j p(x_j) = \sum_{j=0}^{n} w_j \phi_{n+1}(x_j)q(x_j) + \sum_{j=0}^{n} w_j r(x_j)$$

$$= \sum_{j=0}^{n} w_j \phi_{n+1}(x_j) q(x_j) + \int_a^b r(x) w(x)\, \mathrm{d}x.$$

This last statement is a consequence of the fact that $I(\cdot)$ 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 $I(r)$ will be exact.)

Notice that we can force agreement between $I(p)$ 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 $2n+1$.

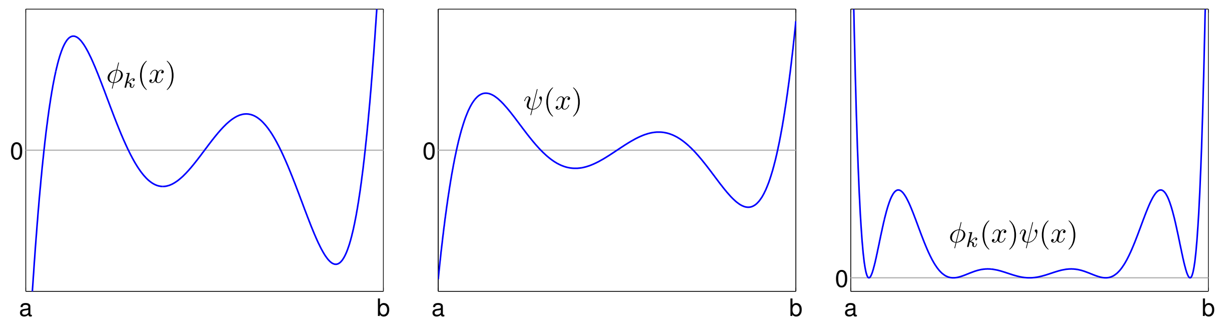
Before we can declare victory, though, we must exercise some caution. Perhaps ϕ_{n+1} has repeated roots (so that the nodes $\{x_j\}$ are not distinct), or perhaps these roots lie at points in the complex plane where f may not even be defined. Since we are integrating f over the interval $[a,b]$, it 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 (Roots of Orthogonal Polynomials). Let $\{\phi_k\}_{k=0}^{n}$ 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. 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

$$\psi(x) = (x - x_1^{(k)})(x - x_2^{(k)}) \cdots (x - x_j^{(k)}) \in \mathcal{P}_j.$$

This function changes sign at exactly the same points as ϕ_k does on $[a,b]$. Thus, the product of these two functions, $\phi_k\psi$, *does not change sign* on $[a,b]$. See the illustration below.



As the weight function $w(x)$ is nonnegative on $[a,b]$, it must also be that $\phi_k\psi w$ does not change sign on $[a,b]$. However, the fact that $\psi \in \mathcal{P}_j$ for $j < k$ implies that

$$\int_a^b \phi_k(x)\psi(x)w(x)\,\mathrm{d}x = 0,$$

since ϕ_k is orthogonal to all polynomials of degree $k-1$or lower. Thus, we conclude that the integral of some continuous nonzero function $\phi_k\psi w$ 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,

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

Integrating this interpolant gives

$$I(f) = \int_a^b p_n(x)w(x)\,\mathrm{d}x = \int_a^b \sum_{j=0}^{n} f(x_j)\ell_j(x)w(x)\,\mathrm{d}x = \sum_{j=0}^{n} f(x_j)\int_a^b \ell_j(x)w(x)\,\mathrm{d}x,$$

revealing a formula for the quadrature weights:

$$w_j = \int_a^b \ell_j(x)w(x)\,\mathrm{d}x.$$

(There are better ways to compute these weights, but the values will be the same. In practice, one solves a symmetric tridiagonal eigenvalue problem with to get the nodes and weights.) By our construction, we have proved the following result.

Theorem. Suppose $I(f)$ is the Gaussian quadrature rule

$$I(f) = \sum_{j=0}^{n} w_j f(x_j),$$

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 $w(x)$, and $w_j = \int_a^b \ell_j(x)w(x)\,\mathrm{d}x$. Then $I(f)$ is exact for all polynomials of degree $2n+1$.

Of course, in many circumstances we are not simply integrating polynomials, but more complicated functions. For that common situation, we have the following error bound, which we state without proof (see Süli and Mayers, pp. 282–283).

Theorem. Suppose $f \in C^{2n+2}[a,b]$ and let $I(f)$ 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

$$\int_a^b f(x)w(x)\,\mathrm{d}x - I(f) = \frac{f^{(2n+2)}(\xi)}{(2n+2)!}\int_a^b \psi^2(x)w(x)\,\mathrm{d}x$$

for some $\xi \in [a,b]$ and $\psi(x) = \prod_{j=0}^n (x - x_j)$.

Lecture 26: More on Gaussian Quadrature [draft]

4.4.3. Examples of Gaussian Quadrature.

Gauss–Legendre quadrature. The best known Gaussian quadrature rule integrates functions over the interval $[-1, 1]$ with the trivial weight function $w(x) = 1$. As we saw in Lecture 19, the orthogonal polynomials for this interval and weight are called *Legendre polynomials*. To construct a Gaussian quadrature rule with $n + 1$ points, we must determine the roots of the degree-$(n + 1)$ Legendre polynomial, then find the associated weights.

First, consider the case of $n = 1$. The quadratic Legendre polynomial is

$$\phi_2(x) = x^2 - 1/3,$$

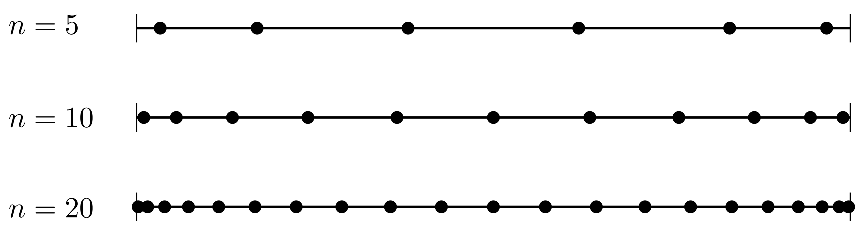
and from this polynomial one can derive the 2-point quadrature rule that is exact for cubic polynomials, with roots $\pm 1/\sqrt{3}$. This agrees with the special 2-point rule derived in Section 4.4.1. The values for the weights follow simply, $w_0 = w_1 = 1$, giving the 2-point Gauss–Legendre rule

$$I(f) = f(-1/\sqrt{3}) + f(1/\sqrt{3}).$$

For Gauss–Legendre quadrature rules based on larger numbers of points, there are various ways to compute the nodes and weights. Traditionally, one would consult a book of mathematical tables, such as the venerable *Handbook of Mathematical Functions*, edited by Milton Abramowitz and Irene Stegun for the National Bureau of Standards in the early 1960s. Now, one can look up these quadrature nodes and weights on web sites, or determine them to high precision using a symbolic mathematics package such as Mathematica. Most effective of all, one can compute these nodes and weights by solving a symmetric tridiagonal eigenvalue problem. See Trefethen and Bau, Lecture 37, for details. When $n = 5$, we obtain (from Mathematica)

j	nodes, x_j	weights, w_j
0	-0.9324695142031520	0.1713244923791703
1	-0.6612093864662645	0.3607615730481386
2	-0.2386191860831969	0.4679139345726910
3	0.2386191860831969	0.4679139345726910
4	0.6612093864662645	0.3607615730481386
5	0.9324695142031520	0.1713244923791703

Notice that the Gauss–Legendre nodes by no means uniformly distributed: like Chebyshev points for optimal interpolation, Legendre points for optimal quadrature cluster near the ends, as seen below (computed with Trefethen's `gauss.m` from *Spectral Methods in MATLAB*).



Given these values, we can implement the associated Gauss-Legendre quadrature rule with the following MATLAB code. (This includes a change of variables technique, described in Section 4.4.4 below, to allow integration over general intervals $[a, b]$, rather than $[-1, 1]$.)

```
  function intf = guasslengendre(f,a,b)
% Approximate the integral of f from a to b using a 6-point Gauss-Legendre rule.
% f is either the name of a function file, or an inline function.

 nodes   = [-0.9324695142031520; -0.6612093864662645; -0.2386191860831969;
             0.2386191860831969;  0.6612093864662645;  0.9324695142031520];
 weights = [ 0.1713244923791703;  0.3607615730481386;  0.4679139345726910;
             0.4679139345726910;  0.3607615730481386;  0.1713244923791703];

% change of variables from [-1,1] to [a,b]
 ab_nodes   = a + (b-a)*(nodes+1)/2;
 ab_weights = weights*(b-a)/2;

% apply Guass-Legendre rule
 intf = sum(ab_weights.*feval(f,ab_nodes));     % requires f to work for vectors
```

We can test this code on $\int_0^1 x^j\,\mathrm{d}x = 1/(j+1)$. Since this quadrature rule uses $n+1 = 6$ points, it should be exact for polynomials of degree $2n+1 = 11$. Indeed, that is what we see (up to rounding error) in the table below: There are negligible errors for $j = 1, \ldots, 11$, but at $j = 12$, suddenly some significant error appears. Even this discrepancy is fairly small, remarkable given that we are evaluating the integrand at only six points.

```
  j      Gauss-Legendre           error
---------------------------------------------------
  1   0.49999999999999994   0.00000000000000006
  2   0.33333333333333326   0.00000000000000006
  3   0.24999999999999994   0.00000000000000006
  4   0.20000000000000001   0.00000000000000000
  5   0.16666666666666663   0.00000000000000003
  6   0.14285714285714285   0.00000000000000000
  7   0.12499999999999999   0.00000000000000001
  8   0.11111111111111110   0.00000000000000000
  9   0.09999999999999999   0.00000000000000001
 10   0.09090909090909090   0.00000000000000001
 11   0.08333333333333331   0.00000000000000001
 12   0.07692298682558422   0.00000009009749270
 13   0.07142798579486889   0.00000058563370253
 14   0.06666454418815693   0.00000212247850974
 15   0.06249433000097107   0.00000566999902893
```

Gauss–Chebyshev quadrature. Another popular class of Gaussian quadrature rules use as their nodes the roots of the Chebyshev polynomials. Recall that the degree-k Chebyshev polynomial is defined as

$$T_k(x) = \cos(k \cos^{-1} x).$$

These are orthogonal polynomials 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

$$x_j = \cos\Big(\frac{(j+1/2)\pi}{n+1}\Big), \qquad j = 0,\ldots,n.$$

One can determine the associated weights to be

$$w_j = \frac{\pi}{n+1}$$

for all $j = 0,\ldots n$. (See Süli and Mayers, Problem 10.4 for a sketch of a proof.)

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

$$I(f) = \sum_{j=0}^{n} w_j f(x_j);$$

one *does not* include $1/\sqrt{1-x^2}$: the weight function is absorbed into the quadrature weights $\{w_j\}$.

Notice that the Chebyshev weight function blows up at ± 1, so if the integrand f doesn't 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 consult numerical tables or employ other fancy tools.

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 $p \in \mathcal{P}_{2n+1}$.

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

$$\int_{0}^{\infty} f(x)\mathrm{e}^{-x}\,\mathrm{d}x.$$

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

$$\int_{-\infty}^{\infty} f(x)\mathrm{e}^{-x^2}\,\mathrm{d}x.$$

4.4.4. Variations on the theme.

Change of variables. One notable drawback of Gaussian quadrature is the need to pre-compute (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 $[c, d]$.[†]

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. See the above MATLAB code for an implementation.) Such techniques can be used to develop Gaussian-based adaptive quadrature rules.

Gauss–Radau and Gauss-Lobatto quadrature. In applications, it is sometimes 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.

[†]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 $[0, 1]$.

Lecture 27: Introduction to ODE Solvers [draft]

5. Numerical Solution of Differential Equations.

The next segment of the course focuses on the numerical approximation of solutions to differential equations. The techniques of interpolation, approximation, and quadrature studied in previous sections are basic tools in your numerical tool chest, ones often used as building blocks to solve more demanding problems. Differential equations play a more central role; their (approximate) solution is required in nearly every corner of physics, chemistry, biology, engineering, finance, and beyond. For many practical problems (involving nonlinearities), there is no way to write down a closed-form solution to differential equations in terms of familiar functions such as polynomials, trigonometric functions, and exponentials. Thus the numerical solution of differential equations is an enormous field, with a great deal of effort in recent decades focused especially on partial differential equations (PDEs). In this course, we only have time to address the numerical solution of ordinary differential equations (ODEs) in detail, though we will briefly address PDEs at the end of this section (and on Problem Set 6).

The subject began in earnest with Leonhard Euler in 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) motivates modern innovations in so-called geometric/symplectic integrators.

5.0. Introduction to ordinary differential equation (ODE) initial value problems (IVPs).

Before computing numerical solutions to ODEs, it is important to understand the variety of problems that arise, and the theoretical conditions under which a solution exists.

5.0.1. Scalar equations.

A standard scalar initial value problem takes the form

$$\begin{aligned} \text{Given:}\quad & x'(t) = f(t,x), \text{ with } x(t_0) = x_0, \\ \text{Determine:}\quad & x(t) \text{ for all } t \ge t_0. \end{aligned}$$

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.

Some examples. Differential equations are an inherently graphical subject, so we should look at a few sample problems, together with plots of their solutions.

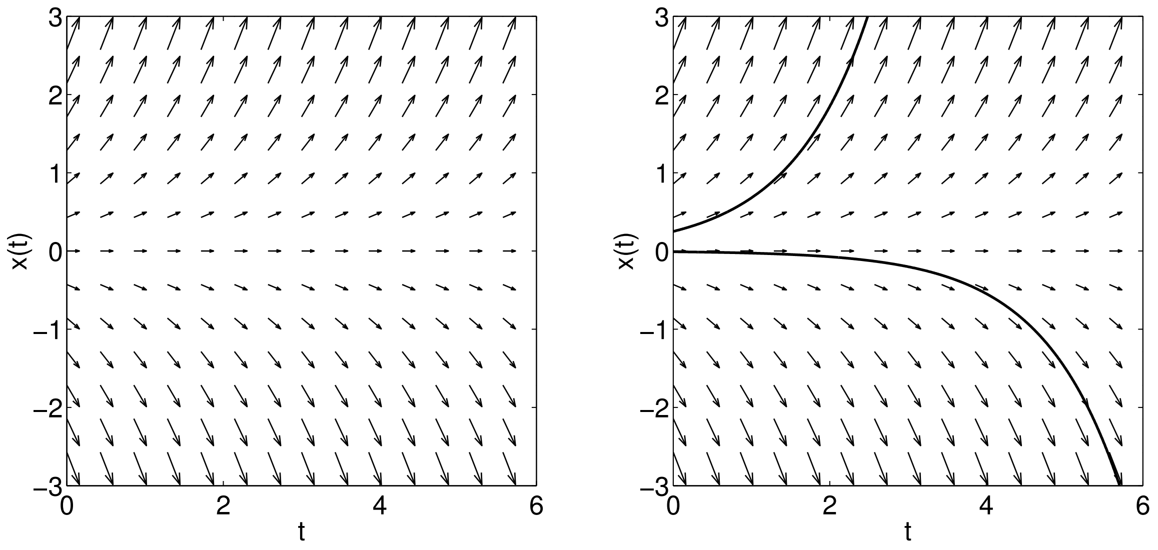
First we consider the simple, essential model problem

$$x'(t) = \lambda x(t),$$

with exact solution $x(t) = \alpha e^{\lambda t}$, where the constant α is derived from the initial data (t_0, 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 technique from the sub-discipline of the *qualitative analysis* of ODEs.[†] To draw a plot of 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.

The plot below shows the direction field for $f(t,x) = x$ on the left; on the right, we superimpose solutions of $x'(t) = x$ for several values of $x(0) = x_0$. (The increasing solution has $x(0) = 1/4$, while the decreasing solution has $x(0) = -1/100$.) Notice how the solutions follow the lines in the direction field.



It is a simple matter to compute direction fields in MATLAB using the `quiver` command. For example, the code below produced the plot on the left above.

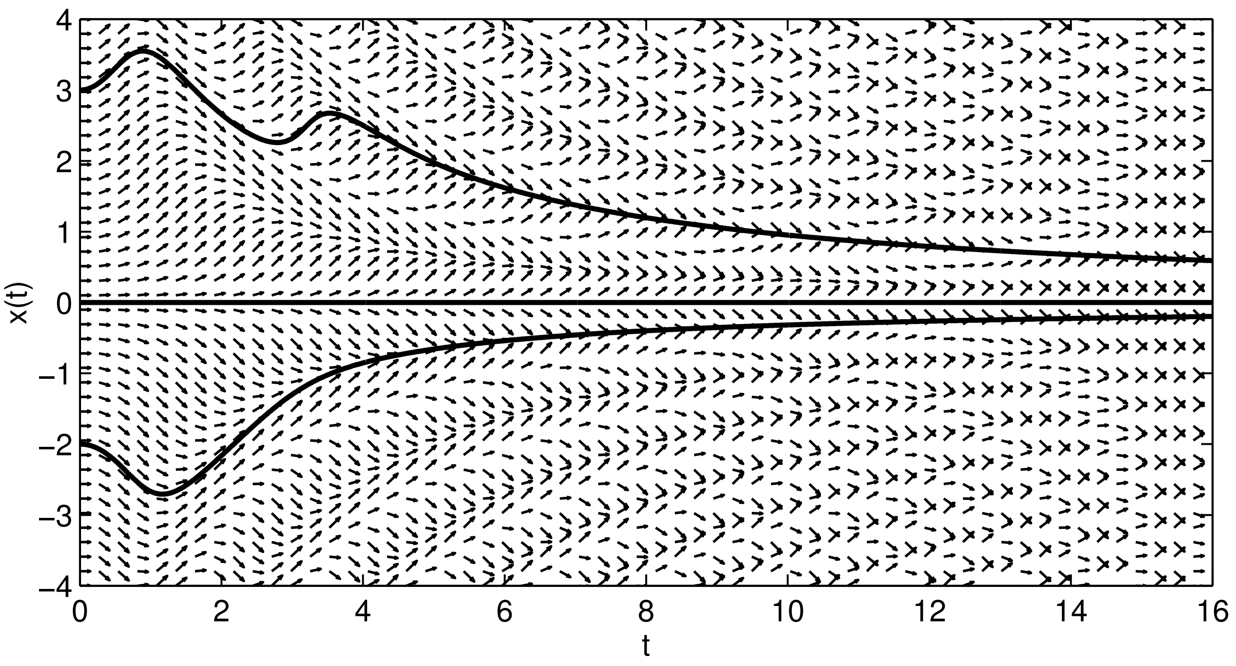
```
f = inline('x','t','x');                          % x' = f(t,x)
x = linspace(-3,3,15); t = linspace(0,6,15);      % grid of points at which to plot the slope
[T,X] = meshgrid(t,x);                            % turn grid vectors into matrices
PX = zeros(length(x),length(t));
for j=1:length(x)                                 % compute the slopes
   for k=1:length(t)
      PX(j,k) = f(t(k),x(j));
   end
end
figure(1), clf
quiver(T,X,ones(size(T)),PX), hold on             % produce a "quiver" plot
axis equal, axis([min(t) max(t) min(x) max(x)])
set(gca,'fontsize',20)
xlabel('t','fontsize',20)
ylabel('x(t)','fontsize',20)
```

[†]For an elementary introduction to this field, see J. H. Hubbard and B. H. West, *Differential Equations: A Dynamical Systems Approach, Part I*, Springer-Verlag, 1991.

Next consider an equation that lacks an elementary solution that can be expressed in closed form,

$$x'(t) = \sin(xt).$$

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 such solutions (for $x(0) = 3$, $x(0) = 0$, and $x(0) = -2$ are superimposed on the direction field. These were computed using Euler's method, which we will discuss momentarily. (Those areas, mainly in the right side of the direction field, where it appears that down and up arrows cross, are asymptotes of the solution: between the up and down arrow is a point where the slope $f(t, x)$ is zero.)



5.0.2. 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

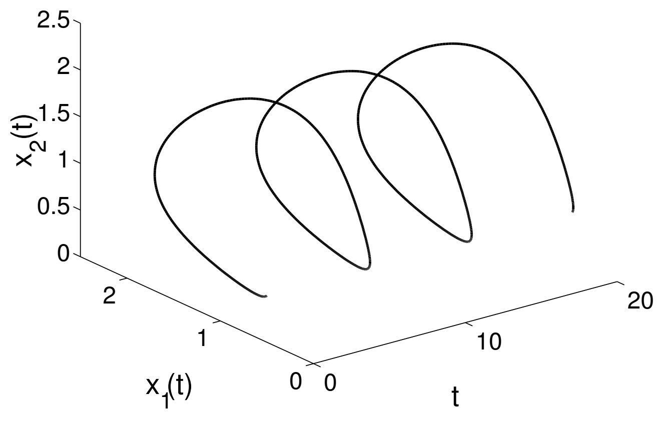
$$\begin{aligned} \text{Given:} &\quad \mathbf{x}'(t) = \mathbf{f}(t, \mathbf{x}), \text{ with } \mathbf{x}(t_0) = \mathbf{x}_0, \\ \text{Determine:} &\quad \mathbf{x}(t) \text{ for all } t \ge t_0. \end{aligned}$$

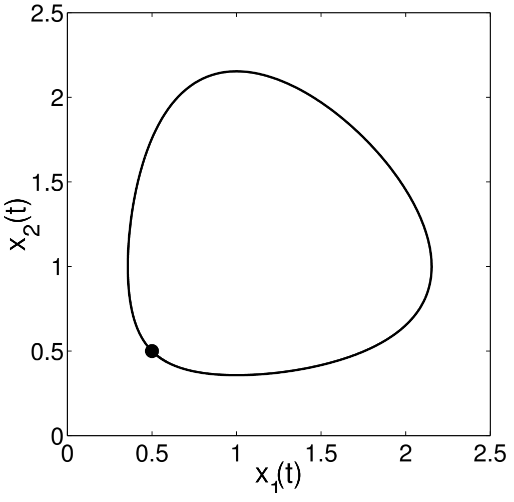
All the techniques for solving scalar initial value problems described in this course can be applied to systems of this type.

An example. Many nonlinear systems of ordinary differential equations give rise to fascinating behavior, widely studied in the field of *dynamical systems*. Many interesting systems are *autonomous* differential equations, meaning that the time variable t does not explicitly appear in the function $\mathbf{f}$. For example, the Lotka–Volterra predator–prey equations are given by

$$\mathbf{x}' = \begin{bmatrix} x_1' \\ x_2' \end{bmatrix} = \begin{bmatrix} x_1 - x_1x_2 \\ -x_2 + x_1x_2 \end{bmatrix} = \mathbf{f}(t, \mathbf{x}).$$

Here x_1 represents the population of prey (e.g., gazelles), while x_2 denotes the population of predators (e.g., lions). This system exhibits cyclic behavior, with the populations oscillating regularly in time. The plots below illustrate one such periodic solution based on the initial condition $x_1(0) = x_2(0) = 1/2$. The left plot shows how x_1 and x_2 evolve as a function of time. The *phase space* is shown on the right. For this plot, we stare down the time axis, seeing how x_1 and x_2 relate independent of time. The black dot denotes the system's position at $t = 0$.





5.0.3. Higher-order ODEs.

Many important ODEs arise from Newton's Second Law, $\mathbf{F}(t) = m\mathbf{a}(t)$. Noting the acceleration $\mathbf{a}(t)$ is the second derivative of position, 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

$$\begin{aligned} \text{Given:}\quad & x''(t) = f(t,x,x'), \text{ with } x(t_0) = x_0,\ x'(t_0) = y_0 \\ \text{Determine:}\quad & x(t) \text{ for all } t \geq t_0. \end{aligned}$$

Note, in particular, that the initial conditions $x(t_0)$ and $x'(t_0)$ must both be supplied.[‡]

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

$$\begin{aligned} x_1'(t) &= x'(t) = x_2(t) \\ x_2'(t) &= x''(t) = f(t,x,x') = f(t,x_1,x_2). \end{aligned}$$

Writing this in vector form, $\mathbf{x} = [x_1\ x_2]^{\mathrm{T}}$, and the differential equation becomes[§]

$$\mathbf{x}'(t) = \begin{bmatrix} x_1'(t) \\ x_2'(t) \end{bmatrix} = \begin{bmatrix} x_2(t) \\ f(t,x_1,x_2) \end{bmatrix} = \mathbf{f}(t,\mathbf{x}).$$

[‡]Problems where the initial data is given at two different t points are called *boundary value problems.* We will study them in Section 5.4.

[§]Note that fonts are important here: $x(t)$ is a scalar quantity, while $\mathbf{x}(t)$ is a vector.

The initial value is given by

$$\mathbf{x}_0 = \begin{bmatrix} x_1(t_0) \\ x_2(t_0) \end{bmatrix} = \begin{bmatrix} x(t_0) \\ x'(t_0) \end{bmatrix}.$$

An example. 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. This gives rise to the system

$$\mathbf{x}' = \begin{bmatrix} x_1' \\ x_2' \end{bmatrix} = \begin{bmatrix} x_2 \\ -x_1 \end{bmatrix}.$$

When we combine Newton's inverse-square description of gravitational force with his Second Law, we obtain 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.

