# Numerical Analysis

Lecture Notes, Fall 2024, Spring 2025

Yifan Chen
NYU Courant Institute of Mathematical Sciences

2025

**Abstract**

These notes accompany the lectures of MATH–UA 252 (Numerical Analysis) at NYU Courant, Fall 2024 and Spring 2025. The course covers: error analysis and one–dimensional root finding; numerical linear algebra (direct and iterative solvers, norms and conditioning, least squares and QR, eigenvalue and singular value problems); approximation of functions (polynomial interpolation, orthogonal polynomials, numerical differentiation and integration, the discrete Fourier transform); and initial–value problems for ordinary differential equations. The primary reference is U. Ascher and C. Greif, *A First Course in Numerical Methods*.

## Contents

# Part I
# Foundations

## 1 What is numerical analysis?

Following Ascher and Greif, numerical analysis is the discipline concerned with the design and study of algorithms for solving mathematical problems arising in science and engineering. The problems we are interested in are typically posed in continuous mathematics — real numbers, derivatives, integrals, functions — while the computer handles only finitely many bits and finitely many operations. The role of numerical analysis is to design discrete algorithms that approximate the continuous problem, and to quantify how accurate and how stable those algorithms are.

The course is organized around three classes of problems:

- **Linear algebra problems:** solving $Ax = b$, least squares, eigenvalue and singular value problems.
- **Function approximation:** polynomial interpolation, $L^2$ approximation by orthogonal polynomials, numerical differentiation, quadrature, and the Fourier transform.
- **Differential equations:** initial–value problems for ordinary differential equations.

For each problem we will ask two questions: how accurate is the computed solution, and how sensitive is the result to small perturbations of the input. These two questions lead to the notions of *error analysis* on the one hand and *conditioning and stability* on the other, and they will recur throughout the course.

## 2 Sources of error

A scientific computation can carry three different kinds of error.

**(1) Modelling error.** The real–world problem is first translated into a mathematical model, and the model itself is only an approximation. This error is independent of the algorithm used to solve the model.

**(2) Approximation (truncation) error.** The mathematical model is then replaced by something a computer can handle: infinite series are truncated, derivatives become finite differences, integrals become quadrature sums, continuous time becomes a grid. The resulting error decreases as we use more terms, finer grids, or smaller step sizes.

**(3) Roundoff error.** A computer stores real numbers in floating–point format with finitely many bits. In IEEE 754 double precision, the unit roundoff (machine epsilon) is

$$\varepsilon_{\text{mach}} \;=\; 2^{-52} \;\approx\; 2.22 \times 10^{-16},$$

and every elementary arithmetic operation introduces a relative error of size $O(\varepsilon_{\text{mach}})$. Unlike truncation error, roundoff error tends to accumulate as we do more work.

## 2.1 Absolute and relative error

If $\hat{x}$ approximates $x$, the *absolute error* is $|x - \hat{x}|$ and, for $x \neq 0$, the *relative error* is $|x - \hat{x}| \,/\, |x|$. The relative error is dimensionless and measures how many leading digits of $\hat{x}$ agree with $x$.

## 2.2 The truncation–vs.–roundoff trade–off

Truncation error and roundoff error often pull in opposite directions, and a good illustration is the one–sided difference quotient

$$D_h f(x) \;:=\; \frac{f(x+h) - f(x)}{h} \;\approx\; f'(x). \tag{1}$$

By Taylor's theorem $D_h f(x) - f'(x) = \frac{1}{2} h\, f''(\xi)$, so the truncation error is at most $\frac{1}{2} M h$ with $M := \max |f''|$, which favours small $h$. At the same time, the values $f(x+h)$ and $f(x)$ are stored with relative error $\sim \varepsilon_{\text{mach}}$, and the computed quotient inherits a roundoff contribution of size $\sim 2\varepsilon_{\text{mach}} \,|f(x)| \,/h$, which favours large $h$. Adding the two contributions,

$$\text{total error}(h) \;\lesssim\; \tfrac{1}{2} M\, h \;+\; \frac{2\varepsilon_{\text{mach}} \,|f(x)|}{h},$$

and minimizing in $h$ gives

$$h_{\text{opt}} \;\approx\; 2\sqrt{\varepsilon_{\text{mach}} \,|f(x)| \,/M},$$

with a minimum error of order $\sqrt{\varepsilon_{\text{mach}}} \sim 10^{-8}$. So even for the simplest derivative formula we lose roughly half of the available digits, and making $h$ smaller eventually makes things worse.