5.0.4. 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. For a proof, see Süli and Mayers, Section 12.1.

Theorem (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 $|f(t,x_0)| \le K$ for all $t \in [t_0, t_{\text{final}}]$, and suppose there exists some *Lipschitz constant* $L > 0$ such that

$$|f(t,u) - f(t,v)| \le L\,|u - v|$$

for all $u, v \in [x_0 - c, x_0 + c]$ and all $t \in [t_0, t_{\text{final}}]$. Finally, suppose that

$$c \ge \frac{K}{L}\Big(\mathrm{e}^{L(t_{\text{final}} - t_0)} - 1\Big).$$

(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 $|x(t) - x_0| \le c$ 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 $t \in [t_0, t_{\text{final}}]$.

5.1. One-step methods.

Finally, 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

$$x'(t) = f(t, x)$$

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. (Actually, the methods we consider in this subsection actually 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:

$$x_0 = x(t_0).$$

5.1.1. 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. Recall from introductory calculus 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 our small but finite time step h 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 have access to the quantity on the left hand side of this approximation. The only quantity we don't know is $x(t_0 + h) = x(t_1)$. Rearranging the above to put $x(t_1)$ on the left hand side, we obtain

$$x(t_1) \approx x(t_0) + hx'(t_0) = x_0 + hf(t_0, x_0).$$

This approximation is precisely the one suggested by the direction field discussion in §5.0.1. There, to progress from the starting point (t_0, x_0), we followed the line of slope $f(t_0, x_0)$ some distance,

¶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.

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

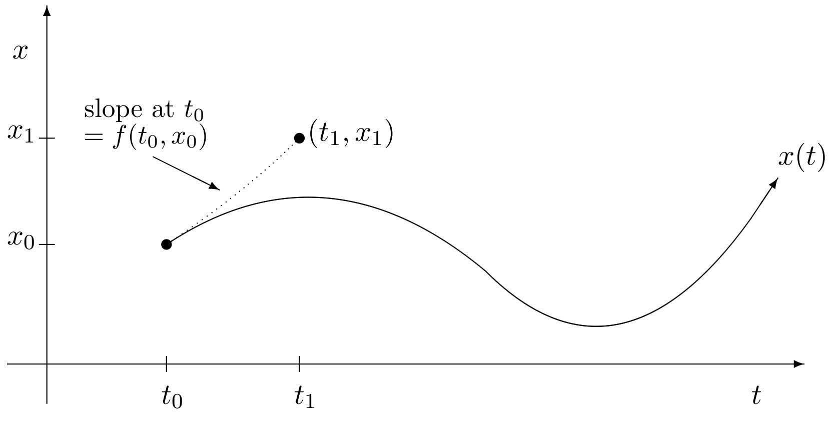
$$x(t_2) \approx x(t_1) + hf(t_1, x(t_1)).$$

However, an error was made in the computation of $x_1 \approx x(t_1)$; we do not have access to the exact value $x(t_1)$. Thus, compute x_2 from x_1, the quantity we have access to. 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 *global error*. Is a small local error enough to ensure small global error? This question is the subject of the next two lectures.

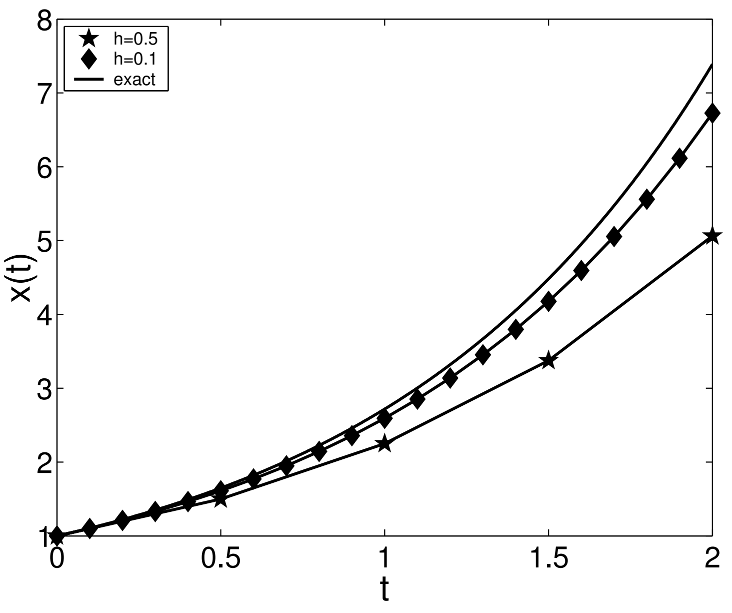
Given the approximation x_2, repeat the same procedure to obtain x_3, and so on. Formally,

$$\text{Euler's Method:} \qquad x_{k+1} = x_k + hf(t_k, x_k).$$

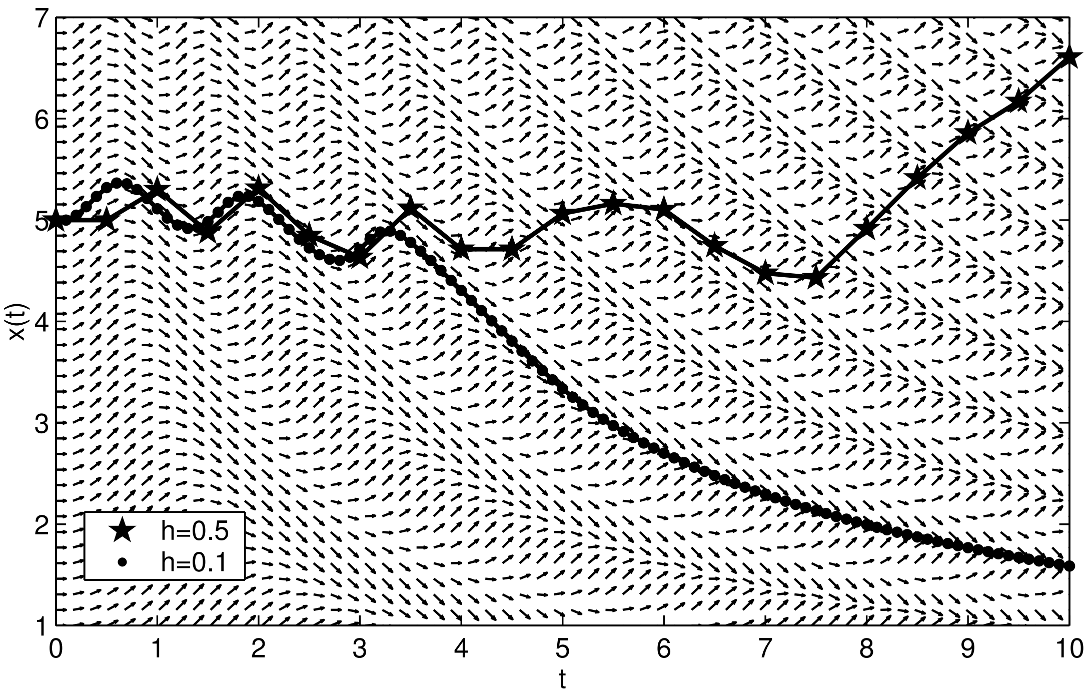
The first step of Euler's method is illustrated in the following schematic.



Examples of Euler's method. Consider the performance of Euler's method on the two examples for §5.0.1. 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 deliver 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? We see in the plot below that both approximations underestimate the true solution, but that indeed, the smaller step size yields the better approximation.



Next, consider our second example, $x'(t) = \sin(tx)$, 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.



For a final example of Euler's method, consider the equation

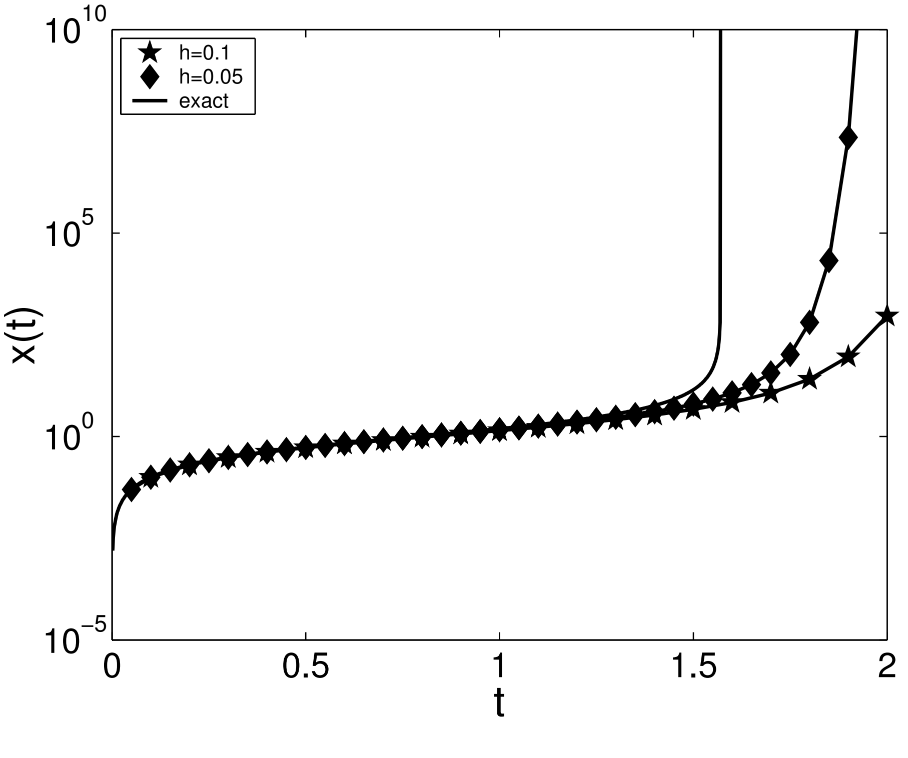
$$x'(t) = 1 + x(t)^2$$

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.

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}.)



Below, we present MATLAB code to implement Euler's method, with sample code illustrating how to call the routine for the second example, $x'(t) = \sin(tx)$.

[‖]This example is given in Kincaid and Cheney, page 525.

```
 function [t,x] = euler(xprime, tspan, x0, h)

% function [t,x] = euler(xprime, [t0 tfinal], x0, h)
% Approximate the solution to the ODE:  x'(t) = xprime(x,t)
% from t=t0 to t=tfinal with initial condition x(t0)=x0.
% xprime should be a function that can be called like: xprime(t,x).
% h is the step size:  reduce h to improve the accuracy.
% The ODE can be a scalar or vector equation.

 t0 = tspan(1); tfinal = tspan(end);

% set up the t values at which we will approximate the solution
 t=[t0:h:tfinal];

% include tfinal even if h does not evenly divide tfinal-t0
 if t(end)~=tfinal, t = [t tfinal]; end

% execute Euler's method
 x = [x0 zeros(length(x0),length(t)-1)];
 for j=1:length(t)-1
    x(:,j+1) = x(:,j) + h*feval(xprime,t(j),x(:,j));
 end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Sample code to call euler.m for the equation x'(t) = sin(t*x).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 xprime = inline('sin(x*t)','t','x');      % x'(t) = sin(t*x)
 tspan = [0 10];                           % integrate from t=0 to t=10
 x0 = 5;                                   % start with x(0) = 5
 h = 0.1;                                  % time step
 [t,x] = euler(xprime, tspan, x0, h);      % call Euler
 figure(1), clf
 plot(t,x,'b.','markersize',15)            % plot output
```

Lecture 28: Analysis of One Step ODE Integrators

5.1.2. 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 this method was derived from a first-order approximation to the derivative, we might expect the error to decay linearly in h. Before making this rigorous, let us first think about 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. The first modification we present may 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.

One can improve upon both Euler methods by averaging the updates they make to x_k:

$$x_{k+1} = x_k + \tfrac{1}{2}h\Big(f(t_k, x_k) + f(t_{k+1}, x_{k+1})\Big),$$

This method is the *trapezoid rule*, 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, $x(t_{k+1}) - x(t_k)$, which can be approximated by $x_{k+1} - x_k$. Rearranging these equations results in the trapezoid rule 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, one can 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 $f(t, x)$.

We must address an important consideration: the greater a method's accuracy, the more evaluations of the function f per step are required. In real applications, it is often computationally expensive to evaluate that function f. Thus one is forced to 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.

The *modified Euler method* takes a similar approach to Heun's method:

$$x_{k+1} = x_k + hf(t_k + \tfrac{1}{2}h, x_k + \tfrac{1}{2}hf(t_k, x_k)),$$

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:

$$x_{k+1} = x_k + \tfrac{1}{6}h\Big(k_1 + 2k_2 + 2k_3 + k_4\Big),$$

where

$$\begin{aligned} k_1 &= f(t_k, x_k) \\ k_2 &= f(t_k + \tfrac{1}{2}h, x_k + \tfrac{1}{2}hk_1) \\ k_3 &= f(t_k + \tfrac{1}{2}h, x_k + \tfrac{1}{2}hk_2) \\ k_4 &= f(t_k + h, x_k + hk_3). \end{aligned}$$

One often encounters this method implemented as a subroutine called `RK4` in old FORTRAN codes.

5.1.3. Truncation Error.

All explicit one-step methods can be written in the general form

$$x_{k+1} = x_k + h\Phi(t_k, x_k; h).$$

For such methods, we wish to understand 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.*

We will now 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 $x(t_k)$. We wish to bound the *global error*, defined simply as

$$e_k := x(t_k) - x_k.$$

Our goal is to understand this error as a function of the step size h.

To analyze the global error e_k, we 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.

Definition. The *truncation error* of a one-step ODE integrator is defined as

$$T_k = \frac{x(t_{k+1}) - x(t_k)}{h} - \Phi(t_k, x(t_k); h).$$

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 our method makes in a single step if we give it perfect data, $x(t_k)$.

Truncation error for Euler's method.

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 value $x(t_k)$. (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 $T_k = O(h)$, 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 $O(h^2)$ truncation error, while the error for the four-stage Runge–Kutta method is $O(h^4)$. Extrapolating from this data, one might expect that a method requiring m evaluations of f can deliver $O(h^m)$ 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 $O(h^5)$ truncation error require *at least six* function evaluations. We will see later how such high order methods can be used to automatically adjust the step-size h at each iteration.

Before addressing such step-adjustment, there remains a more fundamental question to address: Does $T_k \to 0$ as $h \to 0$ ensure global convergence, $e_k \to 0$?

Lecture 29: Global Error Analysis

5.1.4. Global Error Analysis for 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 Lecture 27. 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

$$|\Phi(t, u; h) - \Phi(t, v; h)| \le L_\Phi |u - v|$$

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 Lecture 27. For 'nice' ODEs and reasonable methods Φ, this condition is not difficult to satisfy.

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:

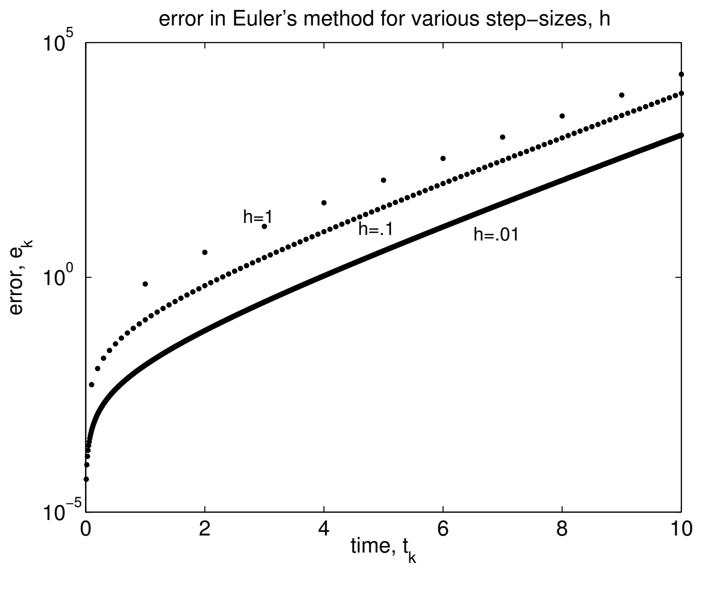
$$\begin{aligned}
|e_0| &= |x(t+0) - x_0| = 0 \\
|e_1| &\le h|T_0| \le hT \\
|e_2| &\le |e_1|(1 + hL_\Phi) + h|T_1| \le hT(1 + hL_\Phi) + hT \\
|e_3| &\le |e_2|(1 + hL_\Phi) + h|T_2| \le hT(1 + hL_\Phi)^2 + hT(1 + hL_\Phi) + hT \\
&\;\;\vdots \\
|e_n| &\le hT\sum_{k=0}^{n-1}(1 + hL_\Phi)^k.
\end{aligned}$$

Notice that this bound for $|e_n|$ is a finite geometric series, and thus we have the convenient formula

$$\begin{aligned}
|e_n| &\le hT\Big(\frac{(1 + hL_\Phi)^n - 1}{(1 + hL_\Phi) - 1}\Big) \\
&= \frac{T}{L_\Phi}((1 + hL_\Phi)^n - 1) \\
&\le \frac{T}{L_\Phi}(\mathrm{e}^{nhL_\Phi} - 1).
\end{aligned}$$

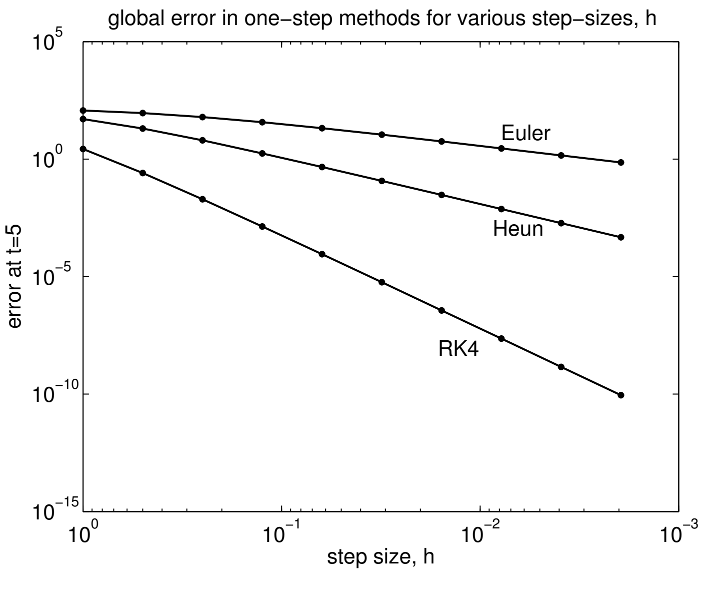
Here we have used the fact that the Taylor's series for e^γ implies that $1 + \gamma \le \mathrm{e}^\gamma$ for all $\gamma \ge 0$, with good agreement when $0 \le \gamma \ll 1$. (This result and proof are given as Theorem 12.2 in Süli and Mayers.)

There are two key lessons to be learned from this bound on $|e_n|$. First the bad news: As $n \to \infty$ and h is fixed, we expect the approximations from the one-step method to *exponentially* drift away from the true solution. This fact is illustrated in the plot below, where Euler's method has been applied to the model problem $x'(t) = x(t)$ for $t \in [0, 10]$ with various step sizes h.



However, this plot also hints at the good news to come: At any fixed time (say, t_{final}), the error gets smaller with decreasing h: this is the essential lesson to draw from this global error analysis. In typical situations, we are interested in the convergence of the global error at some fixed time t_{final} as the step size is reduced, $h \to 0$. In that case, set $h = (t_{\text{final}} - t_0)/n$, implying that $hn = t_{\text{final}} - t_0$ is fixed. Since L_Φ is independent of the step size h, if the truncation error converges, $T \to 0$ as $h \to 0$, then the global error at t_{final} will also converge. Moreover, if $T_k = O(h^p)$, then the global error at t_{final} will also be $O(h^p)$. This is a beautiful fact: the global error reduces at the same rate as the truncation error for one-step methods!

The plot below confirms this observation. Again for the model problem $x'(t) = x(t)$ with $(t_0, x_0) = (0, 1)$, we investigate convergence of Euler's method ($T_k = O(h)$), Heun's method ($T_k = O(h^2)$), and the four-stage Runge–Kutta method ($T_k = O(h^4)$) at the fixed time $t_{\text{final}} = 5$.



Note that the slopes of these global error curves agree with the order of the truncation error for each method – just as predicted by our global error analysis.

5.1.5. Adaptive Time-Step Selection.

One-step methods make it very to change the time-step h at each iteration. For complicated nonlinear problems, it is quite natural that 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 $O(h^5)$ 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 $O(h^4)$ 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 $O(h^4)$ method when an $O(h^5)$ 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 $O(h)$, $O(h^2)$, $O(h^3)$, and $O(h^4)$ approximations.

Lecture 30: Linear Multistep Methods: Truncation Error

5.2. Linear multistep methods.

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 $O(N^2)$ arithmetic operations to compute, costly indeed when N is in the millions. Every f evaluation counts.[†]

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 iteration (as is the case, for example, with higher order Runge–Kutta methods). Consider the method

$$x_{k+1} = x_k + h(\tfrac{3}{2}f(t_k,x_k) - \tfrac{1}{2}f(t_{k-1},x_{k-1})),$$

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 iteration 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 $O(h^2)$. 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: it is difficult to adjust the step size, 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.

5.2.1. General linear multistep methods.

This section considers a general class of integrators known as *linear multistep methods*.[‡]

Definition. A 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 $|\alpha_0| + |\beta_0| \neq 0$,

[†]A landmark improvement to this N^2 approach, the *fast multipole method*, was developed by Leslie Greengard and Vladimir Rokhlin in the late 1980s. Rokhlin is a CAAM alumnus (Ph.D.), and is now a professor at Yale.

[‡]These notes on linear multistep methods draw heavily from the excellent presentation in Endre Süli and David F. Mayers, *Numerical Analysis: An Introduction*, Cambridge University Press, 2003.

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

$$f_j = f(t_j, x_j).$$

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:	$x_{k+1} - x_k = hf_k$	$\alpha_0 = -1,\ \alpha_1 = 1,\ \beta_0 = 1,\ \beta_1 = 0.$
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

$$x_{k+4} = x_{k+3} + \frac{h}{24}\Big(55f_{k+3} - 59f_{k+2} + 37f_{k+1} - 9f_k\Big).$$

Here $\alpha_0 = \alpha_1 = \alpha_2 = 0$, $\alpha_3 = -1$, $\alpha_4 = 1$; $\beta_0 = -\frac{9}{24}$, $\beta_1 = \frac{37}{24}$, $\beta_2 = -\frac{59}{24}$, $\beta_3 = \frac{55}{24}$, and $\beta_4 = 0$.

The *Adams–Moulton* methods are a parallel class of *implicit* formulas. The 3-step version of this method is

$$x_{k+3} = x_{k+2} + \frac{h}{24}\Big(9f_{k+3} + 19f_{k+2} - 5f_{k+1} + f_k\Big).$$

Here $\alpha_0 = \alpha_1 = 0$, $\alpha_2 = -1$, $\alpha_3 = 1$; $\beta_0 = \frac{1}{24}$, $\beta_1 = -\frac{5}{24}$, $\beta_2 = \frac{19}{24}$, and $\beta_3 = \frac{9}{24}$.

5.2.2. Truncation error for 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:

$$T_k = \frac{\sum_{j=0}^{m}\Big[\alpha_j\, x(t_{k+j}) - h\,\beta_j\, f(t_{k+j}, x(t_{k+j}))\Big]}{h\sum_{j=0}^{m}\beta_j}.$$

(The $\sum_{j=0}^{m}\beta_j$ term in the denominator is a normalization term; if it were absent, then multiplying the entire multistep formula by a constant would alter the truncation error, but the not the iterates x_j.) In order to get a simple form of the truncation error, we turn to Taylor series:

$$\begin{array}{rcccccccccccc}
x(t_{k+1}) = & x(t_k+h) & = & x(t_k) & + & hx'(t_k) & + & \frac{h^2}{2!}x''(t_k) & + & \frac{h^3}{3!}x'''(t_k) & + & \frac{h^4}{4!}x^{(4)}(t_k) & + \cdots \\
x(t_{k+2}) = & x(t_k+2h) & = & x(t_k) & + & 2hx'(t_k) & + & \frac{2^2h^2}{2!}x''(t_k) & + & \frac{2^3h^3}{3!}x'''(t_k) & + & \frac{2^4h^4}{4!}x^{(4)}(t_k) & + \cdots \\
x(t_{k+3}) = & x(t_k+3h) & = & x(t_k) & + & 3hx'(t_k) & + & \frac{3^2h^2}{2!}x''(t_k) & + & \frac{3^3h^3}{3!}x'''(t_k) & + & \frac{3^4h^4}{4!}x^{(4)}(t_k) & + \cdots \\
 & & \vdots & & & & & & & & & & \\
x(t_{k+m}) = & x(t_k+mh) & = & x(t_k) & + & mhx'(t_k) & + & \frac{m^2h^2}{2!}x''(t_k) & + & \frac{m^3h^3}{3!}x'''(t_k) & + & \frac{m^4h^4}{4!}x^{(4)}(t_k) & + \cdots
\end{array}$$

and also

$$\begin{array}{rcccccccccl}
f(t_{k+1},x(t_{k+1})) = x'(t_k+h) &=& x'(t_k) &+& hx''(t_k) &+& \frac{h^2}{2!}x'''(t_k) &+& \frac{h^3}{3!}x^{(4)}(t_k) &+& \cdots\\
f(t_{k+2},x(t_{k+2})) = x'(t_k+2h) &=& x'(t_k) &+& 2hx''(t_k) &+& \frac{2^2h^2}{2!}x'''(t_k) &+& \frac{2^3h^3}{3!}x^{(4)}(t_k) &+& \cdots\\
f(t_{k+3},x(t_{k+3})) = x'(t_k+3h) &=& x'(t_k) &+& 3hx''(t_k) &+& \frac{3^2h^2}{2!}x'''(t_k) &+& \frac{3^3h^3}{3!}x^{(4)}(t_k) &+& \cdots\\
&\vdots&&&&&&&&&\\
f(t_{k+m},x(t_{k+m})) = x'(t_k+mh) &=& x'(t_k) &+& mhx''(t_k) &+& \frac{m^2h^2}{2!}x'''(t_k) &+& \frac{m^3h^3}{3!}x^{(4)}(t_k) &+& \cdots.
\end{array}$$

Substituting these expansions into the expression for T_k (eventually) yields a convenient formula:

$$\begin{aligned}
\Big(\sum_{j=0}^m \beta_j\Big)T_k &= \frac{\sum_{j=0}^m\Big[\alpha_j\,x(t_{k+j}) - h\,\beta_j\,f(t_{k+j},x(t_{k+j}))\Big]}{h}\\
&= h^{-1}\Big[\sum_{j=0}^m \alpha_j\Big]x(t_k) + \sum_{\ell=0}^\infty h^\ell\Big[\sum_{j=0}^m\Big(\alpha_j\frac{1}{(\ell+1)!}j^{\ell+1} - \beta_j\frac{1}{\ell!}j^\ell\Big)x^{(\ell+1)}(t_k)\Big]\\
&= \frac{1}{h}\Big[\sum_{j=0}^m\alpha_j\Big]x(t_k) + \Big[\sum_{j=0}^m j\alpha_j - \sum_{j=0}^m\beta_j\Big]x'(t_k)\\
&\quad + h\Big[\sum_{j=0}^m\frac{j^2}{2}\alpha_j - \sum_{j=0}^m j\beta_j\Big]x''(t_k)\\
&\quad + h^2\Big[\sum_{j=0}^m\frac{j^3}{6}\alpha_j - \sum_{j=0}^m\frac{j^2}{2}\beta_j\Big]x'''(t_k)\\
&\quad + h^3\Big[\sum_{j=0}^m\frac{j^4}{24}\alpha_j - \sum_{j=0}^m\frac{j^3}{6}\beta_j\Big]x^{(4)}(t_k) + \cdots.
\end{aligned}$$

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. A linear multistep method is *consistent* if $T_k \to 0$ as $h \to 0$.

A condition for consistency is obvious from the formula for T_k:

Theorem. 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

$$\sum_{j=0}^{m} \alpha_j = 0 \qquad \text{and} \qquad \sum_{j=0}^{m} j\alpha_j = \sum_{j=0}^{m} \beta_j.$$

Definition. A linear multistep method is *order-p accurate* if $T_k = O(h^p)$ as $h \to 0$.

Theorem. 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 $\ell = 1, \ldots, p-1$.

We now compute the order of accuracy of some multistep methods discussed earlier.

Example (Euler's method). $\alpha_0 = -1$, $\alpha_1 = 1$, $\beta_0 = 1$, $\beta_1 = 0$.
Clearly $\alpha_0 + \alpha_1 = -1 + 1 = 0$ and $(0\alpha_0 + 1\alpha_1) - (\beta_0 + \beta_1) = 0$. When we analyzed this algorithm as a one-step method, we saw it had $T_k = O(h)$. We expect the same result from this multistep analysis. Indeed,

$$\Big(\tfrac{1}{2}0^2\alpha_0 + \tfrac{1}{2}1^2\alpha_1\Big) - \Big(0\beta_0 + 1\beta_1\Big) = \tfrac{1}{2} \neq 0.$$

Thus, $T_k = O(h)$.

Example (Trapezoid rule). $\alpha_0 = -1$, $\alpha_1 = 1$, $\beta_0 = \frac{1}{2}$, $\beta_1 = \frac{1}{2}$.
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,

$$\Big(\tfrac{1}{2}0^2\alpha_0 + \tfrac{1}{2}1^2\alpha_1\Big) - (0\beta_0 + 1\beta_1) = \frac{1}{2} - \frac{1}{2} = 0,$$

so $T_k = O(h^2)$, but

$$\Big(\tfrac{1}{6}0^3\alpha_0 + \tfrac{1}{6}1^3\alpha_1\Big) - \big(\tfrac{1}{2}0^2\beta_0 + \tfrac{1}{2}1^2\beta_1\big) = \frac{1}{6} - \frac{1}{4} \neq 0,$$

so the trapezoid rule is not third order accurate.

Example (2-step Adams–Bashforth). $\alpha_0 = 0$, $\alpha_1 = -1$, $\alpha_2 = 1$, $\beta_0 = -\frac{1}{2}$, $\beta_1 = \frac{3}{2}$, $\beta_2 = 0$.
We can now verify that this method presented earlier in this lecture lives up to its name. Consistency follows: $\alpha_0 + \alpha_1 + \alpha_2 = 0 - 1 + 1 = 0$; $(0\alpha_0 + 1\alpha_1 + 2\alpha_2) - (\beta_0 + \beta_1) = 1 - 1 = 0$. The second order condition is also satisfied,

$$\Big(\tfrac{1}{2}0^2\alpha_0 + \tfrac{1}{2}1^2\alpha_1 + \tfrac{1}{2}2^2\alpha_2\Big) - \Big(0\beta_0 + 1\beta_1\Big) = \tfrac{3}{2} - \tfrac{3}{2} = 0,$$

but not the third order,

$$\Big(\tfrac{1}{6}0^3\alpha_0 + \tfrac{1}{6}1^3\alpha_1 + \tfrac{1}{6}2^3\alpha_2\Big) - \Big(\tfrac{1}{2}0^2\beta_0 + \tfrac{1}{2}1^2\beta_1\Big) = \tfrac{7}{6} - \tfrac{3}{4} \neq 0.$$

Example (4-step Adams–Bashforth). $\alpha_0 = \alpha_1 = \alpha_2 = 0$, $\alpha_3 = -1$, $\alpha_4 = 1$; $\beta_0 = \frac{9}{24}$, $\beta_1 = \frac{37}{24}$, $\beta_2 = -\frac{59}{24}$, $\beta_3 = \frac{55}{24}$, $\beta_4 = 0$.
Consistency holds, since $\sum \alpha_j = -1 + 1 = 0$ and

$$\sum_{j=0}^{4} j\alpha_j - \sum_{j=0}^{4} \beta_j = \Big(3(-1) + 4(1)\Big) - \Big(\tfrac{9}{24} + \tfrac{37}{24} - \tfrac{47}{24} + \tfrac{55}{24}\Big) = 1 - 1 = 0.$$

The coefficients of h, h^2, and h^3 in the expansion for T_k all vanish:

$$\sum_{j=0}^{4} \tfrac{1}{2}j^2\alpha_j - \sum_{j=0}^{4} j\beta_j = \Big(\tfrac{3^2}{2}(-1) + \tfrac{4^2}{2}(1)\Big) - \Big(0(-\tfrac{9}{24}) + 1(\tfrac{37}{24}) + 2(-\tfrac{59}{24}) + 3(\tfrac{55}{24})\Big) = \tfrac{7}{2} - \tfrac{84}{24} = 0;$$

$$\sum_{j=0}^{4} \tfrac{1}{6}j^3\alpha_j - \sum_{j=0}^{4} \tfrac{1}{2}j^2\beta_j = \Big(\tfrac{3^3}{6}(-1) + \tfrac{4^3}{6}(1)\Big) - \Big(\tfrac{1^2}{2}(\tfrac{37}{24}) + \tfrac{2^2}{2}(-\tfrac{59}{24}) + \tfrac{3^2}{2}(\tfrac{55}{24})\Big) = \tfrac{37}{6} - \frac{148}{24} = 0;$$

$$\sum_{j=0}^{4} \tfrac{1}{24}j^4\alpha_j - \sum_{j=0}^{4} \tfrac{1}{6}j^3\beta_j = \Big(\tfrac{3^4}{24}(-1) + \tfrac{4^4}{24}(1)\Big) - \Big(\tfrac{1^3}{6}(\tfrac{37}{24}) + \tfrac{2^3}{6}(-\tfrac{59}{24}) + \tfrac{3^3}{6}(\tfrac{55}{24})\Big) = \tfrac{175}{24} - \tfrac{1050}{144} = 0.$$

However, the $O(h^4)$ term is not eliminated:

$$\sum_{j=0}^{4} \tfrac{1}{120}j^5\alpha_j - \sum_{j=0}^{4} \tfrac{1}{24}j^4\beta_j = \Big(\tfrac{3^5}{120}(-1) + \tfrac{4^5}{120}(1)\Big) - \Big(\tfrac{1^4}{24}(\tfrac{37}{24}) + \tfrac{2^4}{24}(-\tfrac{59}{24}) + \tfrac{3^4}{24}(\tfrac{55}{24})\Big) = \tfrac{1267}{120} - \tfrac{887}{144} \neq 0.$$

A similar computation establishes fourth-order accuracy for the 4-step Adams–Moulton formula

$$x_{k+3} = x_{k+2} + \tfrac{1}{24}h\Big(9f_{k+3} + 19f_{k+2} - 5f_{k+1} + f_k\Big).$$

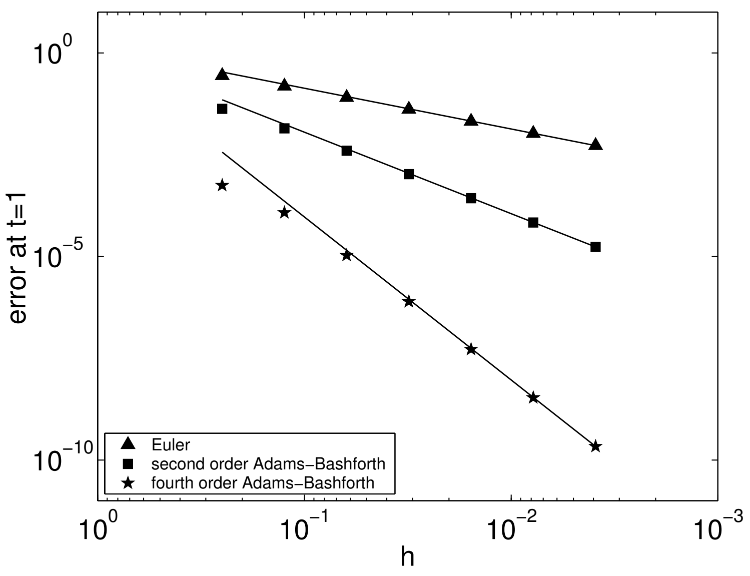
Computational verification. In the last lecture, we proved that *consistency implies convergence* for one-step methods. Essentially, provided the differential equation is sufficiently well-behaved (in the sense of Picard's Theorem), then the numerical solution produced by a consistent one-step method 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 $x_3 = x(t_3)$.

The plot below reports the results of this exercise, showing the absolute error at t_{final} as a function of h. (Note the log-log axis here.) Near the data points, we have drawn lines indicating pure h, h^2, and h^4 convergence. For this particular problem, the global error shrinks at the same rate as the truncation error.[§] Will this be true in more general settings? That is the subject of the next lecture.

[§]MATLAB programming note: The horizontal axis of this `loglog` plot would, by default, run from smallest h to largest h, (i.e., 10^{-3} to 10^0). But since we are interested in convergence as $h \to 0$, it is intuitively appealing for h to get smaller as we read from left to right. This is accomplished via: `set(gca,'XDir','reverse')`.

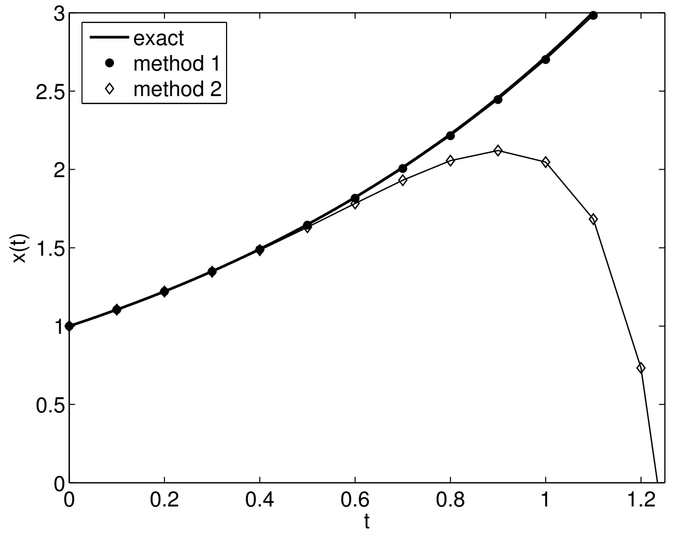


We close by offering evidence that there is more to the analysis of linear multistep methods than truncation error. Here are two explicit methods that are both second order:

$$x_{k+2} - \tfrac{3}{2}x_{k+1} + \tfrac{1}{2}x_k \;=\; h(\tfrac{5}{4}f_{k+1} - \tfrac{3}{4}f_k)$$

$$x_{k+2} - 3x_{k+1} + 2x_k \;=\; h(\tfrac{1}{2}f_{k+1} - \tfrac{3}{2}f_k).$$

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 31: Linear Multistep Methods: Zero Stability

Does consistency imply convergence for linear multistep methods? We saw that this was always the case for one-step methods, 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.

5.2.3 Zero stability for linear multistep methods.

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. 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 a way to check zero stability. To begin with, consider a particular example.

A novel second order method. The truncation error formulas from the previous 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

$$x_{k+2} = 2x_k - x_{k+1} + h(\tfrac{1}{2}f_k + \tfrac{5}{2}f_{k+1})$$

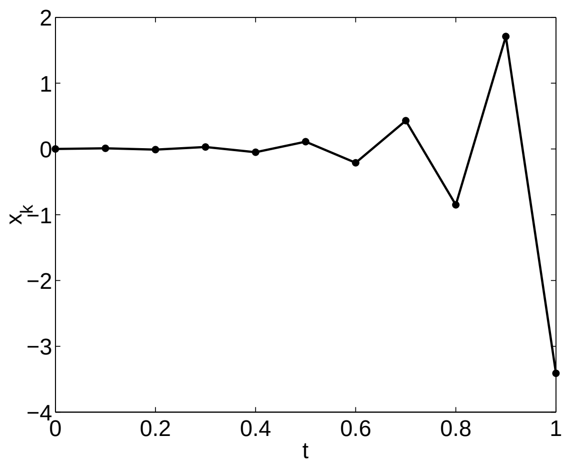
is second order accurate. Now we will test the zero-stability of this algorithm on the trivial model problem, $x'(t) = 0$ with $x(0) = 0$. Since $f(t,x) = 0$ in this case, the method reduces to

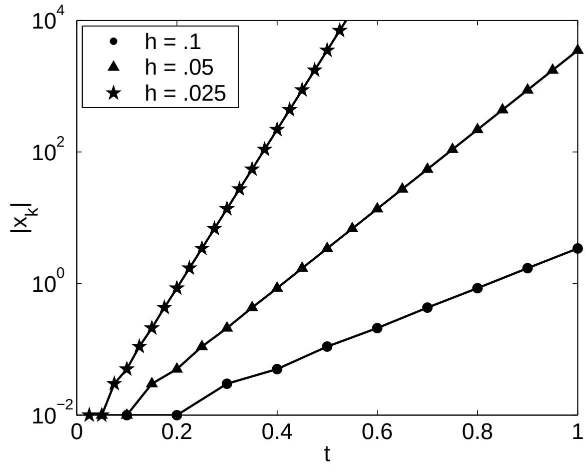
$$x_{k+2} = 2x_k - x_{k+1}.$$

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

$$\begin{aligned}
x_2 = 2x_0 - x_1 &= 2\cdot 0 - \varepsilon = -\varepsilon \\
x_3 = 2x_1 - x_2 &= 2(\varepsilon) - (-\varepsilon) = 3\varepsilon \\
x_4 = 2x_2 - x_3 &= 2(-\varepsilon) - 3\varepsilon = -5\varepsilon \\
x_5 = 2x_3 - x_4 &= 2(3\varepsilon) - (-5\varepsilon) = 11\varepsilon \\
x_6 = 2x_4 - x_5 &= 2(-5\varepsilon) - (11\varepsilon) = -21\varepsilon \\
x_7 = 2x_5 - x_6 &= 2(11\varepsilon) - (-21\varepsilon) = 43\varepsilon \\
x_8 = 2x_6 - x_7 &= 2(-21\varepsilon) - (43\varepsilon) = 85\varepsilon.
\end{aligned}$$

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. This is illustrated in the plot on the left below, which shows the evolution of x_k when $h = 0.1$ and $\varepsilon = 0.01$. There is 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 $t_{\rm final} = hk$ gets worse as h gets smaller and k grows accordingly! The figure on the right below illustrates this fact, showing $|x_k|$ over $t \in [0,1]$ for three different values of h. Clearly the smallest h leads to the most rapid error growth.





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 find all sequences $\{x_j\}_{j=0}^{\infty}$ that satisfy the 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 formula for x_k into the recurrence relation to see if we 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

$$A\gamma_1^k\Big(\gamma_1^2 + \gamma_1 - 2\Big) = B\gamma_2^k\Big(\gamma_2^2 + \gamma_2 - 2\Big) = 0.$$

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

$$x_k = \frac{\varepsilon}{3} - \frac{\varepsilon}{3}(-2)^k$$

generates the iterates $x_0 = 0$, $x_1 = \varepsilon$, $x_2 = -\varepsilon$, $x_3 = 3\varepsilon$, $x_4 = -5\varepsilon$, $\ldots$ computed previously. Notice the *exponential* growth with k: this 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.

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. It is 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 $\gamma^2 = 2 - \gamma$.

Repeat this process for the general linear multistep method

$$\sum_{j=0}^{m} \alpha_j x_{k+j} = h \sum_{j=0}^{m} \beta_j f(t_{k+j}, x_{k+j}).$$

For the differential equation $x'(t) = 0$, the method 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. 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 γ_1, γ_2, $\ldots$, γ_m, the general form of the solution of

$$\sum_{j=0}^{m} \alpha_j x_{k+j} = 0$$

[†]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 $c_{1,1}\gamma_1^k + c_{1,2}k\gamma_1^k + \cdots + c_{1,p}k^{p-1}\gamma_1^k$.

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 $x_0, \ldots, x_m$.

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., $|\gamma_j| \le 1$ 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). What is remarkable is that this criterion actually characterizes zero stability not just for $x'(t) = 0$, but for *all* well-behaved differential equations! This was discovered in the late 1950s by Germund Dahlquist.[§]

Theorem. A linear multistep method is zero-stable for any 'well-behaved' initial value problem provided

- all roots of $\rho(\gamma) = 0$ lie in the unit disk, i.e., $|\gamma| \le 1$;
- any roots on the unit circle ($|\gamma| = 1$) are simple (i.e., not multiple).

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 (Dahlquist Equivalence Theorem). Suppose an m-step linear multistep method applied to a 'well-behaved' initial value problem on $[t_0, t_{\rm final}]$ 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.,

$$x_{\lceil (t-t_0)/h \rceil} \to x(t) \quad \text{for all } t \in [t_0, t_{\rm final}].$$

as $h \to 0$ if and only if the method is *consistent* and *zero-stable.*

If the exact solution is sufficiently smooth, $x(t) \in C^{p+1}[t_0, t_{\rm final}]$ and the multistep method is order-p accurate ($T_k = O(h^p)$), then

$$x(t_k) - x_k = O(h^p)$$

for all $t_k \in [t_0, t_{\rm final}]$.

Dahlquist also characterized the maximal order of convergence for a zero-stable m-step multistep method.

[‡]Following on from the previous footnote, 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.

[§]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.

Theorem (First Dahlquist Stability Barrier). A zero-stable m-step linear multistep method has truncation error no better than

- $O(h^{m+1})$ if m is odd
- $O(h^m)$ if m is even.

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

$$x_{k+2} = x_k + 2hf_{k+1},$$

which has $O(h^2)$ 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. Let us use it to solve the equation $x'(t) = \lambda x$ 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

$$\gamma^2 - 2\lambda h\gamma - 1 = 0.$$

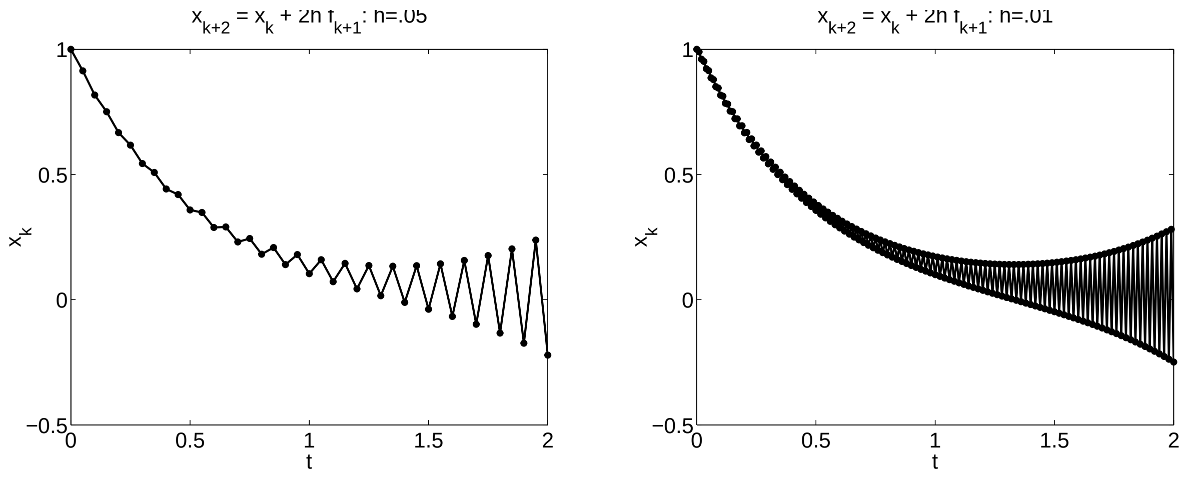
In fact, those roots are simply

$$\gamma = \lambda h \pm \sqrt{\lambda^2 h^2 + 1}.$$

Since $\sqrt{\lambda^2h^2+1} \geq 1$ 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 $x(t) = \mathrm{e}^{\lambda t}$, 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*.

The plots below show what this means. Set $\lambda = -2$ and $[t_0, t_{\text{final}}] = [0, 2]$. Start the method with $x_0 = 1$ and $x_1 = 1.01\ \mathrm{e}^{-2h}$. That is, the second data point has an initial error of 1%. The plot on the left 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 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.)

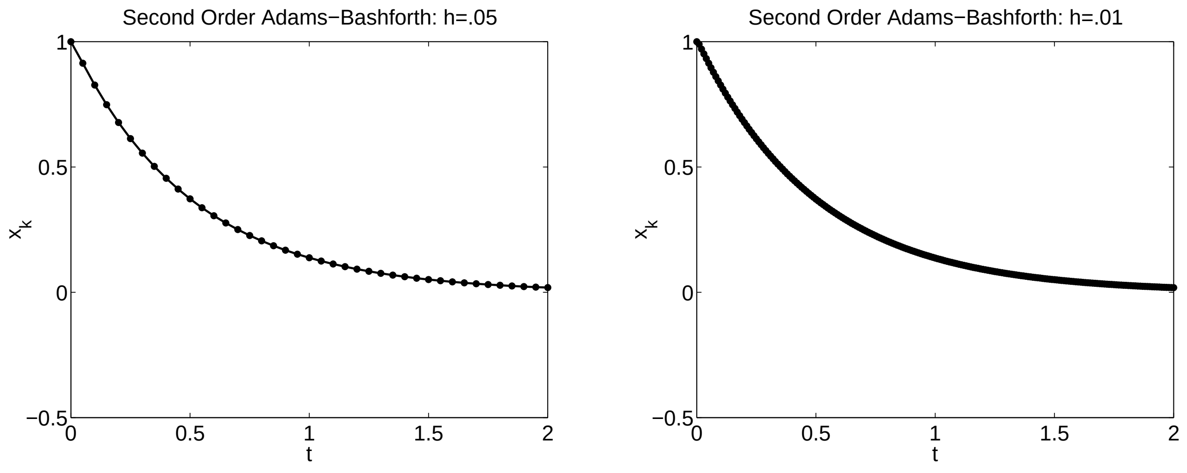


The solution does not blow up, but nor does it converge as $h \to 0$. So does this example contradict the Dahlquist Equivalence Theorem? No! The hypotheses for that theorem require consistent starting values. In this case, that means $x_1 \to x(t_0 + h)$ as $h \to 0$. (We assume that $x_0 = x(t_0)$ is 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 $\rho(z) = z^2 - z = z(z-1)$. When we repeat the exercise shown above with the same errors in x_1, we obtain the plots below. 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 32: 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. (You might have 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.)