### 2.3 Operations to handle with care

**Cancellation.** If $x$ and $y$ agree in their first $k$ leading digits, then $x - y$ loses those $k$ digits of relative accuracy. When possible, rearrange the formula to avoid the subtraction. For example,

$$\sqrt{x+1} - \sqrt{x} \;=\; \frac{1}{\sqrt{x+1} + \sqrt{x}},$$

and the right–hand side is well behaved for large $x$.

**Division by small numbers.** An error of size $\varepsilon$ in the numerator becomes an error of size $\varepsilon/h$ in the quotient.

**Adding numbers of very different magnitudes.** In double precision, $1 + 10^{-20} = 1$. Summing many small numbers into a large running total can lose them entirely.

**An unstable recurrence.** The recurrence $I_n = 1 - n\, I_{n-1}$, starting from $I_0 = 1 - 1/e$, computes $I_n = \int_0^1 x^n\, e^{x-1}\, dx$. Mathematically it is exact, but each step multiplies the error by $n$, so a roundoff of $10^{-16}$ in $I_0$ grows like $n! \cdot 10^{-16}$ and the result is useless for moderate $n$. This shows that the algebraic form of an otherwise correct algorithm can determine whether it is usable.

# Part II
# One-dimensional root finding

## 3 The problem and the players

We want to solve

$$f(x) \;=\; 0, \qquad f \in C[a,b], \tag{2}$$

where $f$ is given and a root $x^*$ exists in $[a,b]$. In practice we never compute $x^*$ in finitely many steps. Instead, we generate an iterative sequence $\{x_k\}$ and stop when one of the following criteria is met:

- $|x_k - x_{k-1}| < \varepsilon_{\text{abs}}$ (absolute change small);
- $|x_k - x_{k-1}| \,/\, |x_k| < \varepsilon_{\text{rel}}$ (relative change small);
- $|f(x_k)| < \varepsilon_{\text{fcn}}$ (residual small);
- a maximum iteration count is reached (safety net).

**Convergence rates.** Let $e_k := x_k - x^*$. We say the iteration converges with *order* $p$ if there is a constant $C$ such that

$$|e_{k+1}| \;\le\; C\,|e_k|^p\,.$$

The case $p = 1$ with $C < 1$ is *linear* convergence (error multiplies by a constant factor each step), $p = 2$ is *quadratic* (the number of correct digits doubles), and any $1 < p < 2$ is *superlinear*.

### 3.1 Bisection

If $f \in C[a,b]$ and $f(a)f(b) < 0$, the intermediate value theorem guarantees a root in $(a,b)$. Test the midpoint $m = (a+b)/2$; whichever half retains the sign change becomes the new bracket.

**Algorithm 3.1** (Bisection)**.** Given $f \in C[a,b]$ with $f(a)f(b) < 0$ and tolerance $\varepsilon$:

1. Set $m \leftarrow (a+b)/2$.
2. If $|b-a| < \varepsilon$, return $m$.
3. If $f(a)f(m) < 0$, set $b \leftarrow m$; else set $a \leftarrow m$.
4. Goto 1.

After $n$ iterations the bracket has length $(b-a)/2^n$, so

$$|x_n - x^*| \;\le\; \frac{b-a}{2^{n+1}}.$$

Bisection is linearly convergent with rate $\rho = 1/2$, gaining one bit of accuracy per iteration. It is guaranteed to converge under the sole assumption that $f$ is continuous and $f(a)f(b) < 0$.

### 3.2 Fixed–point iteration

Rewrite $f(x) = 0$ as $x = g(x)$ for some auxiliary $g$, and iterate

$$x_{k+1} \;=\; g(x_k). \tag{3}$$

Any limit of the sequence is automatically a fixed point of $g$ and therefore a root of $f$.

**Theorem 3.2** (Contraction mapping / fixed–point theorem)**.** *Let $g \in C^1[a,b]$ with*

(i) $g\big([a,b]\big) \subseteq [a,b]$, *and*

(ii) $|g'(x)| \le \rho < 1$ *for all* $x \in [a,b]$.