5.2.4. Absolute Stability.

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

$$\begin{aligned} x_{k+1} &= x_k + hf_k \\ &= x_k + h\lambda x_k \\ &= (1 + h\lambda)x_k. \end{aligned}$$

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:

$$\begin{aligned} x_{k+1} &= x_k + hf_{k+1} \\ &= x_k + h\lambda x_{k+1}. \end{aligned}$$

Solve this equation for x_{k+1} to obtain

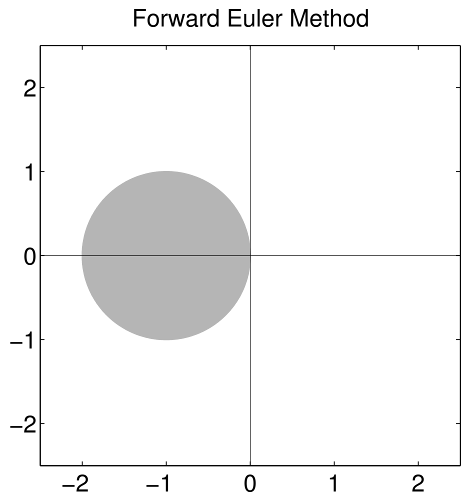
$$x_{k+1} = \frac{1}{1 - h\lambda}x_k,$$

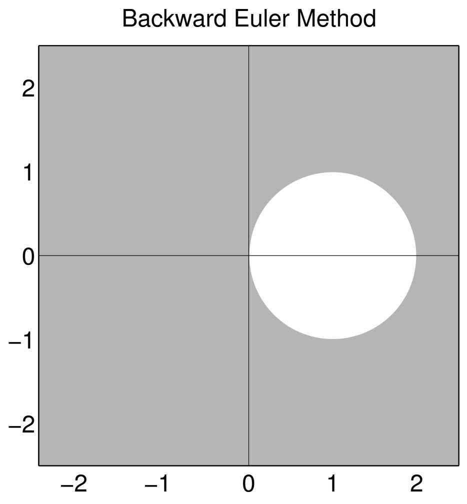
from which it follows that

$$x_k = (1 - h\lambda)^{-k}x_0.$$

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 $h\lambda$. The stability regions for the explicit and backward Euler methods are shown below. The gray region shows values of λh in the complex plane for which the method is absolutely stable. (For the implicit method, this regions extend beyond the range of the plot.)





$x_{k+1} = x_k + hf_k$ $\qquad\qquad$ $x_{k+1} = x_k + hf_{k+1}$

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

$$\sum_{j=0}^{m} (\alpha_j - h\lambda\beta_j) x_{k+j}.$$

Note that this closely resembles the equation 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 $h\lambda$.

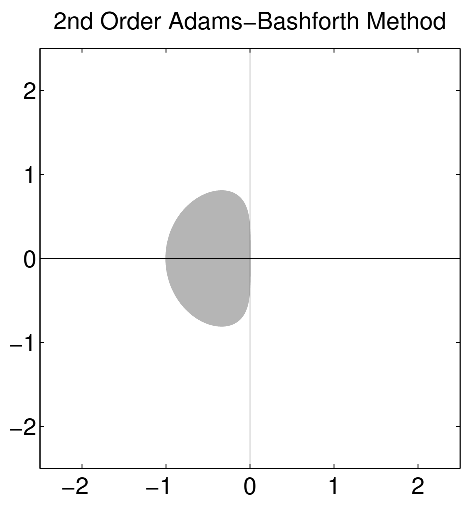
We seek linear multistep methods that share the following properties:

- high order truncation error;
- zero stability;
- absolute stability region that contains as much of the left half of the complex plane as possible.

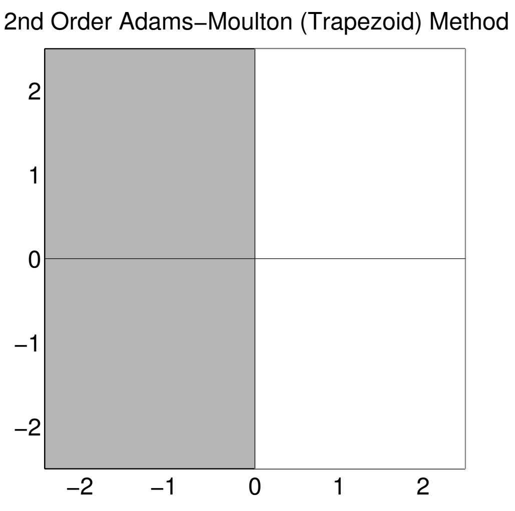
Those methods for which the stability region contains the entire left half plane are distinguished, as they will produce, *for any value of* h, exponentially decaying numerical solutions to linear problems that have exponentially decaying true solutions, i.e., when $\operatorname{Re}\lambda < 0$.

Definition. A linear multistep method is *A-stable* provided that its stability region contains the entire left half of the complex plane.

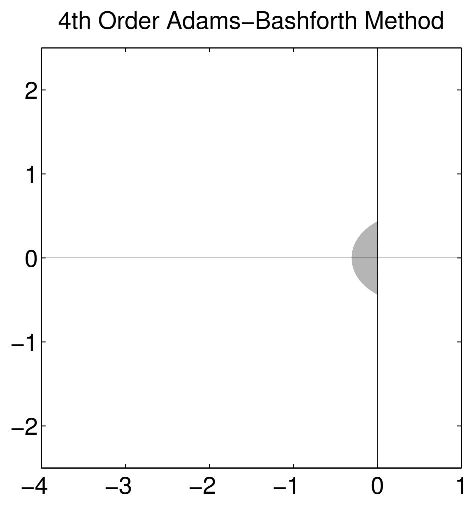
Next we show the stability regions for several different methods.



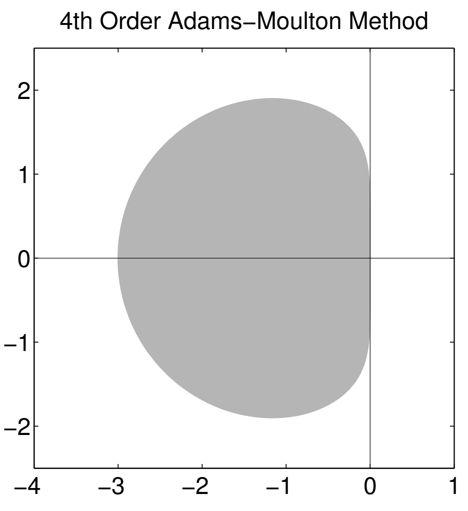
$$x_{k+2} - x_{k+1} = h(\tfrac{3}{2}f_{k+1} - \tfrac{1}{2}f_k)$$



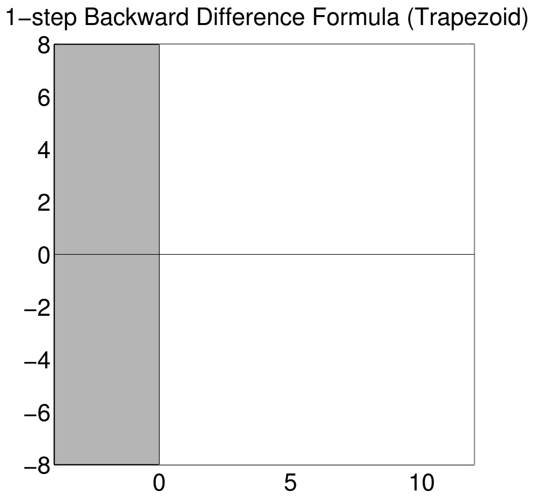
$$x_{k+1} - x_k = \tfrac{1}{2}h(f_k + f_{k+1})$$



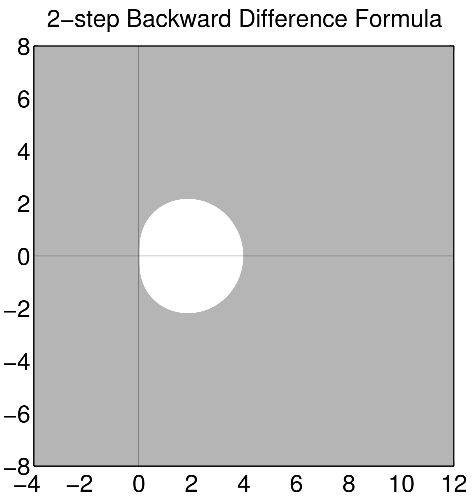
$$24x_{k+4} - 24x_{k+3} = h\big(55f_{k+3} - 59f_{k+2} + 37f_{k+1} - 9f_k\big)$$



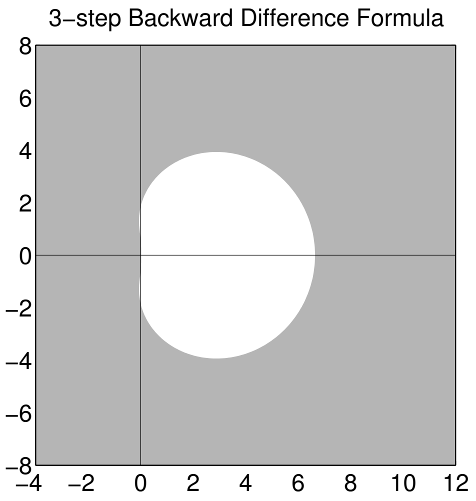
$$24x_{k+3} - 24x_{k+2} = h\big(9f_{k+3} + 19f_{k+2} - 5f_{k+1} + f_k\big)$$



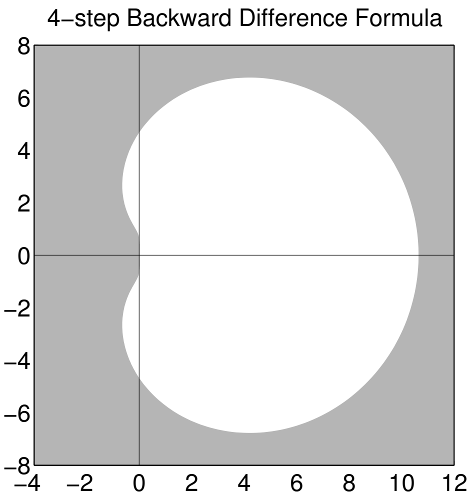
$x_{k+1} - x_k = \frac{1}{2}h(f_k + f_{k+1})$



$3x_{k+2} - 4x_{k+1} + x_k = 2hf_{k+2}$



$11x_{k+3} - 18x_{k+2} + 9x_{k+1} - 2x_k = 6hf_{k+3}$



$25x_{k+4} - 48x_{k+3} + 36x_{k+2} - 16x_{k+1} + 3x_k = 12hf_{k+4}$

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

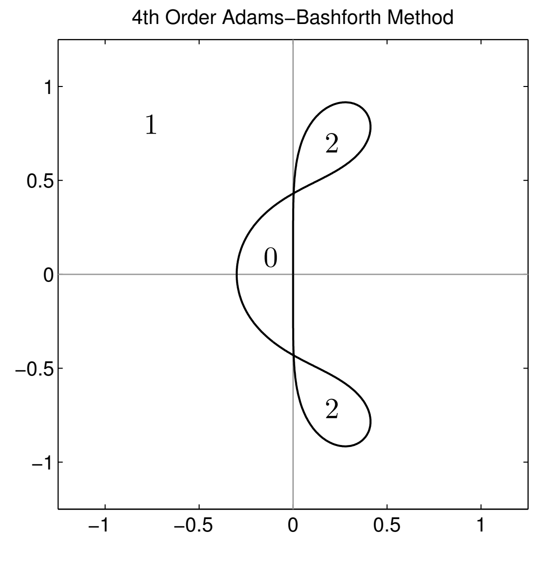
$$\lambda h = \frac{z^2 - z}{\frac{3}{2}z - 1}.$$

For general methods, this expression takes the form

$$\lambda h = \frac{\sum_{j=0}^{m} \alpha_j z^j}{\sum_{j=0}^{m} \beta_j z^j},$$

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 $z = \mathrm{e}^{\mathrm{i}\theta}$, $\theta \in [0, 2\pi)$. 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 the plot below; it divides the complex plane into regions where the stability 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, as shown in the plot several pages earlier.



Lecture 33: Stiff Differential Equations

5.2.5. Stiff 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 $x(t)$ is replaced by a vector $\mathbf{x}(t)$. In these notes, we shall focus upon *linear systems* of ODEs. (In applications one often encounters nonlinear ODEs, but behavior of such a system near a steady state can often be understood by examining the *linearization* of the equation about that 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. Assume 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 $\mathbf{\Lambda} = \mathrm{diag}(\lambda_1, \ldots, \lambda_n)$. Premultiplying the differential equation by $\mathbf{V}^{-1}$ yields

$$\mathbf{V}^{-1}\mathbf{x}'(t) = \mathbf{\Lambda}\mathbf{V}^{-1}\mathbf{x}(t), \quad \mathbf{V}^{-1}\mathbf{x}(0) = \mathbf{V}^{-1}\mathbf{x}_0.$$

Now let $\mathbf{y}(t) = \mathbf{V}^{-1}\mathbf{x}(t)$, which can be thought of as the vector $\mathbf{x}(t)$ represented in a transformed coordinate system. In these new coordinates, the matrix equation decouples into a system of n linear independent scalar equations, as the above equation takes the form

$$\mathbf{y}'(t) = \mathbf{\Lambda}\mathbf{y}(t), \quad \mathbf{y}(t) = \mathbf{y}(0).$$

This is equivalent to

$$\begin{aligned} y_1'(t) &= \lambda_1 y_1(t), \quad y_1(0) = [\mathbf{V}^{-1}\mathbf{x}_0]_1; \\ &\vdots \\ y_n'(t) &= \lambda_n y_n(t), \quad y_n(0) = [\mathbf{V}^{-1}\mathbf{x}_0]_n, \end{aligned}$$

and each of these equations has the simple solution

$$y_j(t) = \mathrm{e}^{\lambda_j t} y_j(0).$$

Now we can use the relationship $\mathbf{x}(t) = \mathbf{V}\mathbf{y}(t)$ to transform back to the original coordinates. Define

$$\mathrm{e}^{\mathbf{\Lambda}t} := \begin{bmatrix} \mathrm{e}^{t\lambda_1} & & \\ & \ddots & \\ & & \mathrm{e}^{t\lambda_n} \end{bmatrix}.$$

Then we can write

$$\mathbf{x}(t) = \mathbf{V}\mathbf{y}(t) = \mathbf{V}\mathrm{e}^{\mathbf{\Lambda}t}\mathbf{y}(0) = \mathbf{V}\mathrm{e}^{\mathbf{\Lambda}t}\mathbf{V}^{-1}\mathbf{x}_0, \tag{33.1}$$

which motivates the definition of the *matrix exponential*,

$$\mathrm{e}^{t\mathbf{A}} := \mathbf{V}\mathrm{e}^{\mathbf{\Lambda}t}\mathbf{V}^{-1},$$

in which case the solution $\mathbf{x}(t)$ has the convenient form

$$\mathbf{x}(t) = \mathrm{e}^{\mathbf{\Lambda}t}\mathbf{x}_0.$$

What can be said of the magnitude of the solution $\mathbf{x}(t)$? 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

$$\frac{\|\mathbf{x}(t)\|_2}{\|\mathbf{x}_0\|_2} \ \le\ \|\mathbf{V}\|_2\|\mathbf{V}^{-1}\|_2 \max_{1\le j\le n} |\mathrm{e}^{t\lambda_j}|. \tag{33.2}$$

Thus the asymptotic decay rate of $\|\mathbf{x}(t)\|_2$ is controlled by the rightmost 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 $t \to \infty$. Note that 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 $t \to 0$.[†]

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:

$$\mathrm{e}^{t\mathbf{A}} = \mathbf{I} + t\mathbf{A} + \frac{1}{2!}t^2\mathbf{A}^2 + \frac{1}{3!}t^3\mathbf{A}^3 + \frac{1}{4!}t^4\mathbf{A}^4 + \cdots.$$

If we set $\mathbf{x}(t) = \mathrm{e}^{t\mathbf{A}}\mathbf{x}_0$, then we have

$$\begin{aligned}
\mathbf{x}'(t) &= \frac{\mathrm{d}}{\mathrm{d}t}\Big(\mathrm{e}^{t\mathbf{A}}\mathbf{x}_0\Big)\\
&= \frac{\mathrm{d}}{\mathrm{d}t}\Big(\mathbf{I} + t\mathbf{A} + \frac{t^2}{2!}\mathbf{A}^2 + \frac{t^3}{3!}\mathbf{A}^3 + \cdots\Big)\mathbf{x}_0\\
&= \Big(\mathbf{A} + t\mathbf{A}^2 + \frac{t^2}{2!}\mathbf{A}^3 + \frac{t^3}{3!}\mathbf{A}^4 + \cdots\Big)\mathbf{x}_0\\
&= \mathbf{A}\Big(\mathbf{I} + t\mathbf{A} + \frac{t^2}{2!}\mathbf{A}^2 + \frac{t^3}{3!}\mathbf{A}^3 + \cdots\Big)\mathbf{x}_0\\
&= \mathbf{A}\mathrm{e}^{t\mathbf{A}}\mathbf{x}_0\\
&= \mathbf{A}\mathbf{x}(t).
\end{aligned}$$

Hence $\mathbf{x}(t) = \mathrm{e}^{t\mathbf{A}}\mathbf{x}_0$ solves the equation $\mathbf{x}'(t) = \mathbf{A}\mathbf{x}(t)$, and satisfies the initial condition $\mathbf{x}(0) = \mathbf{x}_0$.

What can be said of the behavior of a linear multistep method applied to this equation? Euler's method, for example, takes the form

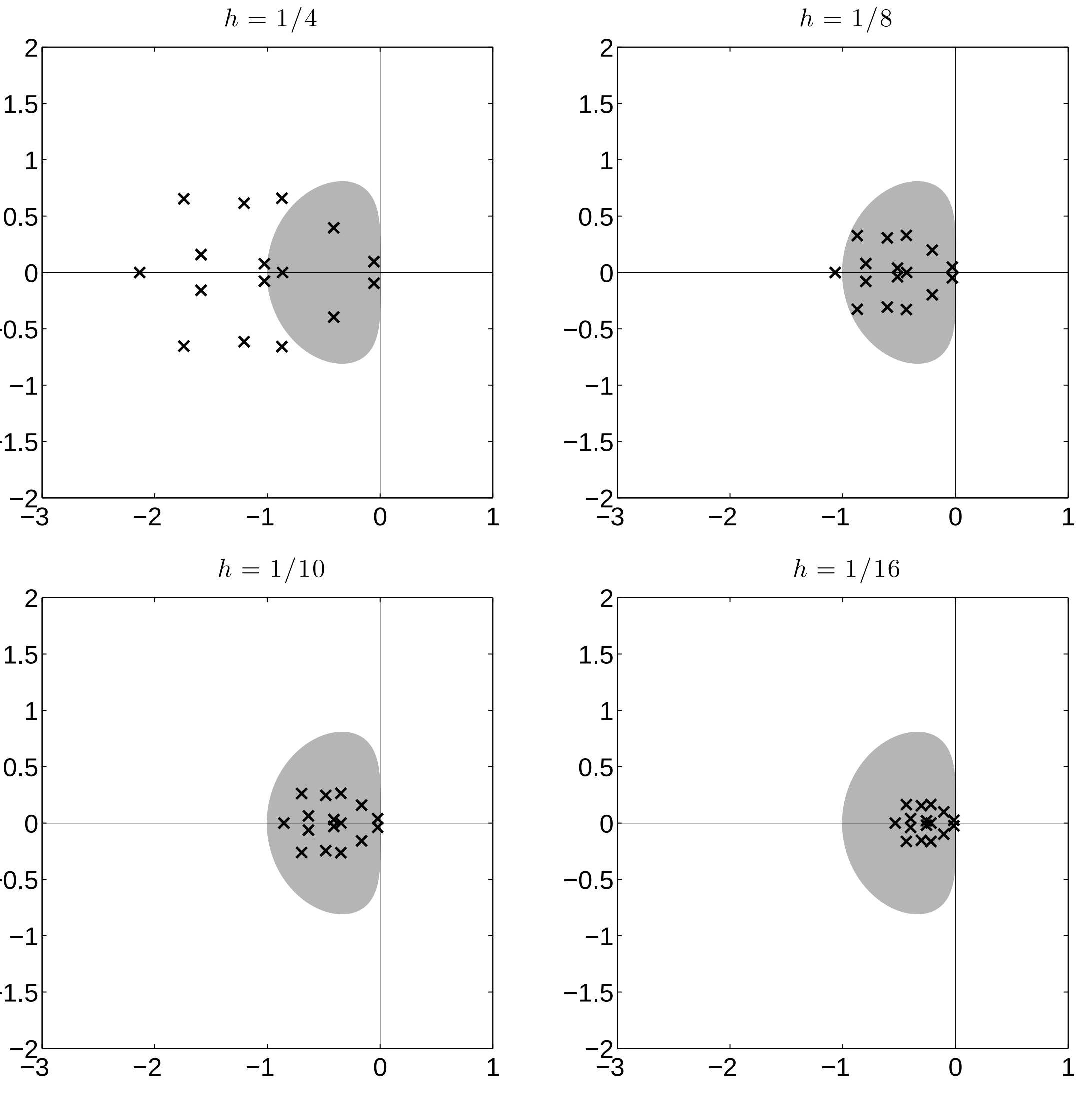
$$\begin{aligned}
\mathbf{x}_{k+1} &= \mathbf{x}_k + h\mathbf{A}\mathbf{x}_k\\
&= (\mathbf{I} + h\mathbf{A})\mathbf{x}_k,
\end{aligned}$$

and hence $\mathbf{x}_k = (\mathbf{I} + h\mathbf{A})^k\mathbf{x}_0$.

[†]The possibility of this *transient growth* complicates the analysis of dynamical systems with non-Hermitian 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.

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 modulus. 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 $(\mathbf{I}+h\mathbf{A})\mathbf{v}_j = \mathbf{v}_+ h\mathbf{A}\mathbf{v}_j = (1+h\lambda_j)\mathbf{v}_j$. Hence, 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, we need $h\lambda_j$ to fall 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 is illustrated in the following example. 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 below 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 gray), then the iteration will *grow exponentially*! If h is sufficiently small that $h\lambda_j$ is in the stability region for all eigenvalues λ_j, then $\mathbf{x}_k \to \mathbf{0}$ as $k \to \infty$, consistent with the fact that $\mathbf{x}(t) \to \mathbf{0}$ as $t \to \infty$.



It is worth looking at this example a little bit closer. 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 (33.1) for the true solution $\mathbf{x}(t)$ in terms of the matrix exponential. As we did in that case, we can bound $\mathbf{x}_k$ as follows:

$$\begin{aligned}\|\mathbf{x}_k\|_2 &= \|\mathbf{V}(\mathbf{I} + h\Lambda)^k\mathbf{V}^{-1}\mathbf{x}_0\|_2\\ &= \|\mathbf{V}(\mathbf{I} + h\Lambda)^k\mathbf{V}^{-1}\|_2\|\mathbf{x}_0\|_2\\ &= \|\mathbf{V}\|_2\|\mathbf{V}^{-1}\|_2\|(\mathbf{I} + h\Lambda)^k\|_2\|\mathbf{x}_0\|_2.\end{aligned}$$

Since $\mathbf{I} + h\Lambda$ is a diagonal matrix, we have

$$(\mathbf{I} + h\Lambda)^k = \begin{bmatrix}(1+h\lambda_1)^k & & & \\ & (1+h\lambda_2)^k & & \\ & & \ddots & \\ & & & (1+h\lambda_n)^k\end{bmatrix},$$

giving

$$\|(\mathbf{I} + h\Lambda)^k\|_2 = \max_{1\le j\le n} |1 + h\lambda_j|^k.$$

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 (33.2) 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

$$\mathbf{x}_k = \mathbf{V}(\mathbf{I} + h\Lambda)^k\mathbf{V}^{-1}\mathbf{x}_0 = \mathbf{V}(\mathbf{I} + h\Lambda)^k\mathbf{V}^{-1}\mathbf{V}\mathbf{c} = \mathbf{V}\Big(\mathbf{I} + h\Lambda)^k\mathbf{c}\Big).$$

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

$$\mathbf{x}_k = [\,\mathbf{v}_1 \quad \mathbf{v}_2 \quad \cdots \mathbf{v}_n\,] \begin{bmatrix} (1+h\lambda_1)^k c_1 \\ (1+h\lambda_2)^k c_2 \\ \vdots \\ (1+h\lambda_n)^k c_n \end{bmatrix} = \sum_{j=1}^{n} c_j (1+h\lambda_j)^k \mathbf{v}_j.$$

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 $|1+h\lambda_j|$:

$$|1+h\lambda_\ell| = \max_{1\le j\le n} |1+h\lambda_j|.$$

In our last example plotted above, the step size did not need to be very small in order 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. Consider the following simple example. Let

$$\mathbf{A} = \begin{bmatrix} -1999 & -1998 \\ 999 & 998 \end{bmatrix},$$

which has the diagonalization

$$\mathbf{A} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^{-1} = \begin{bmatrix} 2 & -1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} -100 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix}.$$

The eigenvalues are $\lambda_1 = -100$ and $\lambda_2 = -1$, and the exact solution takes the form

$$\mathbf{x}(t) = \mathrm{e}^{t\mathbf{A}}\mathbf{x}_0 = \mathbf{V} \begin{bmatrix} \mathrm{e}^{-100t} & 0 \\ 0 & \mathrm{e}^{-t} \end{bmatrix} \mathbf{V}^{-1}\mathbf{x}_0.$$

If the initial condition has the form

$$\mathbf{x}_0 = \mathbf{V}\begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = c_1 \begin{bmatrix} 2 \\ -1 \end{bmatrix} + c_2 \begin{bmatrix} -1 \\ 1 \end{bmatrix},$$

then the solution can be written as

$$\mathbf{x}(t) = \mathbf{V} \begin{bmatrix} \mathrm{e}^{-100t} & 0 \\ 0 & \mathrm{e}^{-t} \end{bmatrix} \mathbf{V}^{-1}\mathbf{x}_0 = \mathbf{V} \begin{bmatrix} \mathrm{e}^{-100t} & 0 \\ 0 & \mathrm{e}^{-t} \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = c_1\mathrm{e}^{-100t} \begin{bmatrix} 2 \\ -1 \end{bmatrix} + c_2\mathrm{e}^{-t} \begin{bmatrix} -1 \\ 1 \end{bmatrix},$$

and so we see that $\mathbf{x}(t) \to \mathbf{0}$ as $t \to \infty$. The eigenvalue $\lambda_1 = -100$ corresponds to a *fast transient*, a component of the solution that decays very rapidly; the eigenvalue $\lambda_2 = -1$ corresponds to a *slow transient*, a component of the solution that decays much more slowly.

Suppose we wish to obtain a solution with the forward Euler method. To obtain a numerical solution $\{\mathbf{x}_k\}$ that mimics the asymptotic behavior of the true solution, $\mathbf{x}(t) \to \mathbf{0}$, we much choose h sufficiently small that $|1+h\lambda_1| = |1-100h| < 1$ and $|1+h\lambda_2| = |1-h| < 1$. The first condition requires $h \in (0, 1/50]$, which the second condition is far less restrictive: $h \in (0,2)$. 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).*

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. (Such examples are seen in the method of lines problems on Problem Set 6.) 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).

Lecture 34: Second Order Equations: Special Topics

5.3. Second Order Equations.

Built upon Newton's Second Law of Motion ($F = ma$), many models in science and engineering lead to second order differential equations of the generic form $x''(t) = f(t, x(t), x'(t))$. We have seen that such equations can be converted to systems of first order equations and solved using the techniques we have been studying in the preceding lectures (see below for details). However, second-order problems often posses special structure that can be exploited. In this lecture, we briefly introduce two such concepts: geometric integration and boundary value problems.

5.3.1. Geometric Integration.

Over the past two decades there has been a growing appreciation of the fact that some numerical methods capture special features of the exact solution, and thus may be preferred over fancier methods of higher order.[†] For example, gravitational systems (in the absence of dissipative forces such as atmospheric drag) conserve energy. If one applies a standard one-step or multistep method to such a problem, the numerical solution will not exhibit such energy conservation, and thus the solutions are, to some degree, not physically sensible. Such effects are minimized as the step size h is reduced, but they will still persist. If given the choice, one might like to use a numerical method that produces approximate solutions that also preserve energy.

Remarkably, such methods exist for a variety of conserved quantities (typically deriving from Hamiltonian systems). Often these methods are implicit, but it is worth mentioning one important special explicit case. For problems of the form

$$x''(t) = f(x(t)),$$

there exists a simple explicit scheme called the *Störmer–Verlet* algorithm. Let $v(t) = x'(t)$ and suppose $v_k \approx v(t_k)$ and $x_k \approx x(t_k)$. Then the algorithm takes the form

$$\begin{aligned} v_{k+1/2} &= v_k + \tfrac{1}{2}hf(x_k) \\ x_{k+1} &= x_k + hv_{k+1/2} \\ v_{k+1} &= v_{k+1/2} + \tfrac{1}{2}hf(x_{k+1}). \end{aligned}$$

Note that operations can be arranged so that only one evaluation of f is required per step. Thus, for essentially the same computational expense of the forward Euler method, one obtains an energy-preserving integrator.

5.3.2. Boundary Value Problems.

So far all the differential equations we have studied have been *initial value problems*. We are always given $x(t_0) = x_0$ (or $x(t_0) = x_0$ and $x'(t_0) = v_0$ for a second-order system), i.e., the complete state of the system at time t_0. If faced with a second order equation,

$$x''(t) = f(t, x(t), x'(t)),$$

[†]See the excellent text *Geometric Numerical Integration: Structure-Preserving Algorithms for Ordinary Differential Equations* by E. Hairer, C. Lubich, and G. Wanner, Springer, Berlin, 2002.

we can covert to a system of two first order equations. Let $v(t) = x'(t)$. Then

$$\begin{bmatrix} x(t) \\ v(t) \end{bmatrix}' = \begin{bmatrix} v(t) \\ f(t, x(t), v(t)) \end{bmatrix}.$$

An initial value problem will supply values of both $x(t_0)$ and $v(t_0)$.

In some situations we have may have, for example, $x(t_0)$ and $x(t_{\rm final})$, rather than $x(t_0)$ and $v(t_0)$. This is an example of a *two-point boundary value problem.* Such problems require an entirely different approach. For one thing, solutions need not even exist for some pairs of $x(t_0)$ and $x(t_{\rm final})$. When the solution does exist, here are several approaches for finding it that readily generalize to higher order problems.

Shooting Method.

Suppose we are given

$$x''(t) = f(t, x(t), x'(t)),$$

with $x(t_0) = x_0$ and $x(t_{\rm final}) = \omega$.

The shooting method begins with a *guess* of a condition for $x'(t_0) = v(t_0)$, say

$$x'(t_0) = \widehat{v},$$

resulting in an initial value problem

$$\widehat{x}''(t) = f(t, \widehat{x}(t), \widehat{x}'(t)), \qquad \widehat{x}(t_0) = x_0, \quad \widehat{x}'(t_0) = \widehat{v}$$

that can be integrated using any of the techniques we previously studied. Since the guess for $\widehat{v}$ was probably not the slope of the true solution, $x'(t_0)$, the solution of the initial value problem will not in general satisfy the condition at the right boundary, i.e.,

$$\widehat{x}(t_{\rm final}) \neq x(t_{\rm final}).$$

The shooting method thus adjusts the value of $\widehat{v}$ and tries again. This procedure resembles the action of adjusting angle of a cannon barrel (hence the initial slope of a shell shot out of that barrel) to zero-in on some distant target, hence the name. Techniques for solving nonlinear equations, such as Newton's method or the secant method, can be used to find the value of $\widehat{v}$ that is a zero of the function

$$g(\widehat{v}) = \widehat{x}(t_{\rm final}; \widehat{v}) - \omega.$$

Finite Differences.

We next present an alternative to the shooting method for the linear boundary value problem

$$x''(t) + p(t)x'(t) + q(t)x(t) = f(t), \qquad \text{given values for } x(t_0) \text{ and } x(t_{\rm final}),$$

that leads ultimately to a linear algebra problem. Construct a grid $t_0, t_1, \ldots, t_n = t_{\rm final}$, where $t_j = t_0 + hj$ for $h = (t_{\rm final} - t_0)/n$. Then expanding in Taylor series, we see that for $1 \leq j \leq n-1$

$$x''(t_j) \approx \frac{x(t_{j+1}) - 2x(t_j) + x(t_{j-1})}{h^2} + O(h^2)$$

and

$$x'(t_j) \approx \frac{x(t_{j+1}) - x(t_{j-1})}{2h} + O(h^2).$$

We do not know $x(t_{j+1})$, $x(t_j)$, and $x(t_{j-1})$ exactly, so we will replace them by the approximations

$$x''(t_j) \approx \frac{x_{j+1} - 2x_j + x_{j-1}}{h^2}$$

and

$$x'(t_j) \approx \frac{x_{j+1} - x_{j-1}}{2h},$$

where $x_0 = x(t_0)$ and $x_n = x(t_{\rm final})$ are exact. The goal now is to find $x_1, \ldots, x_{n-1}$. We obtain a linear system of the form

$$\left[\begin{array}{ccccc} \alpha_1 & \gamma_1 & & & \\ \beta_2 & \alpha_2 & \gamma_2 & & \\ & \ddots & \ddots & \ddots & \\ & & \beta_{n-2} & \alpha_{n-2} & \gamma_{n-2} \\ & & & \beta_{n-1} & \alpha_{n-1} \end{array}\right] \left[\begin{array}{c} x_1 \\ x_2 \\ \vdots \\ x_{n-2} \\ x_{n-1} \end{array}\right] = \left[\begin{array}{c} f(t_1) - (1/h^2 - p(t_0)/2h)x_0 \\ f(t_2) \\ \vdots \\ f(t_{n-2}) \\ f(t_{n-1}) - (1/h^2 + p(t_n)/2h)x_n \end{array}\right],$$

where

$$\alpha_j = -\frac{2}{h^2} + q(t_j) \qquad \beta_j = \frac{1}{h^2} - \frac{p(t_j)}{2h} \qquad \gamma_j = \frac{1}{h^2} + \frac{p(t_j)}{2h}$$

for $j = 1, \ldots n-1$. To solve the differential equation, we simply have to solve the linear algebraic system $\mathbf{Ax} = \mathbf{f}$, which we can do using the QR factorization discussed at the beginning of the semester, or the LU factorization we shall begin discussing in the next lecture. Nonlinear boundary value problems result in a nonlinear system of equations, which you can tackle using techniques that will be taught in CAAM 454/554 in the Spring.

Lecture 35: Gaussian Elimination

6. Linear Systems Revisited, and Eigenvalue Problems.

Our study of differential equations reminds us that many numerical analysis problems eventually boil down to questions of linear algebra. For example, if we solve the linear system $\mathbf{x}'(t) = \mathbf{A}\mathbf{x}(t)$ using the forward Euler method, we should select our time-step based on the eigenvalues of the matrix $\mathbf{A}$. If we instead use the backward Euler method, we arrive at the iteration

$$\mathbf{x}_{k+1} = (\mathbf{I} - h\mathbf{A})^{-1}\mathbf{x}_k,$$

which requires us to solve systems of linear algebraic equations of the form $(\mathbf{I} - h\mathbf{A})\mathbf{x}_{k+1} = \mathbf{x}_k$ for the unknown $\mathbf{x}_{k+1}$. (Since the matrix $\mathbf{I} - h\mathbf{A}$ is independent of k, we will certainly benefit from factoring $\mathbf{I} - h\mathbf{A}$ *once* at the beginning, and repeatedly applying that factorization at each iteration.) In the last lecture, we saw how systems of linear equations arise when solving boundary value problems by finite difference discretization. Indeed, linear algebra is ubiquitous in numerical analysis.

In the next few lectures, we shall see how to solve linear systems of algebraic equations two to four times faster than we could with the QR factorization, and we shall also learn the basics of an algorithm for computing eigenvalues.

6.1. Solving Linear Systems with Gaussian Elimination.

When you first encountered the linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$ in a linear algebra class, you were almost certainly taught to solve it with the *Gaussian elimination* algorithm. One typically must solve many 3×3 systems by hand as homework exercises, which one accomplishes by performing elementary row operations on the 'augmented matrix' $[\mathbf{A}\ \mathbf{b}]$, until the matrix $\mathbf{A}$ has been transformed into upper triangular form; the entries of $\mathbf{x}$ can then be found by back substitution. (When $\mathbf{A}$ is reduced to a diagonal matrix, the process is called *Gauss–Jordan* elimination.)

Compare that process to our methodology for computing a QR factorization of a square matrix $\mathbf{A}$: Premultiply $\mathbf{A}$ with a series of Householder transformations $\mathbf{Q}_1, \ldots, \mathbf{Q}_{n-1}$ to reduce $\mathbf{A}$ to upper triangular form:

$$\mathbf{Q}_{n-1} \cdots \mathbf{Q}_2\mathbf{Q}_1\mathbf{A} = \mathbf{R}.$$

Now move the $\mathbf{Q}_j$ matrices to the other side:

$$\mathbf{A} = \mathbf{Q}_1^{-1}\mathbf{Q}_2^{-1} \cdots \mathbf{Q}_{n-1}^{-1}\mathbf{R}.$$

Since the $\mathbf{Q}_j$ matrices are Hermitian and unitary, we have $\mathbf{Q}_k^{-1} = \mathbf{Q}_k$, and hence

$$\begin{aligned}\mathbf{A} &= (\mathbf{Q}_1\mathbf{Q}_2 \cdots \mathbf{Q}_{n-1})\mathbf{R} \\ &= \mathbf{Q}\mathbf{R}.\end{aligned}$$

This QR factorization might seen much more sophisticated than Gaussian elimination, but it turns out that we can formulate the familiar Gaussian elimination procedure in a very similar manner. One can learn much by taking a matrix-level view of the process, rather than getting lost in the tedious entry-by-entry mechanics that bog down many linear algebra students.

The idea is to premultiply $\mathbf{A}$ by a series of *Gauss transformations*, which are lower triangular matrices with ones on the main diagonal and nonzeros below the diagonal in only one column: they encode elementary row operations. Each Gauss transformation $\mathbf{L}_k$ will introduce zeros below the diagonal in the kth column, eventually giving

$$\mathbf{L}_{n-1}\cdots\mathbf{L}_2\mathbf{L}_1\mathbf{A} = \mathbf{U},$$

where $\mathbf{U}$ is upper triangular (like $\mathbf{R}$ in the QR factorization, though $\mathbf{U} \neq \mathbf{R}$ in general). Now invert the $\mathbf{L}_k$ matrices to move them to the other side:

$$\mathbf{A} = (\mathbf{L}_1^{-1}\mathbf{L}_2^{-1}\cdots\mathbf{L}_{n-1}^{-1})\mathbf{U}.$$

Since $\mathbf{L}_k$ is unit lower triangular, its inverse shares these properties. Moreover, the product of unit lower triangular matrices is also unit lower triangular, so we can form

$$\mathbf{L} = \mathbf{L}_1^{-1}\mathbf{L}_2^{-1}\cdots\mathbf{L}_{n-1}^{-1},$$

which is unit lower triangular, and gives

$$\mathbf{A} = \mathbf{L}\mathbf{U}.$$

Small Example. Consider the matrix

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 4 \\ 2 & 1 & -1 \\ -1 & 4 & 2 \end{bmatrix}.$$

We wish to multiply $\mathbf{A}$ by a unit lower triangular matrix $\mathbf{L}_1$ so that $\mathbf{L}_1\mathbf{A}$ has zeros in the first column below the main diagonal. To do so, we will modify rows 2 and 3 with multiples of the first row:

$$\text{ROW } 2 \leftarrow \text{ROW } 2 - 2 \times \text{ROW } 1;$$

$$\text{ROW } 3 \leftarrow \text{ROW } 3 + 1 \times \text{ROW } 1.$$

We encode these operations in the following matrix-matrix product:

$$\begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 2 & 4 \\ 2 & 1 & -1 \\ -1 & 4 & 2 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 4 \\ 0 & -3 & -9 \\ 0 & 6 & 6 \end{bmatrix}.$$

We call the unit lower triangular matrix $\mathbf{L}_1$:

$$\mathbf{L}_1 = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}.$$

At the next stage, we seek to zero out the subdiagonal entry from the second column of $\mathbf{L}_1\mathbf{A}$:

$$\text{ROW } 3 \leftarrow \text{ROW } 3 + 2 \times \text{ROW } 2.$$

This operation is equivalent to the matrix-matrix product

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 2 & 1 \end{bmatrix}\begin{bmatrix} 1 & 2 & 4 \\ 0 & -3 & -9 \\ 0 & 6 & 6 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 4 \\ 0 & -3 & -9 \\ 0 & 0 & -12 \end{bmatrix}.$$

Thus we have

$$\mathbf{L}_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 2 & 1 \end{bmatrix}.$$

We now need to compute $\mathbf{L}_1^{-1}$ and $\mathbf{L}_2^{-1}$. These matrices turn out to have an amazingly simple form:

$$\mathbf{L}_1^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix}, \qquad \mathbf{L}_2^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -2 & 1 \end{bmatrix}.$$

We have inverted these matrices simply by flipping the sign on the nonzero entries below the diagonal! Check that this works out by verifying $\mathbf{L}_1^{-1}\mathbf{L}_1 = \mathbf{L}_2^{-1}\mathbf{L}_2 = \mathbf{I}$. This is what Trefethen and Bau call 'the first stroke of luck'.

Now we need to compute the product $\mathbf{L}_1^{-1}\mathbf{L}_2^{-1} =: \mathbf{L}$ to get the $\mathbf{L}$ factor in the LU factorization. Again, this product has a basic form:

$$\mathbf{L} = \mathbf{L}_1^{-1}\mathbf{L}_2^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -2 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & -2 & 1 \end{bmatrix}.$$

The subdiagonal entries of $\mathbf{L}_1^{-1}$ and $\mathbf{L}_2^{-1}$ are simply copied into the matrix $\mathbf{L}$. This holds in general: if we know the individual Gauss transformation matrices $\mathbf{L}_1, \ldots, \mathbf{L}_{n-1}$, then we can write down $\mathbf{L}$ with no further computations. Trefethen and Bau call this 'the second stroke of luck'. Finally we confirm the $\mathbf{LU}$ factorization:

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 4 \\ 2 & 1 & -1 \\ -1 & 4 & 2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & -2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 4 \\ 0 & -3 & -9 \\ 0 & 0 & -12 \end{bmatrix} = \mathbf{LU}.$$

Bird's Eye View. Now consider the more general case:

$$\mathbf{A} = \begin{bmatrix} \alpha & \mathbf{u}^* \\ \mathbf{v} & \mathbf{C} \end{bmatrix}.$$

We want to apply a Gauss transformation to zero out the $\mathbf{v}^*$ vector. If we define

$$\mathbf{L}_1 = \begin{bmatrix} 1 & \mathbf{0} \\ -\frac{1}{\alpha}\mathbf{v} & \mathbf{I} \end{bmatrix},$$

then

$$\mathbf{L}_1\mathbf{A} = \begin{bmatrix} \alpha & \mathbf{u}^* \\ \mathbf{0} & \mathbf{C} - \frac{1}{\alpha}\mathbf{v}\mathbf{u}^* \end{bmatrix}.$$

Note that

$$\mathbf{L}_1^{-1} = \begin{bmatrix} 1 & \mathbf{0} \\ \frac{1}{\alpha}\mathbf{v} & \mathbf{I} \end{bmatrix},$$

and hence

$$\begin{aligned} \mathbf{A} &= \begin{bmatrix} 1 & \mathbf{0} \\ \frac{1}{\alpha}\mathbf{v} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \alpha & \mathbf{u}^* \\ \mathbf{0} & \mathbf{C} - \frac{1}{\alpha}\mathbf{v}\mathbf{u}^* \end{bmatrix} \\ &= \begin{bmatrix} 1 & \mathbf{0} \\ \frac{1}{\alpha}\mathbf{v} & \mathbf{I} \end{bmatrix} \begin{bmatrix} 1 & \mathbf{0} \\ \mathbf{0} & \mathbf{C} - \frac{1}{\alpha}\mathbf{v}\mathbf{u}^* \end{bmatrix} \begin{bmatrix} \alpha & \mathbf{u}^* \\ \mathbf{0} & \mathbf{I} \end{bmatrix}. \end{aligned}$$

This final factorization reveals part of the final $\mathbf{L}$ matrix (first column of left matrix), part of the final $\mathbf{U}$ matrix (first row of the right matrix), and the *active submatrix* $\mathbf{C} - \frac{1}{\alpha}\mathbf{v}\mathbf{u}^*$. Repeat this procedure to zero out the subdiagonal entries in the first column of the active submatrix, and this will provide the second column of $\mathbf{L}$ and the second row of $\mathbf{U}$.

This procedure is encapsulated in the following algorithm.

```
$\mathbf{U} = \mathbf{A}$, $\mathbf{L} = \mathbf{I}$
for $k = 1, 2, \ldots, n-1$
    for $j = k+1, \ldots, n$
        $\ell_{j,k} = u_{j,k}/u_{k,k}$
        for $m = k, \ldots, n$
            $u_{j,m} = u_{j,m} - \ell_{j,k}u_{k,m}$
        end
    end
end
```

This algorithm requires approximately $\frac{2}{3}n^3$ floating point operations, *half as many operations as a QR factorization.*

6.1.1. Pivoting.

Unfortunately, this algorithm can fail for some matrices and give poor results for others. For example, if

$$\mathbf{A} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix},$$

then there would be a division by zero at the first step, even though the matrix $\mathbf{A}$ is invertible. A small modification of this matrix gives a famous example for which the LU factorization algorithm works, but gives an incorrect answer due to a floating point rounding error:

$$\mathbf{A} = \begin{bmatrix} \varepsilon & 1 \\ 1 & 1 \end{bmatrix}.$$

The above algorithm would produce

$$\mathbf{A} = \mathbf{L}\mathbf{U} = \begin{bmatrix} 1 & 0 \\ 1/\varepsilon & 1 \end{bmatrix} \begin{bmatrix} \varepsilon & 1 \\ 0 & 1 - 1/\varepsilon \end{bmatrix}.$$

However, if ε is smaller than $\varepsilon_{\text{mach}}$, the machine epsilon value for the floating point number system, then the second matrix would be rounded to

$$\widetilde{\mathbf{U}} := \begin{bmatrix} \varepsilon & 1 \\ 0 & -1/\varepsilon \end{bmatrix}.$$

This small relative change in the (2,2) entry of $\mathbf{U}$, which is perfectly consistent with the floating point axioms we discussed earlier in the semester, turns out to be crucial. Assuming no rounding errors are made in $\mathbf{L}$, so that $\widetilde{\mathbf{L}} := \mathbf{L}$, we would have

$$\widetilde{\mathbf{L}}\widetilde{\mathbf{U}} = \begin{bmatrix} 1 & 0 \\ 1/\varepsilon & 1 \end{bmatrix} \begin{bmatrix} \varepsilon & 1 \\ 0 & -1/\varepsilon \end{bmatrix} = \begin{bmatrix} \varepsilon & 1 \\ 1 & 0 \end{bmatrix} =: \widetilde{\mathbf{A}},$$

which has an error of size $1 \gg \varepsilon$:

$$\mathbf{A} - \widetilde{\mathbf{A}} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}.$$

Perhaps we would not mind this so much if $\mathbf{A}$ were close to singular, i.e., if $\|\mathbf{A}^{-1}\|$ or $\kappa(\mathbf{A}) = \|\mathbf{A}\|\|\mathbf{A}^{-1}\|$ were large. But that is not the case when ε is small, for

$$\mathbf{A}^{-1} = \frac{1}{1-\varepsilon}\begin{bmatrix} -1 & 1 \\ 1 & -\varepsilon \end{bmatrix}.$$

A potential remedy to the problem observed here is *pivoting*: swapping rows (and possibly columns, too) so that we never add a large multiple of one row to another row.