*Then $g$ has a* unique *fixed point $x^* \in [a,b]$, and the iterates* (3) *converge to it from any starting point $x_0 \in [a,b]$, with*

$$|x_k - x^*| \;\le\; \rho^k\,|x_0 - x^*|\,.$$

*Sketch.* Existence: $h(x) := g(x) - x$ satisfies $h(a) \geq 0$ and $h(b) \leq 0$, so the IVT gives a zero. Uniqueness: if $g(x_1) = x_1$ and $g(x_2) = x_2$, then by the mean value theorem $|x_1 - x_2| = |g(x_1) - g(x_2)| \leq \rho\,|x_1 - x_2|$, forcing $x_1 = x_2$. Convergence: similarly, $|e_{k+1}| = |g(x_k) - g(x^*)| \leq \rho\,|e_k|$, and iterating gives $\rho^k\,|e_0| \to 0$. □

Convergence is linear with rate $\rho = |g'(x^*)|$. When $g'(x^*) = 0$, convergence becomes superlinear; this is the case exploited by Newton's method.

**Example 3.3.** Take $f(x) = \sin(2x) - x$, equivalently $g(x) = \sin(2x)$. Near the root $x^* = 0$, $|g'(x)| = 2\,|\cos(2x)|$ is not $< 1$, and the iteration fails to converge. Rewriting the equation in a different fixed–point form may yield a $g$ with smaller derivative at the root, and then the iteration converges.

### 3.3 Newton's method

Suppose $f$ is smooth and $f(x^*) = 0$. Linearize around the current iterate $x_k$:

$$0 \;=\; f(x^*) \;\approx\; f(x_k) + f'(x_k)\big(x^* - x_k\big).$$

Solving for $x^*$ and renaming the result $x_{k+1}$ gives *Newton's method*:

$$x_{k+1} \;=\; x_k \;-\; \frac{f(x_k)}{f'(x_k)}. \tag{4}$$

Geometrically, $x_{k+1}$ is the $x$–intercept of the tangent line to $y = f(x)$ at $x = x_k$.

**Theorem 3.4** (Quadratic convergence of Newton)**.** *Suppose $f \in C^2$ in a neighbourhood of a simple root $x^*$, i.e. $f(x^*) = 0$ and $f'(x^*) \neq 0$. Then for $x_0$ sufficiently close to $x^*$, the Newton iterates (4) converge to $x^*$ and satisfy*

$$|e_{k+1}| \;\leq\; \frac{M}{2\,m}\;|e_k|^2\,,$$

*where $m = \min|f'|$ and $M = \max|f''|$ in the neighbourhood.*

*Sketch.* By Taylor, $0 = f(x^*) = f(x_k) + f'(x_k)(x^* - x_k) + \frac{1}{2}f''(\xi_k)(x^* - x_k)^2$. Divide by $f'(x_k)$ and rearrange: $x_{k+1} - x^* = -\frac{f''(\xi_k)}{2f'(x_k)}(x^* - x_k)^2$, which gives the claimed bound. □

Each iteration of Newton's method costs one $f$ evaluation, one $f'$ evaluation, and one division. Its limitations are: (i) it requires $f'$; (ii) convergence is only local, so a poor initial guess may diverge or oscillate; (iii) it slows to linear convergence at multiple roots ($f'(x^*) = 0$).

**Example 3.5.** For $f(x) = e^x - 1$ (root $x^* = 0$), $x_{k+1} = x_k - (e^{x_k} - 1)/e^{x_k} = x_k - 1 + e^{-x_k}$. Starting from $x_0 = 1$ gives $x_1 \approx 0.3679$, $x_2 \approx 0.0601$, $x_3 \approx 0.00177$, $x_4 \approx 1.6 \times 10^{-6}$, with the number of correct digits roughly doubling each step.

### 3.4 The secant method

Newton's method requires $f'$, which may be expensive or unavailable. Replace it by a finite difference computed from the two most recent iterates:

$$f'(x_k) \;\approx\; \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}},$$

giving the *secant method*

$$x_{k+1} \;=\; x_k \;-\; f(x_k)\,\frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})}. \tag{5}$$

Geometrically, $x_{k+1}$ is the $x$–intercept of the secant line through $(x_{k-1}, f(x_{k-1}))$ and $(x_k, f(x_k))$.