Partial pivoting. The most popular form of pivoting uses row interchanges at each elimination step to ensure that the pivot value, α, that we are about to invert is at least as large as any entry below it. We return to our small example:

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 4 \\ 2 & 1 & -1 \\ -1 & 4 & 2 \end{bmatrix}.$$

First, we scan down column 1 to find the largest magnitude entry: this is 2, in the (2,1) position. Hence:

Swap ROW 1 and ROW 2

which can be encoded as premultiplication of a by a *permutation matrix*:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 2 & 4 \\ 2 & 1 & -1 \\ -1 & 4 & 2 \end{bmatrix} = \begin{bmatrix} 2 & 1 & -1 \\ 1 & 2 & 4 \\ -1 & 4 & 2 \end{bmatrix},$$

which we will write as $\mathbf{P}_1\mathbf{A}$. Now, we can proceed with the first elimination step, zeroing out subdiagonal entries in the first column:

$$\text{ROW } 2 \longleftarrow \text{ROW } 2 - \tfrac{1}{2} \times \text{ROW } 1;$$

$$\text{ROW } 3 \longleftarrow \text{ROW } 3 + \tfrac{1}{2} \times \text{ROW } 1.$$

Notice that this procedure ensures that the coefficient multiplying ROW 1 will not exceed 1. We encode these operations in the following matrix-matrix product:

$$\begin{bmatrix} 1 & 0 & 0 \\ -1/2 & 1 & 0 \\ 1/2 & 0 & 1 \end{bmatrix}\begin{bmatrix} 2 & 1 & -1 \\ 1 & 2 & 4 \\ -1 & 4 & 2 \end{bmatrix} = \begin{bmatrix} 2 & 1 & -1 \\ 0 & 3/2 & 9/2 \\ 0 & 9/2 & 3/2 \end{bmatrix}.$$

Again, the unit lower triangular matrix is labeled $\mathbf{L}_1$:

$$\mathbf{L}_1 = \begin{bmatrix} 1 & 0 & 0 \\ -1/2 & 1 & 0 \\ 1/2 & 0 & 1 \end{bmatrix}.$$

Before zeroing out the subdiagonal entry in the second column, we must scan down the second column to find the largest magnitude entry on the diagonal or below it. That largest entry is 9/2 in the (3,2) position of $\mathbf{L}_1\mathbf{P}_1\mathbf{A}$, and hence:

$$\text{Swap ROW 2 and ROW 3,}$$

which is encoded in a matrix $\mathbf{P}_2$:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 2 & 1 & -1 \\ 0 & 3/2 & 9/2 \\ 0 & 9/2 & 3/2 \end{bmatrix} = \begin{bmatrix} 2 & 1 & -1 \\ 0 & 9/2 & 3/2 \\ 0 & 3/2 & 9/2 \end{bmatrix}.$$

Now the final elimination step takes the form:

$$\text{ROW } 3 \leftarrow \text{ROW } 3 - \tfrac{1}{3} \times \text{ROW } 2,$$

resulting in

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1/3 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & -1 \\ 0 & 9/2 & 3/2 \\ 0 & 3/2 & 9/2 \end{bmatrix} = \begin{bmatrix} 2 & 1 & -1 \\ 0 & 9/2 & 3/2 \\ 0 & 0 & 4 \end{bmatrix}.$$

Altogether, we have computed the factorization

$$\mathbf{L}_2\mathbf{P}_2\mathbf{L}_1\mathbf{P}_1\mathbf{A} = \mathbf{U}.$$

It remains to show that one can disentangle the permutation matrices and the lower triangular matrices to obtain a factorization of the form

$$\mathbf{PA} = \mathbf{LU},$$

where $\mathbf{P}$ is a permutation matrix that aggregates all the previous row swaps. First, invert the lower triangular matrix $\mathbf{L}_2$ to obtain

$$\mathbf{P}_2\mathbf{L}_1\mathbf{P}_1\mathbf{A} = \mathbf{L}_2^{-1}\mathbf{U}.$$

We would like to somehow move the $\mathbf{P}_2$ matrix to the other side of $\mathbf{L}_1$, but these matrices do not commute in general. The trick is to substitute $\mathbf{I} = \mathbf{P}_2^{-1}\mathbf{P}_2$ in just the right place:

$$\mathbf{P}_2\mathbf{L}_1\mathbf{P}_2^{-1}\mathbf{P}_2\mathbf{P}_1\mathbf{A} = \mathbf{L}_2^{-1}\mathbf{U}.$$

Let $\widetilde{\mathbf{L}}_1 := \mathbf{P}_2\mathbf{L}_1\mathbf{P}_2^{-1}$, and note that in our case

$$\widetilde{\mathbf{L}}_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ -1/2 & 1 & 0 \\ 1/2 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1/2 & 1 & 0 \\ -1/2 & 0 & 1 \end{bmatrix},$$

which again is a unit lower triangular matrix with the structure of a Gauss transformation, i.e., only one column has nonzero entries below the diagonal, This convenient fact holds in general; Trefethen and Bau call it 'the third stroke of luck'. We can thus write our decomposition in the form $\mathbf{PA} = \mathbf{LU}$:

$$\mathbf{P}_2\mathbf{P}_1\mathbf{A} = \widetilde{\mathbf{L}}_1^{-1}\mathbf{L}_2^{-1}\mathbf{U}.$$

The details get a little more complicated for larger matrices. For example, if $\mathbf{A} \in \mathbb{C}^{4\times 4}$, we have

$$\mathbf{L}_3\mathbf{P}_3\mathbf{L}_2\mathbf{P}_2\mathbf{L}_1\mathbf{P}_1\mathbf{A} = \mathbf{U}.$$

Define $\widetilde{\mathbf{L}}_3 := \mathbf{L}_3$, and invert to get

$$\mathbf{P}_3\mathbf{L}_2\mathbf{P}_2\mathbf{L}_1\mathbf{P}_1\mathbf{A} = \widetilde{\mathbf{L}}_3^{-1}\mathbf{U}.$$

Now define $\widetilde{\mathbf{L}}_2 := \mathbf{P}_3\mathbf{L}_2\mathbf{P}_3^{-1}$, and note that

$$\widetilde{\mathbf{L}}_2\mathbf{P}_3\mathbf{P}_2\mathbf{L}_1\mathbf{P}_1\mathbf{A} = \widetilde{\mathbf{L}}_3^{-1}\mathbf{U}.$$

It follows that

$$\mathbf{P}_3\mathbf{P}_2\mathbf{L}_1\mathbf{P}_1\mathbf{A} = \widetilde{\mathbf{L}}_2^{-1}\widetilde{\mathbf{L}}_3^{-1}\mathbf{U}.$$

Now define $\widetilde{\mathbf{L}}_1 = \mathbf{P}_3\mathbf{P}_2\mathbf{L}_1\mathbf{P}_2^{-1}\mathbf{P}_3^{-1}$, giving

$$\widetilde{\mathbf{L}}_1\mathbf{P}_3\mathbf{P}_2\mathbf{P}_1\mathbf{A} = \widetilde{\mathbf{L}}_2^{-1}\widetilde{\mathbf{L}}_3^{-1}\mathbf{U}.$$

Finally, we have

$$\mathbf{P}_3\mathbf{P}_2\mathbf{P}_1\mathbf{A} = \widetilde{\mathbf{L}}_1^{-1}\widetilde{\mathbf{L}}_2^{-1}\widetilde{\mathbf{L}}_3^{-1}\mathbf{U},$$

and if we define

$$\mathbf{P} := \mathbf{P}_3\mathbf{P}_2\mathbf{P}_1,$$

and

$$\mathbf{L} := \widetilde{\mathbf{L}}_1^{-1}\widetilde{\mathbf{L}}_2^{-1}\widetilde{\mathbf{L}}_3^{-1},$$

then we have the factorization

$$\mathbf{PA} = \mathbf{LU}.$$

Thus our factorization with step-by-step row swapping is equivalent to computing the no-pivot LU factorization of a matrix $\mathbf{PA}$ whose rows have been 'perfectly ordered' in advance. The details may look quite technical, but this process is readily organized into efficient software.

Stability. We have seen above that Gaussian elimination without pivoting can behave badly when rounding errors are made in the decomposition. Trefethen and Bau do a fine job of describing the behavior of Gaussian elimination in floating-point arithmetic. We summarize the main results here.

If standard Gaussian elimination (no pivoting) is applied to a matrix $\mathbf{A}$, and no zero pivots are encountered, then the algorithm computes, in finite precision arithmetic, a factorization

$$\widehat{\mathbf{L}}\widehat{\mathbf{U}} = \mathbf{A} + \delta\mathbf{A},$$

where

$$\frac{\|\delta\mathbf{A}\|}{\|\widehat{\mathbf{L}}\|\|\widehat{\mathbf{U}}\|} = O(\varepsilon_{\mathrm{mach}}).$$

Unfortunately, as we saw in the 2×2 example above, $\|\widehat{\mathbf{L}}\|\|\widehat{\mathbf{U}}\|$ can be very large compared to $\|\mathbf{A}\|$, which means that the error $\|\delta\mathbf{A}\|$ can be unsatisfactorily big.

Partial pivoting ensures that no entry of $\mathbf{L}$ exceeds 1, and hence $\|\mathbf{L}\|$ remains modest. However, $\|\mathbf{U}\|$ can still grow very large, as is evident if one applies Gaussian elimination to Kahan's famous example

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ -1 & 1 & 0 & 0 & 1 \\ -1 & -1 & 1 & 0 & 1 \\ -1 & -1 & -1 & 1 & 1 \\ -1 & -1 & -1 & -1 & 1 \end{bmatrix}.$$

(Partial pivoting will not make any row swaps on this example.) The (j, n) entry of $\mathbf{U}$ equals 2^{j-1}, even though all the entries in $\mathbf{A}$ are small. This motivates the definition of the *growth factor*

$$\rho := \frac{\max_{j,k} |u_{j,k}|}{\max_{j,k} |a_{j,k}|}.$$

For Kahan's example, $\rho = 2^{n-1}$, the largest possible value.

One can show that Gaussian elimination with partial pivoting computes a factorization

$$\widehat{\mathbf{L}}\widehat{\mathbf{U}} = \widehat{\mathbf{P}}\mathbf{A} + \delta\mathbf{A}$$

in finite precision arithmetic with

$$\frac{\|\delta\mathbf{A}\|}{\|\mathbf{A}\|} = O(\rho\varepsilon_{\text{mach}}).$$

Provided ρ is not large, the factorization will be accurate.

One can do even better with *complete pivoting*, which pivots with the largest magnitude entry in the entire active submatrix at each iteration, rather than the largest magnitude entry in the first column of the active submatrix. This more expensive form of pivoting produces a factorization of the form

$$\mathbf{PAQ} = \mathbf{LU},$$

where $\mathbf{P}$ and $\mathbf{Q}$ are permutation matrices that encode row and columns swaps, respectively.

In practice, the faster speed of partial pivoting trumps the security of complete pivoting. Though partial pivoting can theoretically lead to disasters, no one has knowingly encountered an example from a real application that exhibits the kind of growth seen for Kahan's example. Thus MATLAB's \ (backslash) uses Gaussian elimination with partial pivoting for square, non-Hermitian $\mathbf{A}$. We shall see in the next lecture that a useful alternative is available when $\mathbf{A}$ is Hermitian positive definite.

Lecture 36: Cholesky Factorization

6.1.2. Cholesky factorization.

When $\mathbf{A} \in \mathbb{C}^{n\times n}$ is Hermitian ($\mathbf{A} = \mathbf{A}^*$) or $\mathbf{A} \in \mathbb{R}^{n\times n}$ is real symmetric ($\mathbf{A} = \mathbf{A}^T$), the Gaussian elimination algorithm takes a special form. A wide variety of applications, including many discretized partial differential equations and optimization problems, give rise to such matrices, so it is worth taking a moment to consider ways to exploit this structure. This discussion follows Golub & Van Loan, *Matrix Computations*, §4.2 and Trefethen & Bau, *Numerical Linear Algebra*, §23, where more details can be found.

First decompose the Hermitian matrix $\mathbf{A} \in \mathbb{C}^{n\times n}$ into the form

$$\mathbf{A} = \begin{bmatrix} \alpha & \mathbf{v}^* \\ \mathbf{v} & \mathbf{C} \end{bmatrix}, \tag{36.1}$$

where $\alpha \in \mathbb{C}$, $\mathbf{v} \in \mathbb{C}^{n-1}$, and $\mathbf{C} \in \mathbb{C}^{(n-1)\times(n-1)}$. Provided $\alpha \neq 0$, one pass through the outer loop of the Gaussian elimination algorithm produces the decomposition

$$\mathbf{A} = \begin{bmatrix} \alpha & \mathbf{v}^* \\ \mathbf{v} & \mathbf{C} \end{bmatrix} = \begin{bmatrix} 1 & \mathbf{0} \\ \frac{1}{\alpha}\mathbf{v} & \mathbf{I} \end{bmatrix} \begin{bmatrix} 1 & \mathbf{0} \\ \mathbf{0} & \mathbf{C} - \frac{1}{\alpha}\mathbf{v}\mathbf{v}^* \end{bmatrix} \begin{bmatrix} \alpha & \mathbf{v}^* \\ \mathbf{0} & \mathbf{I} \end{bmatrix}. \tag{36.2}$$

These final three matrices represent a partial $\mathbf{LU}$ factorization: The first column of the first matrix becomes the first column of $\mathbf{L}$; the first row of the third matrix becomes the first row of $\mathbf{U}$. Further steps of Gaussian elimination reduce the central matrix to the identity.

This $\mathbf{LU}$ factorization exhibits an unnatural asymmetry. The standard Gaussian elimination algorithm forces $\mathbf{L}$ to be *unit* lower triangular, i.e., to have ones on the main diagonal. Correspondingly, the diagonal entires of $\mathbf{U}$ contain the pivots, which of course generally differ from 1. Relaxing the requirement that $\mathbf{L}$ be unit lower triangular provides sufficient liberty to enforce symmetry in the factorization, so that the first row of $\mathbf{U}$ will simply be the conjugate-transpose of the first column of $\mathbf{L}$. In particular, we can replace (36.2) by the factorization

$$\mathbf{A} = \begin{bmatrix} \alpha & \mathbf{v}^* \\ \mathbf{v} & \mathbf{C} \end{bmatrix} = \begin{bmatrix} \sqrt{\alpha} & \mathbf{0} \\ \frac{1}{\sqrt{\alpha}}\mathbf{v} & \mathbf{I} \end{bmatrix} \begin{bmatrix} 1 & \mathbf{0} \\ \mathbf{0} & \mathbf{C} - \frac{1}{\alpha}\mathbf{v}\mathbf{v}^* \end{bmatrix} \begin{bmatrix} \sqrt{\alpha} & \frac{1}{\sqrt{\alpha}}\mathbf{v}^* \\ \mathbf{0} & \mathbf{I} \end{bmatrix}.$$

Continuing in the same fashion at later stages of the elimination yields $\mathbf{U} = \mathbf{L}^*$, so that $\mathbf{A} = \mathbf{L}\mathbf{L}^*$.† Of course, this assumes that Gaussian elimination does not break down (i.e., encounter $\alpha = 0$). Furthermore, when $\mathbf{A}$ is real, the matrix $\mathbf{L}$ should be real as well, so we must beware of the case of $\alpha \le 0$. It turns out that for many matrices, we can ensure that $\alpha > 0$ and that this property will be inherited by the submatrix $\mathbf{C} - \frac{1}{\alpha}\mathbf{v}\mathbf{v}^*$. Toward this end, recall the following definition from our discussion of the singular value decomposition.

Definition. A matrix $\mathbf{A} \in \mathbb{C}^{n\times n}$ is *Hermitian positive definite* (HPD) provided $\mathbf{A} = \mathbf{A}^*$ and $\mathbf{x}^*\mathbf{A}\mathbf{x} > 0$ for all nonzero $\mathbf{x} \in \mathbb{C}^n$.

Positive definite matrices are endowed with many beautiful properties. For example, a Hermitian matrix is positive definite if and only if all its eigenvalues are positive. (They must be real since the matrix is Hermitian.)

†The claim that $\mathbf{U} = \mathbf{L}^*$ requires that α be a positive real number: α must be real because $\mathbf{A}$ is Hermitian; we will soon see that $\alpha > 0$ for a broad class of interesting matrices.

Another important property of HPD matrices is that they always possess a Cholesky factorization. Proving this is our next task. First observe that if $\mathbf{A}$, written in the form (36.1), is HPD, then α must be a positive real number. This follows from the fact that $\mathbf{x}^*\mathbf{A}\mathbf{x} > 0$ for all nonzero $\mathbf{x}$, so in particular,

$$[1 \quad \mathbf{0}] \begin{bmatrix} \alpha & \mathbf{v}^* \\ \mathbf{v} & \mathbf{C} \end{bmatrix} \begin{bmatrix} 1 \\ \mathbf{0} \end{bmatrix} = \alpha > 0.$$

Consequently, there is no breakdown at the first step of the Cholesky factorization for any HPD matrix.

As we aim to prove that no breakdown will occur at any step of the factorization, we must analyze the 'unfactored' submatrix that remains after the first step,

$$\mathbf{C} - \frac{1}{\alpha}\mathbf{v}\mathbf{v}^*.$$

The second step of Cholesky factorization simply factors this submatrix in the same fashion that $\mathbf{A}$ was factored at the first step. Thus if we can show the unfactored submatrix is HPD, then we can recursively apply the above result (which proved that no breakdown can occur at the first step of Cholesky factorization of an HPD matrix) to show that no breakdown will occur at the second step, or any future step.

It is clear that $\mathbf{C} - \frac{1}{\alpha}\mathbf{v}\mathbf{v}^*$ is Hermitian, since $\mathbf{C} = \mathbf{C}^*$ and $(\mathbf{v}\mathbf{v}^*)^* = \mathbf{v}\mathbf{v}^*$. Now we need to show that $\mathbf{C} - \frac{1}{\alpha}\mathbf{v}\mathbf{v}^*$ is positive definite, i.e., $\mathbf{y}^*\mathbf{C}\mathbf{y} - \frac{1}{\alpha}(\mathbf{y}^*\mathbf{v})(\mathbf{v}^*\mathbf{y}) > 0$ for all nonzero $\mathbf{y} \in \mathbb{C}^{n-1}$. We can derive this fact from the positive definiteness of $\mathbf{A}$. Since $\mathbf{x}^*\mathbf{A}\mathbf{x} > 0$ for all nonzero $\mathbf{x} \in \mathbb{C}^n$, it must be that

$$\mathbf{x}^*\mathbf{A}\mathbf{x} = \begin{bmatrix} \xi \\ \mathbf{y} \end{bmatrix}^* \begin{bmatrix} \alpha & \mathbf{v}^* \\ \mathbf{v} & \mathbf{C} \end{bmatrix} \begin{bmatrix} \xi \\ \mathbf{y} \end{bmatrix} = \alpha\xi\overline{\xi} + \overline{\xi}\mathbf{v}^*\mathbf{y} + \xi\mathbf{y}^*\mathbf{v} + \mathbf{y}^*\mathbf{C}\mathbf{y} > 0 \qquad (36.3)$$

for all $\xi \in \mathbb{C}$ and nonzero $\mathbf{y} \in \mathbb{C}^{n-1}$. Now choose $\xi = -\mathbf{v}^*\mathbf{y}/\alpha$, so that (36.3) implies

$$\alpha\frac{(\mathbf{v}^*\mathbf{y})(\mathbf{y}^*\mathbf{v})}{\alpha^2} - \frac{(\mathbf{y}^*\mathbf{v})(\mathbf{v}^*\mathbf{y})}{\alpha} - \frac{(\mathbf{v}^*\mathbf{y})(\mathbf{y}^*\mathbf{v})}{\alpha} + \mathbf{y}^*\mathbf{C}\mathbf{y} > 0.$$

Canceling like terms in this formula yields

$$\mathbf{y}^*\mathbf{C}\mathbf{y} - \frac{1}{\alpha}\mathbf{y}^*\mathbf{v}\mathbf{v}^*\mathbf{y} = \mathbf{y}^*\Big(\mathbf{C} - \frac{1}{\alpha}\mathbf{v}\mathbf{v}^*\Big)\mathbf{y} > 0.$$

As this holds for all nonzero $\mathbf{y} \in \mathbb{C}^{n-1}$, we have proved that the submatrix $\mathbf{C} - \frac{1}{\alpha}\mathbf{v}\mathbf{v}^*$ is HPD, and hence the next step of Cholesky factorization will not break down. Applying this result recursively proves that each successive unfactored submatrix is HPD, and thus we can compute a complete Cholesky factorization without ever breaking down.

Theorem. Every Hermitian positive definite matrix $\mathbf{A}$ can be written in the form $\mathbf{A} = \mathbf{L}\mathbf{L}^*$, where $\mathbf{L}$ is a lower triangular matrix.

Beyond its aesthetic appeal, the Cholesky factorization gives several performance advantages. Most importantly, $\mathbf{L}$ can be computed in roughly $\frac{1}{3}n^3$ floating point operations, as opposed to the usual $\frac{2}{3}n^3$ operations required for standard Gaussian elimination. Though we have not addressed numerical stability, it turns out that the Cholesky factorization is stable without any need for pivoting. Finally, note that we need only store the factor $\mathbf{L}$, which has $\frac{1}{2}n(n+1)$ nonzero entries, as opposed to the n^2 entries needed to store $\mathbf{L}$ and $\mathbf{U}$ in standard Gaussian elimination.

The LDL* factorization. One aspect of the Cholesky factorization that might trouble you is the need to compute $\sqrt{\alpha}$, since the square root will be more expensive to compute than the basic floating point operations like addition and multiplication. A variant, called the 'square root-free Cholesky factorization' takes the form $\mathbf{A} = \mathbf{LDL}^*$. Its derivation closely resembles that of the standard Cholesky factorization, with the first step of the factorization now taking the form

$$\mathbf{A} = \begin{bmatrix} \alpha & \mathbf{v}^* \\ \mathbf{v} & \mathbf{C} \end{bmatrix} = \begin{bmatrix} 1 & \mathbf{0} \\ \frac{1}{\alpha}\mathbf{v} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \alpha & \mathbf{0} \\ \mathbf{0} & \mathbf{C} - \frac{1}{\alpha}\mathbf{v}\mathbf{v}^* \end{bmatrix} \begin{bmatrix} 1 & \frac{1}{\alpha}\mathbf{v}^* \\ \mathbf{0} & \mathbf{I} \end{bmatrix}.$$

As before, $\mathbf{A}$ is factored into a trio of matrices, with the first and third building $\mathbf{L}$ and $\mathbf{L}^*$. However, the matrix $\mathbf{L}$ is now unit lower triangular, and the central matrix is no longer transformed into the identity, but rather a diagonal matrix $\mathbf{D}$ whose diagonal entries contain the pivots.

What happens when $\mathbf{A}$ is Hermitian but not positive definite? Pivoting in general is necessary. Among the options is the Bunch–Kaufman algorithm, which computes

$$\mathbf{PAP}^* = \mathbf{LDL}^*,$$

where $\mathbf{P}$ is a permutation matrix that encodes symmetry-preserving row and column interchanges, and $\mathbf{D}$ is a *block diagonal* matrix, with 1-by-1 and 2-by-2 matrices on the main diagonal. See Golub & Van Loan, *Matrix Computations*, §4.4 for details and alternatives.

Lecture 37: Eigenvalue Computations

6.2. Eigenvalue Computations.

Eigenvalue computations play an essential role in scientific computing. We have already seen that the linear dynamical system

$$\mathbf{x}'(t) = \mathbf{A}\mathbf{x}(t), \quad \mathbf{x}(0) = \mathbf{x}_0$$

has the solution

$$\mathbf{x}(t) = \mathrm{e}^{t\mathbf{A}}\mathbf{x}_0,$$

and that this solution will decay, $\|\mathbf{x}(t)\| \to 0$ provided all the eigenvalues of $\mathbf{A}$ are in the left half of this complex plane. This simple model problem turns out to be of central importance to stability theory for dynamical systems, and consequently it arises in many important applications. Here is a typical scenario:

- For a nonlinear dynamical system $\mathbf{y}'(t) = \mathbf{f}(\mathbf{y}(t))$, somehow find a *steady state* $\widehat{\mathbf{y}}$, that is, find some constant vector $\widehat{\mathbf{y}}$ such that $\mathbf{f}(t, \widehat{\mathbf{y}}) = \mathbf{0}$.
- We would like to determine if this steady state is *stable*, i.e., if $\mathbf{y}(t)$ is near $\widehat{\mathbf{y}}$, will $\mathbf{y}(t) \to \widehat{\mathbf{y}}$ as $t \to \infty$? Note that
$$\frac{\mathrm{d}}{\mathrm{d}t}(\mathbf{y}(t) - \widehat{\mathbf{y}}) = \mathbf{y}'(t) = \mathbf{f}(\mathbf{y}(t)),$$
since $\widehat{\mathbf{y}}$ is constant. This means that the rate at which $\mathbf{y}(t)$ is attracted or repelled from $\widehat{\mathbf{y}}$ is controlled by $\mathbf{f}(\mathbf{y}(t))$.
- Provided $\|\mathbf{y}(t) - \widehat{\mathbf{y}}\|$ is small, we can expand $\mathbf{f}(\mathbf{y}(t))$ in a Taylor series to obtain
$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}t}(\mathbf{y}(t) - \widehat{\mathbf{y}}) &= \mathbf{f}(\mathbf{y}(t)) \\ &= \mathbf{f}(\widehat{\mathbf{y}}) + \mathbf{A}(\mathbf{y}(t) - \widehat{\mathbf{y}}) + O(\|\mathbf{y}(t) - \widehat{\mathbf{y}}\|^2).\end{aligned}$$
- Since $\widehat{\mathbf{y}}$ is a steady state, $\mathbf{f}(\widehat{\mathbf{y}}) = \mathbf{0}$, and provided $\|\mathbf{y}(t) - \widehat{\mathbf{y}}\|$ is small, it is conventional to make the *approximation*
$$\frac{\mathrm{d}}{\mathrm{d}t}(\mathbf{y}(t) - \widehat{\mathbf{y}}) = \mathbf{A}(\mathbf{y}(t) - \widehat{\mathbf{y}}),$$
or simply
$$\mathbf{x}'(t) = \mathbf{A}\mathbf{x}(t),$$
where $\mathbf{x}(t)$ is the deviation of $\mathbf{y}(t)$ from $\widehat{\mathbf{y}}$, i.e., $\mathbf{x}(t) := \mathbf{y}(t) - \widehat{\mathbf{y}}$.
- If all the eigenvalues of this linearization matrix $\mathbf{A}$ fall in the left half of the complex plane, then $\mathbf{x}(t) \to \mathbf{0}$ as $t \to \infty$, suggesting that $\mathbf{y}(t) \to \widehat{\mathbf{y}}$ as $t \to \infty$. In this case the steady state $\widehat{\mathbf{y}}$ is said to be *stable*.

It must be emphasized that this mode of analysis is far from fool-proof.[†] Nevertheless, this brand of analysis is a staple of scientific computing, and proves descriptive in a wide variety of settings.

[†]See, for example, Section 33 of Trefethen and E., *Spectra and Pseudospectra: The Behavior of Nonnormal Matrices and Operators*, Princeton, 2005.

This is but one of numerous important settings that give rise to eigenvalue problems. So now how does one go about computing these eigenvalues? There are a variety of techniques, some most appropriate for dense matrices (say, with dimension $n \leq 1000$), others better suited for large, sparse matrices. MATLAB's `eig` command uses the QR algorithm described below for the former, while `eigs` tackles large-scale problems via the 'implicitly restarted Arnoldi method'. This latter method, invented by Prof. Sorensen in the CAAM department, is taught in detail in CAAM 551.

Our time in this course is regrettably short, so we shall only enjoy a brief sample of the available algorithms. First, we begin with a very bad idea.

6.2.1. Computing Eigenvalues by Factoring the Characteristic Polynomial.

When one learns how to compute the eigenvalues of small matrices by hand in a linear algebra class, one inevitably encounters the *characteristic polynomial* of a matrix, defined as

$$p(\lambda) = \det(\lambda\mathbf{I} - \mathbf{A}).$$

The eigenvalues of $\mathbf{A}$ are those points $\lambda \in \mathbb{C}$ for which there exists some nonzero $\mathbf{v} \in \mathbb{C}^n$ such that $\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$, in other words, $(\lambda\mathbf{I} - \mathbf{A})\mathbf{v} = \mathbf{0}$ has a nontrivial solution, $\mathbf{v}$. This is equivalent to $\lambda\mathbf{I} - \mathbf{A}$ being non-invertible, and hence λ is an eigenvalue provided $\det(\lambda\mathbf{I} - \mathbf{A}) = 0$, i.e., $p(\lambda) = 0$. We first learn to compute the eigenvalues of $\mathbf{A}$ by factoring the characteristic polynomial. For example, the matrix

$$\mathbf{A} = \begin{pmatrix} 2 & 2 \\ 2 & 5 \end{pmatrix}$$

has characteristic polynomial

$$p(z) = (\lambda - 2)(\lambda - 5) - 4 = \lambda^2 - 7\lambda + 6 = (\lambda - 1)(\lambda - 6),$$

and hence the eigenvalues of $\mathbf{A}$ are 1 and 6. While this procedure works fine for small examples computed by hand, it turns out to be a terrible idea for problems of any appreciable size. It is difficult to accurately compute the roots of a polynomial in practice; small changes to coefficients of the characteristic polynomial can change the eigenvalues significantly. Suppose we have the matrix `A = diag(1:25)`, a Hermitian matrix with eigenvalues $1, 2, \ldots, 25$. In MATLAB, try

```
roots(poly(A))
```

The `poly(A)` command will construct the characteristic polynomial of a `A`, then the `roots` command computes the roots of the polynomial, i.e., the eigenvalues. Hence we expect MATLAB to return the values $1, 2, \ldots, 25$. Instead, we get:

```
25.06688344076654
24.02393410924768 + 0.72502281860060i
24.02393410924768 - 0.72502281860060i
22.28328229283404 + 1.86517379171456i
22.28328229283404 - 1.86517379171456i
20.00370925655028 + 2.66139122859282i
20.00370925655028 - 2.66139122859282i
17.49273885864172 + 2.91159377855117i
17.49273885864172 - 2.91159377855117i
15.03618870933039 + 2.63913115171857i
15.03618870933039 - 2.63913115171857i
```

```
12.81478707271267 + 1.96578674260601i
12.81478707271267 - 1.96578674260601i
10.92964560504511 + 1.01992698949984i
10.92964560504511 - 1.01992698949984i
 9.75727458256601
 8.99618639090924
 8.01361076890918
 6.99717556033484
 6.00031703510833
 4.99997972494391
 4.00000069862736
 2.99999998905382
 2.00000000005767
 0.99999999999985
```

In fact, 14 of the 25 eigenvalues MATLAB finds have significant imaginary parts, even though the true eigenvalues are real! From our discussion of *conditioning* and *stability*, one should suspect two potential culprits: The eigenvalues of this matrix are very sensitive to perturbations (i.e., they are *ill-conditioned*), or the algorithm itself is an *unstable* way to compute eigenvalues. It turns out that the eigenvalues of this matrix are perfectly conditioned: small changes to the matrix entries inflict only small changes to the eigenvalues. Unfortunately, the algorithm is completely unstable. A better approach is required. This famous example is due to J. H. Wilkinson, an early pioneer of numerical linear algebra who made essential contributions to the theory of backward error analysis and wrote a timely and highly influential book, *The Algebraic Eigenvalue Problem* (1965).

We now turn our attention to more promising algorithms.

6.2.2. Power Method.

Our first decent method is closely related to our earlier study of absolute stability for linear multistep methods for linear differential equations, where powers of a matrix magnified the components of a vector in certain eigenvector directions, depending on the location of the corresponding eigenvalues in the complex plane.

Given $\mathbf{A} \in \mathbb{C}^{n\times n}$ and a vector $\mathbf{x}_0 \in \mathbb{C}^n$, consider

$$\mathbf{x}_k = \mathbf{A}^k\mathbf{x}_0.$$

Suppose $\mathbf{A}$ is diagonalizable, $\mathbf{A} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^{-1}$, with $\mathbf{V} = [\mathbf{v}_1\ \mathbf{v}_2\ \cdots\ \mathbf{v}_n]$ and $\mathbf{\Lambda} = \mathrm{diag}(\lambda_1, \ldots, \lambda_n)$. We can express $\mathbf{x}_0$ as a linear combination of the eigenvectors of $\mathbf{A}$,

$$\mathbf{x}_0 = \sum_{j=1}^{n} \gamma_j \mathbf{v}_j,$$

where $\mathbf{V}^{-1}\mathbf{x}_0 = [\gamma_1, \ldots, \gamma_n]^T$. Since $\mathbf{A}^k\mathbf{v}_j = \lambda_j^k\mathbf{v}_j$, we have

$$\mathbf{x}_k = \mathbf{A}^k\mathbf{x}_0 = \sum_{j=1}^{n} \gamma_j \lambda_j^k \mathbf{v}_j.$$

Now suppose that

$$|\lambda_1| > |\lambda_2| \geq \cdots \geq |\lambda_n|.$$

Then we see that provided $\gamma_1 \neq 0$, then the $\gamma_1\lambda_1^k\mathbf{v}_1$ term will dominate in the expression for $\mathbf{x}_k$. The rate at which this component dominates is $|\lambda_1|/|\lambda_2|$.

To make this a practical algorithm, we need to normalize keep $\mathbf{x}_k$ from growing (or shrinking) in norm and thus causing overflow or underflow problems in floating point arithmetic. For example, we can set

$$\widehat{\mathbf{x}}_k = \mathbf{A}\mathbf{x}_{k-1}, \qquad \mathbf{x}_k = \frac{\widehat{\mathbf{x}}_k}{\|\widehat{\mathbf{x}}_k\|}.$$

If $|\lambda_1|/|\lambda_2|$ is large, then the power method is a very effective and inexpensive way to compute the largest magnitude eigenvalue. In practice, this ratio is usually not so large; also, we are often interested eigenvalues other than the largest magnitude one. For this reason the power method often shows up as a building block in a more sophisticated algorithm.

Inverse iteration. Suppose we wish to find an eigenvalue of our diagonalizable matrix $\mathbf{A}$ near some point $\mu \in \mathbb{C}$. Notice that the eigenvalues ν_j of the matrix $(\mu\mathbf{I} - \mathbf{A})^{-1}$ are simply

$$\nu_j = \frac{1}{\mu - \lambda_j};$$

the eigenvectors of $\mathbf{A}$ and $(\mu\mathbf{I} - \mathbf{A})^{-1}$ are identical. If there is one eigenvalue λ_j closer to μ than all the others, then we can find this eigenvalue by performing the power method on the matrix $(\mu\mathbf{I} - \mathbf{A})^{-1}$:

$$\widehat{\mathbf{x}}_k = (\mu\mathbf{I} - \mathbf{A})^{-1}\mathbf{x}_{k-1}, \qquad \mathbf{x}_k = \frac{\widehat{\mathbf{x}}_k}{\|\widehat{\mathbf{x}}_k\|}.$$

Now the vector $\mathbf{x}_k$ becomes dominated by the eigenvector of $\mathbf{A}$ whose eigenvalue is closest to μ. Each iteration requires computing $(\mu\mathbf{I} - \mathbf{A})^{-1}\mathbf{x}_{k-1}$. For a dense matrix $\mathbf{A}$, we can compute the LU factorization of $(\mu\mathbf{I} - \mathbf{A})^{-1}$ in $O(n^3)$ operations, and then each iteration of the power method will require $O(n^2)$ operations.

Rayleigh quotient iteration. Rayleigh quotient iteration is a variant of inverse iteration in which the point μ is updated at each step, hopefully to be a better approximation to an eigenvalue of $\mathbf{A}$:

$$\mu_{k-1} = \frac{\mathbf{x}_{k-1}^*\mathbf{A}\mathbf{x}_{k-1}}{\mathbf{x}_{k-1}^*\mathbf{x}_{k-1}}, \qquad \widehat{\mathbf{x}}_k = (\mu_{k-1}\mathbf{I} - \mathbf{A})^{-1}\mathbf{x}_{k-1}, \qquad \mathbf{x}_k = \frac{\widehat{\mathbf{x}}_k}{\|\widehat{\mathbf{x}}_k\|}.$$

Now an LU factorization of $(\mu_{k-1}\mathbf{I} - \mathbf{A})$ is required at each step, which is more expensive than standard inverse iteration. However, if μ_k is near an eigenvalue, then this iteration converges with fantastic speed: the error reduces *quadratically* for non-Hermitian $\mathbf{A}$, and *cubically* for Hermitian $\mathbf{A}$, due to a close connection to Newton's method for solving nonlinear equations.

6.2.3. Similarity transformations and the Schur form.

The simple iterations discussed above compute one eigenvalue at a time. What if we wish to compute the entire spectrum in one fell swoop? We have already seen the hazards of factoring the characteristic polynomial. The present method, remarkable though it may seem at first, turns out to be much better.

The eigenvalues of a diagonal or upper/lower triangular matrix can be read off immediately: they are simply the diagonal entries. One strategy for computing eigenvalues is thus to manipulate the matrix $\mathbf{A}$ into one of these forms, or something very close to it, while preserving the eigenvalues of $\mathbf{A}$ in the process. We affect this change using a series of *similarity transformations*.

Definition. Let $\mathbf{S}$ be any invertible matrix. Then $\mathbf{SAS}^{-1}$ is a *similarity transformation* of $\mathbf{A}$. The matrices $\mathbf{A}$ and $\mathbf{SAS}^{-1}$ are said to be *similar*. If $\mathbf{U}$ is a unitary matrix, then $\mathbf{UAU}^*$ is a *unitary similarity transformation.*

Similar matrices must have identical eigenvalues. Suppose that $(\lambda, \mathbf{v})$ is an eigenpair of $\mathbf{A}$, i.e., $\mathbf{Av} = \lambda\mathbf{v}$. Then premultiply by $\mathbf{S}$ to obtain

$$\mathbf{SA}(\mathbf{S}^{-1}\mathbf{S})\mathbf{v} = \lambda\mathbf{Sv},$$

and hence $(\lambda, \mathbf{Sv})$ is an eigenpair of $\mathbf{SAS}^{-1}$.

For reasons of numerical stability, we prefer similarity transformations with unitary matrices. The foundation of our ultimate algorithm is the important fact that any matrix can be reduced by such transformations to upper triangular form.

Theorem (Schur decomposition). For any matrix $\mathbf{A} \in \mathbb{C}^{n\times n}$ there exists a unitary matrix $\mathbf{U} \in \mathbb{C}^{n\times n}$ and upper triangular matrix $\mathbf{T} \in \mathbb{C}^{n\times n}$ such that

$$\mathbf{A} = \mathbf{UTU}^*.$$

Given a Schur decomposition of $\mathbf{A}$, the eigenvalues are obvious: they are simply the diagonal entries of $\mathbf{T}$. Hence we seek an algorithm that constructed the unitary matrix $\mathbf{U}$.

Unfortunately, it is *impossible* to find $\mathbf{U}$ when $n > 4$. Precisely, *no algorithm can find the eigenvalues of a general matrix of dimension $n > 4$ in finitely many basic arithmetic operations.* This limitation is a consequence of the relationship between eigenvalues and polynomial roots. With the polynomial $p(z) = z^n + c_{n-1}z^{n-1} + \cdots + c_1 z + c_0$ is associated the *companion matrix*

$$\mathbf{A} = \begin{pmatrix} -c_{n-1} & -c_{n-2} & \cdots & -c_1 & -c_0 \\ 1 & & & & \\ & 1 & & & \\ & & \ddots & & \\ & & & 1 & \end{pmatrix},$$

whose eigenvalues are the roots of $p(z)$. (Unspecified entries in $\mathbf{A}$ are zero.) Thus, if we could find eigenvalues of all matrices of arbitrary dimension, we could also factor polynomials. Abel and Galois proved in the early nineteenth century that there was no way to factor polynomials of degree $n > 4$ in finitely many basic algebraic operations. This result, established years before the first matrix was formally written down, ensures that any algorithm that determines eigenvalues of a general matrix must be inherently *iterative*, and the resulting eigenvalues cannot be determined exactly.

6.2.4. QR algorithm for computing eigenvalues.

Fortunately, we need not end on such a sad note. While it is theoretically impossible to find the Schur decomposition, it turns out to be quite tractable in practice, in that one can use unitary similarity transformations to reduce $\mathbf{A}$ not exactly to an upper triangular matrix, but to a matrix whose lower triangular entries are as small as you like. The algorithm that delivers this practical Schur decomposition is called the *QR eigenvalue algorithm*, because it involves repeated use of the QR factorization we studied in connection with linear systems and least squares problems. The QR eigenvalue algorithm has its theoretical roots in the 'LR algorithm' proposed by Heinz Rutishauser in the 1950s; John Francis and Vera Kublanovskaya proposed the QR algorithm independently in 1961, with Francis in particular making several key innovations that have made the algorithm so useful in practice.

The idea behind the QR algorithm is very straightforward: Compute the QR decomposition of a matrix, multiply the $\mathbf{Q}$ and $\mathbf{R}$ factors together in reverse order to get a new matrix, and repeat. In code, we have:

```
$\mathbf{A}_0 = \mathbf{A}$
for $k = 0, 1, \ldots$
    $\mathbf{Q}_k\mathbf{R}_k := \mathbf{A}_k$          (QR factorization)
    $\mathbf{A}_{k+1} := \mathbf{R}_k\mathbf{Q}_k$      (multiplication)
end
```

Notice that the QR factorization implies that $\mathbf{R}_k = \mathbf{Q}_k^*\mathbf{A}_k$, which we can substitute into the multiplication on the next line to see that

$$\mathbf{A}_{k+1} = \mathbf{Q}_k^*\mathbf{A}_k\mathbf{Q}_k,$$

which means that $\mathbf{A}_{k+1}$ is a unitary similarity transformation of $\mathbf{A}_k$, and hence these two matrices have the same eigenvalues. Repeating this argument recursively, we see that $\mathbf{A}_k$ and $\mathbf{A}$ have the same eigenvalues. What is more important—but not at all obvious!—is that as this iteration progresses (with a few caveats): *the matrices* $\mathbf{A}_k$ *converge toward an upper triangular matrix as* $k \to \infty$. In practice, those entries in the lower triangular part of $\mathbf{A}_k$ eventually become sufficiently small that we can neglect them, and read off the eigenvalues of $\mathbf{A}$ as the diagonal entries of $\mathbf{A}_k$. Here is a simple example:

```
X = randn(4);
A = X*diag([1:4])*inv(X);
Ak = A;
while max(max(abs(tril(Ak,-1)))) > 1e-10
   [Q,R] = qr(Ak);
   Ak   = R*Q
   pause
end
```

Some iterations of the QR algorithm are shown below. Note how the entries in the lower triangular part of $\mathbf{A}_k$ shrink as k increases. When these entries are sufficiently small, one simply reads off the diagonal entries of $\mathbf{A}_k$ as estimates of the eigenvalues of $\mathbf{A}$.

```
  A  =  31.8249   -2.5381   -4.8385  -12.7864
        15.5233    0.1637   -2.2322   -7.3828
       -80.5824    3.9198   17.6708   32.5473
        99.2703   -7.1950  -17.1917  -39.6594

A_1  =  6.1064   -0.5794    1.5588  145.2803
        0.8542    2.8167    0.3044    0.5471
       -2.0517   -1.0486    1.0614   -0.0575
       -0.0287    0.0186   -0.0080    0.0155

A_2  =  5.1539   -0.9232    2.6899 -136.6413
        0.6908    2.7253    0.7795   34.0333
       -0.6148   -0.2463    1.2847  -35.8728
        0.0037   -0.0053    0.0001    0.8361

A_3  =  4.6913   -1.3339    3.1369  125.7940
        0.3871    2.6696    0.5384  -57.1354
       -0.2240   -0.0994    1.6533   45.0376
       -0.0007    0.0017    0.0008    0.9858

A_4  =  4.4363   -1.6008    3.2942 -118.3915
        0.2297    2.7117    0.2959   69.9670
       -0.0932   -0.0733    1.8404  -46.9572
        0.0001   -0.0006   -0.0007    1.0115

A_5  =  4.2887   -1.7873    3.3366  113.6042
        0.1459    2.7708    0.1299  -77.7791
       -0.0417   -0.0626    1.9293   46.4891
       -0.0000    0.0002    0.0004    1.0112

A_10 =  4.0534   -2.1784    3.2545 -104.9072
        0.0246    2.9452   -0.1404   91.7184
       -0.0011   -0.0161    2.0009  -41.2149
        0.0000   -0.0000   -0.0000    1.0006

A_20 =  4.0028   -2.2580    3.2148 -102.8199
        0.0013    2.9971   -0.1966   94.7239
       -0.0000   -0.0003    2.0001  -39.6471
        0.0000   -0.0000   -0.0000    1.0000

A_30 =  4.0002   -2.2603    3.2138 -102.7072
        0.0001    2.9998   -0.2001   94.8591
       -0.0000   -0.0000    2.0000  -39.6160
        0.0000   -0.0000   -0.0000    1.0000

A_40 =  4.0000   -2.2604    3.2138 -102.7009
        0.0000    3.0000   -0.2003   94.8661
       -0.0000   -0.0000    2.0000  -39.6155
        0.0000   -0.0000   -0.0000    1.0000
```

Practical QR algorithm. Remarkable though this algorithm may be, we can make it much faster. Each step requires a QR factorization and a matrix-matrix multiplication, both $O(n^3)$ operations. Furthermore, as we can see from the above example, typically we need many more than n steps of the QR iteration to obtain convergence to good accuracy. Together, this suggests that the QR algorithm as stated above requires at least $O(n^4)$ floating point operations: doubling the dimension of $\mathbf{A}$ makes the algorithm at least sixteen times more expensive!

Fortunately, there are several ways we can prod the QR algorithm along, reducing the overall complexity to $O(n^3)$ operations. We summarize these improvements here; for details, see the books by Trefethen and Bau, and Golub and Van Loan.

Reduction to Hessenberg Form. Although it is impossible to reduce $\mathbf{A}$ to upper triangular form with finitely many arithmetic operations, $\mathbf{A}$ can be reduced to upper Hessenberg form, i.e., a matrix that is zero everywhere below the first subdiagonal. In particular, similarity transformations involving Householder reflectors can be used to construct the factorization $\mathbf{A} = \mathbf{UHU}^*$, where $h_{j,k} = 0$ when $j > k + 1$. This is a *finite* procedure that requires $O(n^3)$ operations; this decomposition can be computed using MATLAB's `hess` command.

Since $\mathbf{A}$ and $\mathbf{H}$ have the same eigenvalues, we can simply apply the QR algorithm to $\mathbf{H}$. The upper Hessenberg structure has one key advantage: It is possible to compute a QR factorization of $\mathbf{H}$ in $O(n^2)$ operations using Givens rotations. Moreover, the $\mathbf{Q}$ matrix in the factorization is itself upper Hessenberg, so that $\mathbf{RQ}$ remains upper Hessenberg. Thus, each step of the QR algorithm, when applied to $\mathbf{H}_k$, requires only $O(n^2)$ operations. We thus have the following modified iteration.

```
$\mathbf{H}_0 = \mathbf{U}^*\mathbf{A}\mathbf{U}$                ($\mathbf{H}_0$ upper Hessenberg)
for $k = 0, 1, \ldots$
    $\mathbf{Q}_k\mathbf{R}_k := \mathbf{H}_k$          (QR factorization)
    $\mathbf{H}_{k+1} := \mathbf{R}_k\mathbf{Q}_k$      (multiplication)
end
```

Shifted QR Algorithm. There is one final modification used to accelerate convergence: the introduction of 'shifts' μ_k at each step.

```
$\mathbf{H}_0 = \mathbf{U}^*\mathbf{A}\mathbf{U}$                          ($\mathbf{H}_0$ upper Hessenberg)
for $k = 0, 1, \ldots$
    Pick a shift, $\mu_k$                      (e.g., $\mu_k = (n, n)$ entry of $\mathbf{H}_k$)
    $\mathbf{Q}_k\mathbf{R}_k := \mathbf{H}_k - \mu_k\mathbf{I}$          (QR factorization)
    $\mathbf{H}_{k+1} := \mathbf{R}_k\mathbf{Q}_k + \mu_k\mathbf{I}$      (multiplication)
end
```

The shift does not affect the upper Hessenberg structure, nor does it interfere with the similarity transformation: you can prove that $\mathbf{H}_{k+1} = \mathbf{Q}_k^*\mathbf{H}_k\mathbf{Q}_k$. However, this description does not make it clear how μ_k should be selected, or why it should have any effect on the convergence rate. Unfortunately, we do not have the time to delve into the convergence theory. It is enough to say that picking μ_k to be the (n, n) entry of $\mathbf{H}_k$ (or a related quantity) is a fine idea. For many problems, this drives the $(n, n-1)$ entry of $\mathbf{H}_k$ to zero quadratically, so in practice one only requires $O(1)$ iterations to reveal a single eigenvalue. At this point, 'deflation' techniques are applied to reduce the problem to an $(n-1)$-by-$(n-1)$ problem, and the process is continued. (Now one would choose μ_k to be the $(n-1, n-1)$ entry of $\mathbf{H}_{k-1}$, etc.) In the end, we require $O(1)$ iterations per each of the n eigenvalues, at a cost of $O(n^2)$ operations per iteration. This, combined with the preliminary upper Hessenberg reduction, gives an $O(n^3)$ algorithm for finding all n eigenvalues of $\mathbf{A}$.

(A hint about why this all works: The first column of the $\mathbf{Q}_k$ matrix behaves like the vector $\mathbf{x}_k$ in the power method, while the final column in the $\mathbf{Q}_k$ matrix behaves like the vector $\mathbf{x}_k$ in inverse iteration. The introduction of the shift μ_k accelerates convergence of the inverse iteration, as seen above. Consult Trefethen and Bau, Lectures 28 and 29, for details, or—better still—take CAAM 551!)

Lecture 38: Bracketing Algorithms for Root Finding

7. Solving Nonlinear Equations.

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.

7.1. Bracketing Algorithms.

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, $x_* \in (a_0, b_0)$.

7.1.1. Bisection.

The simplest technique for finding that root 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 method. 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, attained if x_* is at a_k or b_k, is $\frac{1}{2}(b_k - a_k) = 2^{-k-1}(b_0 - a_0)$.

Theorem. The kth bisection point c_k is no further than $(b_0 - a_0)/2^{k+1}$ from a root.

We say this iteration *converges linearly* (the log of the error is bounded by a straight line when plotted against iteration count – an example is given later in this lecture) 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.

7.1.2. 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 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 estimate 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 Step 3 differs from the bisection method. In the former case, we are forcing the bracket width $b_k - a_k$ to zero as we find our root. In the present case, there is nothing in the algorithm to drive that width to zero: We 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.

Theorem. Suppose $f \in C^2[a_0, b_0]$ for $a_0 < b_0$ with $f(a_0) < 0 < f(b_0)$ and $f''(x) \geq 0$ for all $x \in [a_0, b_0]$. Then *regula falsi* converges.

Proof. (See Stoer & Bulirsch, *Introduction to Numerical Analysis*, 2nd ed., §5.9.)

The condition that $f''(x) \geq 0$ for $x \in [a_0, b_0]$ means that f is convex on this interval, and hence $p_0(x) \geq f(x)$ for all $x \in [a_0, b_0]$. (If $p_0(x) < f(x)$ for some $x \in (a_0, b_0)$, then f has a local maximum at $\widehat{x} \in (a_0, b_0)$, implying that $f''(\widehat{x}) < 0$.) Since $p_0(c_0) = 0$, it follows that $f(c_0) \leq 0$, and so the new bracket will be $[a_1, b_1] = [c_0, b_0]$. If $f(c_0) = 0$, we have converged; otherwise, since $f''(x) \geq 0$ on $[a_1, b_1] \subset [a_0, b_0]$ and $f(a_1) = f(c_0) < 0 < f(b_0) = f(b_1)$, we can repeat this argument over again to show that $[a_2, b_2] = [c_1, b_1]$, and in general, $[a_{k+1}, b_{k+1}] = [c_k, b_k]$. Since $c_k > a_k = c_{k-1}$, we see that the points c_k are monotonically increasing, while we always have $b_k = b_{k-1} = \cdots = b_1 = b_0$. Since $c_k \leq b_k = \cdots = b_0$, the sequence $\{c_k\} = \{a_{k-1}\}$ is bounded. A fundamental result in real analysis tells us that bounded, monotone sequences must converge.† Thus, $\lim_{k\to\infty} a_k = \alpha$ with $f(\alpha) \leq 0$, and we have

$$\alpha = \frac{\alpha f(b_0) - b_0 f(\alpha)}{f(b_0) - f(\alpha)}.$$

This can be rearranged to get $(\alpha - b_0)f(\alpha) = 0$. Since $f(b_k) = f(b_0) > 0$, we must have $\alpha \neq b_0$, so it must be that $f(\alpha) = 0$. Thus, *regula falsi* converges in this setting. ■

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 will often 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!)

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.

†If this result is unfamiliar, a few minutes of reflection should convince you that it is reasonable. (Imagine a ladder with infinitely many rungs stretching from floor to ceiling in a room with finite height: eventually the rungs must get closer and closer.) For a proof, see Rudin, *Principles of Mathematical Analysis*, Theorem 3.14.

MATLAB code. A bracketing algorithm for zero-finding available in the MATLAB routine `fzero.m`. This is more sophisticated than the two algorithms described here, but the basic principle is the same. Below are simple MATLAB codes that implement bisection and *regula falsi.*

```
 function xstar = bisect(f,a,b)
% Compute a root of the function f using bisection.
% f:    a function name, e.g., bisect('sin',3,4), or bisect('myfun',0,1)
% a, b: a starting bracket:  f(a)*f(b) < 0.
 fa = feval(f,a);
 fb = feval(f,b);         % evaluate f at the bracket endpoints
 delta = (b-a);           % width of initial bracket
 k = 0; fc = inf;         % initialize loop control variables
 while (delta/(2^k)>1e-18) & abs(fc)>1e-18
    c = (a+b)/2; fc = feval(f,c);   % evaluate function at bracket midpoint
    if fa*fc < 0, b=c; fb = fc;     % update new bracket
    else a=c; fa=fc; end
    k = k+1;
    fprintf(' %3d  %20.14f  %10.7e\n', k, c, fc);
 end
 xstar = c;
```

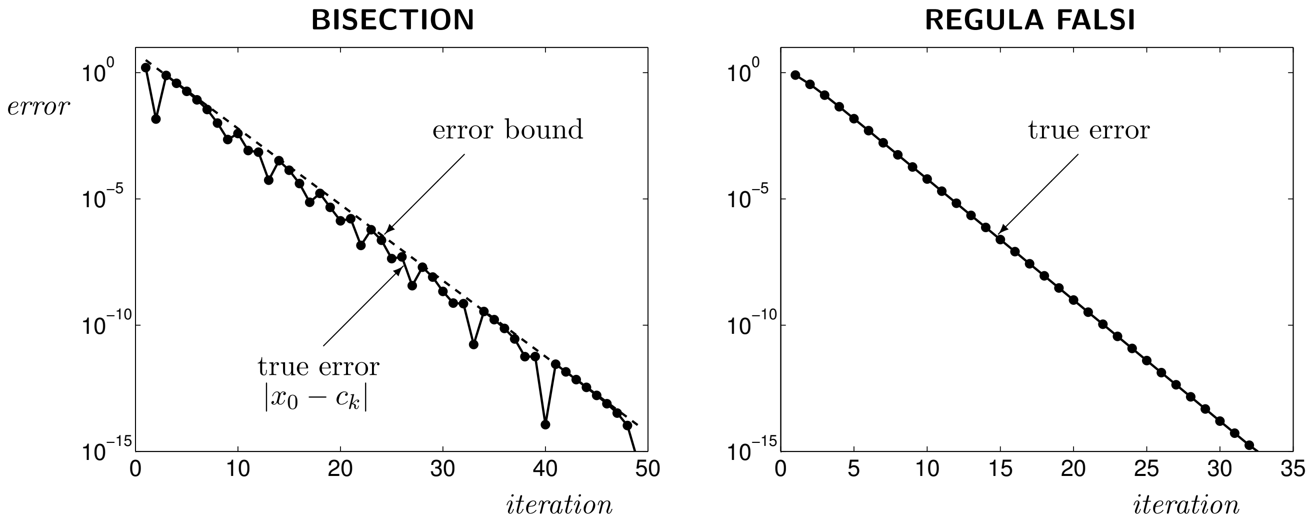
```
 function xstar = regulafalsi(f,a,b)
% Compute a root of the function f using regula falsi
% f:    a function name, e.g., regulafalsi('sin',3,4), or regulafalsi('myfun',0,1)
% a, b: a starting bracket:  f(a)*f(b) < 0.
 fa = feval(f,a);
 fb = feval(f,b);         % evaluate f at the bracket endpoints
 delta = (b-a);           % width of initial bracket
 k = 0; fc = inf;         % initialize loop control variables
 maxit = 1000;
 while (abs(fc)>1e-15) & (k < maxit)
    c = (a*fb - b*fa)/(fb-fa);    % generate new root estimate
    fc = feval(f,c);              % evaluate function at new root estimate
    if fa*fc < 0, b=c; fb = fc;   % update new bracket
    else a=c; fa=fc; end
    k = k+1;
    fprintf(' %3d  %20.14f  %10.7e\n', k, c, fc);
 end
 xstar = c;
```

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 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.*

Minimization. A closely related problem is finding a local *minimum* of f. Note that this can be accomplished by computing and analyzing the zeros of f'.[‡]

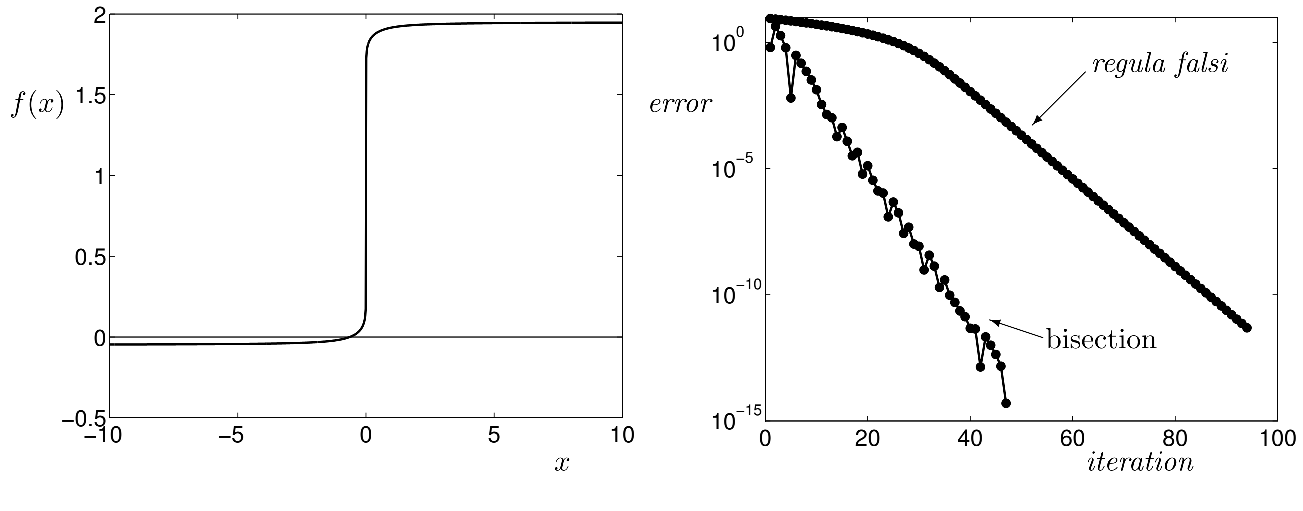
[‡]For details, see J. Stoer and R. Bulirsch, *Introduction to Numerical Analysis*, 2nd ed., Springer-Verlag, 1993, §5.9, or L. W. Johnson and R. D. Riess, *Numerical Analysis*, 2nd ed., Addison-Wesley, 1982, §4.2.

Below we show the convergence behavior of bisection and *regula falsi* when applied to solve the nonlinear equation $M = E - e\sin E$ for the unknown E, a famous problem from celestial mechanics known as *Kepler's equation*; see §7.4 in Lecture 40.



Error in root computed for Kepler's equation with $M = 4\pi/3$, $e = 0.8$ and initial bracket $[0, 2\pi]$.

Is *regula falsi* always superior to bisection? For any function for which we can construct a root bracket, one can always rig that initial bracket so the root is exactly at its midpoint, $\frac{1}{2}(a_0 + b_0)$, 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. Can one construct less contrived examples? Consider the function shown on the left below;[§] we see on the right that bisection outperforms *regula falsi*. The plot on the right shows the convergence of bisection and *regula falsi* for this example. *Regula falsi* begins much slower, then speeds up, but even this improved rate is slower than the rate of 1/2 guaranteed for bisection.



[§]This function is $f(x) = \text{sign}(\tan^{-1}(x)) * |2\tan^{-1}(x)/\pi|^{1/20} + 19/20$, whose only root is at $x \approx -0.6312881\ldots$.

Lecture 39: Root Finding via 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 it may be difficult to find an initial interval that brackets a root. Though they exhibit steady linear convergence, rather many evaluations of f may be required to attain sufficient accuracy. In this lecture, we will swap these reliable methods for a famous algorithm that often converges with amazing speed, but is more temperamental. Versions of this algorithm spring up everywhere.[†]

7.2. Newton's Method.

The idea behind the method is similar to *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, we will now require that $f \in C^2(\mathbb{R})$, that is, f and its first two derivatives should be continuous. This will allow us to expand f in a Taylor series around some approximate root x_k,

$$f(x_*) = f(x_k) + f'(x_k)(x_* - x_k) + \tfrac{1}{2}f''(\xi)(x_* - x_k)^2, \tag{39.1}$$

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,

$$0 = f(x_*) \approx f(x_k) + f'(x_k)(x_* - x_k), \quad \text{which implies} \quad x_* \approx x_k - \frac{f(x_k)}{f'(x_k)},$$

so we get an iterative method by replacing x_* in the above formulas with x_{k+1},

$$x_{k+1} := x_k - \frac{f(x_k)}{f'(x_k)}. \tag{39.2}$$

This celebrated iteration is Newton's method, implemented in the MATLAB code below.

```
 function xstar = newton(f,fprime,x0)
% Compute a root of the function f using Newton's method
% f:      a function name
% fprime: a derivative function name
% x0:     the starting guess
% Example: newton('sin','cos',3), or newton('my_f','my_fprime',1)
 maxit = 60;
 fx = feval(f,x0); x=x0; k=0;       % initialize
 fprintf(' %3d  %20.14f  %10.7e\n', k, x, fx);
 while (abs(fx) > 1e-15) & (k < maxit)
    x = x - fx/feval(fprime,x);     % Newton's method
    k = k+1;
    fx = feval(f,x);
    fprintf(' %3d  %20.14f  %10.7e\n', k, x, fx);
 end
 xstar = x;
```

[†]Richard Tapia gives a lecture titled 'If It Is Fast and Effective, It Must be Newton's Method.'

What distinguishes this iteration? 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 fractal illustrations. 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 (39.2) to obtain a recurrence for the error,

$$e_{k+1} = e_k - \frac{f(x_k)}{f'(x_k)}.$$

The Taylor expansion of $f(x_*)$ about the point x_k given in (39.1) 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 for e_{k+1} we just derived, we obtain

$$e_{k+1} = e_k - \frac{f'(x_k)e_k + \frac{1}{2}f''(\xi)e_k^2}{f'(x_k)} = -\frac{f''(\xi)e_k^2}{2f'(x_k)}.$$

Supposing that x_* is a simple root, so that $f'(x_*) \neq 0$, the above analysis suggests that when x_k is near x_*,

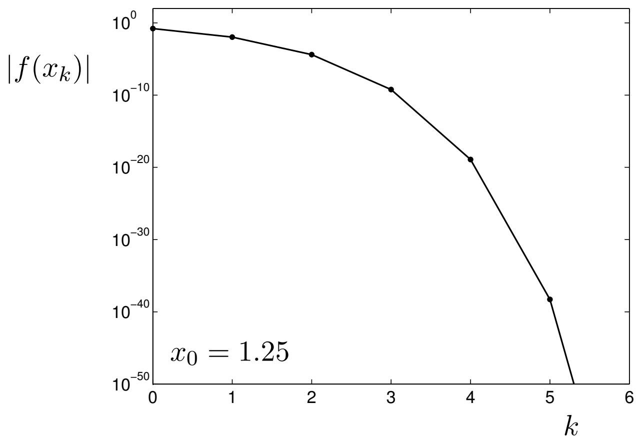
$$|e_{k+1}| \le C|e_k|^2$$

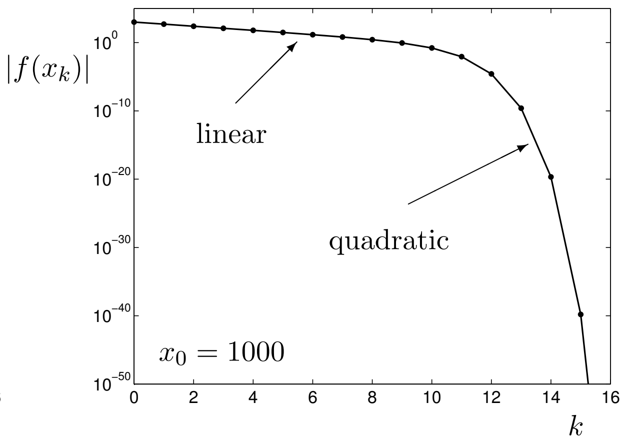
for some constant C independent of k. This is *quadratic convergence*, and it roughly means that you *double the number of correct digits* at each iteration. Compare this to bisection, where

$$|e_{k+1}| \le \tfrac{1}{2}|e_k|,$$

meaning that the error was halved at each step. Significantly, Newton's method will often exhibit a transient period of linear convergence while it gets sufficiently close to the answer, but once in a region of quadratic convergence, full machine precision is attained in just a couple more iterations.

The following example approximates the zero of $f(x) = x^2 - 2$, i.e., $x_* = \sqrt{2}$. As initial guesses, we choose $x_0 = 1.25$ (left), which gives us very rapid convergence, and $x_0 = 1000$ (right), which is a ridiculous estimate of $\sqrt{2}$, but illustrates the linear phase of convergence that can precede superlinear convergence when x_0 is far from x_*.





The table below 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 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 (underlined) at each iteration.

```
k                                              x_k
-------------------------------------------------------------------------------------------
0     1000.000000000000000000000000000000000000000000000000000000000000000000000000000000
1      500.001000000000000000000000000000000000000000000000000000000000000000000000000000
2      250.002499996000007999984000031999936000127999744000511998976002047995904008191983616 03
3      125.005249958000467994584063055265128565980148235956223934416958004774466857994638964 84
4       62.510624643017033148886913584033204645297599443257445666311646006310173914783097613 41
5       31.271309602062194555964223587717005483745658018423320865363652365782780804061538273 64
6       15.667632994868366400307555271002816520651001597103244594525815437674034799218340122 48
7        7.897642347856358067190513609342362381169683651741670251164610341607776282173649601 11
8        4.075441240519498920887985733870671333529911499613092671593339801915483080753609618 62
9        2.283092824392553839863066903581779461443392336343777816060555384816372007595553762 36
10       1.579548752406015365275470017274989351274639817763890161889757913639395862658603232 51
11       1.422866579578668250912096838563098183093109294287639281628909346738470362381849926 93
12       1.414239873591530623193646164411200351825294893478601267163957468963926900407745583 75
13       1.414213562617848512655890003591743966322076285489689082423989443916154363356253600 56
14       1.414213562373095048822868077757171182214181147294231166372548043770313324404061557 16
15       1.414213562373095048801688724209698078569830467059499948604396400794607650938583051 90
16       1.414213562373095048801688724209698078569671875376948073176679737990732478462107047 74

exact:   1.41421356237309504880168872420969807856967187537694807317667973799073247846210703885038753...
```

7.2.1. Convergence analysis.

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 will 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 our notion of the rate of convergence.

Definition. 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.*

Newton's method is second-order convergent (i.e., it converges quadratically) for $f \in C^2(\mathbb{R})$ when $f'(x_*) \neq 0$ and x_0 is sufficiently close to x_*. Bisection is linearly convergent for $f \in C[a_0, b_0]$ with rate $C = 1/2$.

Functional iteration. One can analyze Newton's method and its variants through the following general framework.[‡] Consider iterations of the form

$$x_{k+1} = \Phi(x_k),$$

for some iteration function Φ. For example, for Newton's method

$$\Phi(x) = x - \frac{f(x)}{f'(x)}.$$

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, i.e.,

$$x_* = \Phi(x_*).$$

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.

Example: Newton's method. 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_*.

What happens when $f'(x_*) = 0$? 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 $x_k \to x_*$. This general convergence framework allows us to investigate this situation more precisely. We wish to understand

$$\lim_{x\to x_*} \Phi'(x) = \lim_{x\to x_*} \frac{f(x)f''(x)}{f'(x)^2}.$$

[‡]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.

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

$$e_{k+1} = \tfrac{1}{2}e_k + O(e_k^2).$$

Note that 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 40: Root Finding via the Secant Method

Newton's method is fast if one has a good initial guess x_0. Even then, it can be inconvenient (or impossible) 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*.

7.3. Secant Method.

Throughout this semester, we saw 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.) 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 last 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 $f(x)$, but the secant method require 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: a poor initial guess can lead to divergence!

Do we recover the convergence behavior of Newton's method? Not quite, but the secant method (under suitable hypotheses) is *superlinear*, i.e., it is pth-order convergent with $p > 1$. In particular, it converges with order equal to the golden ratio, $\phi = \frac{1}{2}(1+\sqrt{5}) \approx 1.6180$:

$$|e_{k+1}| \le C|e_k|^{\phi},$$

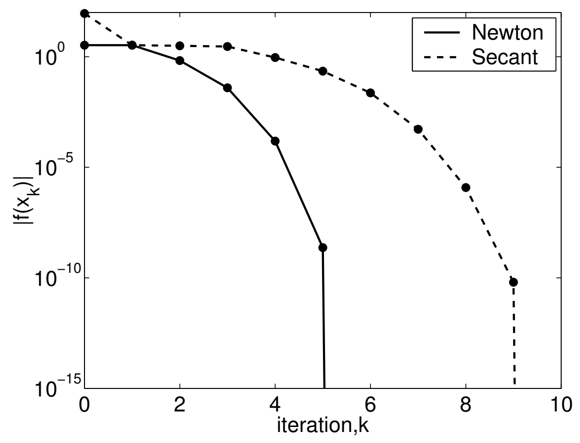
for a constant $C > 0$. Though you may regret that the secant method does not recover quadratic convergence, take solace in the fact that only one function evaluation $f(x_k)$ is required at each iteration, as opposed to Newton's method, which requires $f(x_k)$ and $f'(x_k)$. Typically the derivative is more expensive to compute than the function itself. If we assume that evaluating $f(x_k)$ and $f'(x_k)$ require the same amount of effort, then we can compute two secant iterates for roughly the same cost as a single Newton iterate. Two steps of the secant method combine to give an improved convergence rate:

$$|e_{k+2}| \;\le\; C|e_{k+1}|^{\phi} \;\le\; C\Big|C|e_k|^{\phi}\Big|^{\phi} \;\le\; C^{1+\phi}|e_k|^{\phi^2},$$

where $\phi^2 = \frac{1}{2}(3+\sqrt{5}) \approx 2.62 > 2$. Hence, in terms of computing time, the secant method can actually be more efficient than Newton's method.[†]

The following plot shows the convergence of Newton's method on $f(x) = 1/x - 10$ with $x_0 = .15$, and the secant method with $x_0 = .01$ and $x_1 = .15$.

[†]This discussion is drawn from Kincaid and Cheney, *Numerical Analysis*, 3rd ed., §3.3.



```
      k      x_k (Newton)   |f(x_k)| (Newton)    x_k (secant)    |f(x_k)| (secant)
-------------------------------------------------------------------------------------
      0    0.15000000000000  -3.3333333e+00   0.01000000000000   9.0000000e+01
      1    0.07500000000000   3.3333333e+00   0.15000000000000  -3.3333333e+00
      2    0.09375000000000   6.6666667e-01   0.14500000000000  -3.1034483e+00
      3    0.09960937500000   3.9215686e-02   0.07750000000000   2.9032258e+00
      4    0.09999847412109   1.5259022e-04   0.11012500000000  -9.1940976e-01
      5    0.09999999997672   2.3283064e-09   0.10227812500000  -2.2273824e-01
      6    0.10000000000000   0.0000000e+00   0.09976933984375   2.3119343e-02
      7                                       0.10000525472668  -5.2544506e-04
      8                                       0.10000001212056  -1.2120559e-06
      9                                       0.09999999999936   6.3689498e-11
     10                                       0.10000000000000   0.0000000e+00
```

```
 function xstar = secant(f, x0, x1)
% function xstar = secant(f, x0, x1)
% Compute a root of the function f using the secant method
% f:      a function name
% x0:     the starting guess
% x1:     the second starting point (defaults to x0+1)
% Example: secant('sin',3), or secant('my_f',1,1.1)


 if (nargin < 3), x1 = x0+1; end
 maxit = 60;
 xprev = x0; fxprev = feval(f,xprev);
 xcur  = x1; fxcur  = feval(f,xcur);  k=1;       % initialize
 fprintf(' %3d  %20.14f  %10.7e\n', 0, xprev, fxprev);
 fprintf(' %3d  %20.14f  %10.7e\n', 1, xcur, fxcur);
 while (abs(fxcur) > 1e-15) & (k < maxit)
    x = xcur - fxcur*(xcur-xprev)/(fxcur-fxprev);     % Secant method
    xprev = xcur; fxprev = fxcur;
    xcur  = x;    fxcur  = feval(f,xcur);
    k = k+1;
    fprintf(' %3d  %20.14f  %10.7e\n', k, xcur, fxcur);
 end
 xstar = x;
```

7.4. A Parting Example: Kepler's Equation.

We close by describing the most famous nonlinear equation, developed in the first two decades of the seventeenth century by Johannes Kepler to solve the two-body problem in celestial mechanics. Kepler determined that a satellite trajectory forms an ellipse with its primary body at a focus point. Let $e \in [0, 1)$ denote the eccentricity of the ellipse and a denote the semi-major axis length. Assume that the satellite makes its closest approach to the primary, called periapsis, at time $t = 0$. The critical question is: Where is the satellite at time $t > 0$? We could numerically integrate the differential equations of motion (e.g., using the Störmer–Verlet algorithm), but in this simple setting there is a more direct approach that avoids differential equations. We measure the satellite's location by the angle ν swept out by the satellite's position vector from $\nu = 0$ at periapsis ($t = 0$) through to $\nu = 2\pi$ when the satellite returns to periapsis, having completed one orbit. If τ denotes the length of time of this one orbit (the *period*), then $M(t) = M := 2\pi t/\tau \in [0, 2\pi)$ describes the proportion of the period elapsed since the satellite last passed periapsis. Kepler solved the two-body problem by finding a formula for ν at a given time, t, $\tan(\nu/2) = \sqrt{(1 + e)/(1 - e)}\tan(E/2)$, where $E \in [0, 2\pi)$ is the *eccentric anomaly*, the solution of

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

This nonlinear equation for the unknown E is Kepler's Equation, an innocuous formula that has received tremendous study.[‡] Despite this scrutiny, it turns out that Kepler's Equation is perfectly suited to the algorithms we study here, and is routinely solved in a few milliseconds. To determine the value of E, simply find the zero of $f(E) = M - E + e\sin E$ for $E \in [0, 2\pi)$.

...

Kepler's equation is but one important and beautiful example of numerical analysis at its most effective: an ideal algorithm applied to a well-conditioned problem gives amazing accuracy in an instant.

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.

[‡]See Peter Colwell, *Solving Kepler's Equation Over Three Centuries*, Willmann-Bell, 1993.