The secant method requires no derivative, uses one new function evaluation per step, and converges with order $p = (1+\sqrt{5})/2 \approx 1.618$, that is, superlinearly but slower than Newton.

**Summary table.**

| Method | Order $p$ | Needs | Comment |
|---|---|---|---|
| Bisection | 1 ($\rho = 1/2$) | $f$, sign change | always converges, slow |
| Fixed point | 1 ($\rho < 1$) | $g$ with contraction | rewrite–dependent |
| Newton | 2 | $f, f'$ | local; very fast |
| Secant | $\approx 1.618$ | $f$ only | two starting values |

# Part III
# Numerical linear algebra

## 4 Direct solvers for $Ax = b$

Given $A \in \mathbb{R}^{n\times n}$ nonsingular and $b \in \mathbb{R}^n$, we want $x$ such that $Ax = b$. Although mathematically $x = A^{-1}b$, in practice one does not form the inverse: solving the system directly is cheaper, more stable, and preserves sparsity.

### 4.1 Triangular systems and back substitution

Triangular systems are easy. If $U$ is upper triangular,

$$\begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & u_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix},$$

the last equation immediately gives $x_n = b_n/u_{nn}$, the second–to–last gives $x_{n-1}$ in terms of $x_n$, and so on:

$$x_k \;=\; \frac{1}{u_{kk}}\Bigg(b_k - \sum_{j=k+1}^{n} u_{kj}\, x_j\Bigg), \qquad k = n, n-1, \dots, 1. \tag{6}$$

This is *back substitution*, and it costs about $n^2$ flops. Forward substitution for lower triangular systems is analogous.

### 4.2 Gaussian elimination and the LU factorization

Gaussian elimination is the systematic way to reduce a general system to a triangular one. At step $k$ we use the $k$th row, scaled by the multiplier $\ell_{ik} = a_{ik}^{(k-1)}/a_{kk}^{(k-1)}$, to zero out every entry below the diagonal in column $k$.

Each elimination step is the same as multiplying $A$ on the left by a unit lower triangular matrix

$$M^{(k)} \;=\; I - \ell^{(k)}\, {e_k}^{\top}, \qquad \ell^{(k)} \;=\; \big(0, \dots, 0,\, \ell_{k+1,k}, \dots, \ell_{nk}\big)^{\top}.$$

After $n-1$ such steps, $M^{(n-1)} \cdots M^{(1)} A = U$ is upper triangular. Inverting and using the fact that the inverse of $M^{(k)}$ is $I + \ell^{(k)} {e_k}^{\top}$, we obtain the *LU factorization*

$$A \;=\; LU, \tag{7}$$

where $L$ is unit lower triangular (with the multipliers $\ell_{ij}$ on its strict lower triangle) and $U$ is upper triangular.

Once $A = LU$ is available, the system $Ax = b$ becomes $L(Ux) = b$, and we solve it in two triangular sweeps:

$$Ly = b \quad \text{(forward)}, \qquad Ux = y \quad \text{(backward)}.$$

The factorization step costs about $\tfrac{2}{3}n^3$ flops, while each triangular sweep costs $O(n^2)$. Once $L$ and $U$ are stored, additional right–hand sides cost only $O(n^2)$.

### 4.3 When does $LU$ exist? Pivoting

The factorization above breaks down whenever a pivot $a_{kk}^{(k-1)}$ is zero. Even when it is nonzero, a small pivot produces a large multiplier and amplifies roundoff error.

**Theorem 4.1** (Existence of LU)**.** *Let $A \in \mathbb{R}^{n\times n}$. If every leading principal submatrix $A_{1:k,1:k}$ ($k = 1, \dots, n-1$) is nonsingular, then $A$ admits a factorization $A = LU$ with $L$ unit lower triangular and $U$ upper triangular. If in addition $A$ itself is nonsingular, the factorization is unique.*

**Example 4.2.** The matrix $A = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ has singular leading $1 \times 1$ submatrix $[0]$, and indeed admits no $LU$ factorization without row swaps. But after swapping rows, $\widetilde{A} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ is its own $LU$.

**Numerical instability without pivoting.** Consider

$$A = \begin{pmatrix} \varepsilon & 1 \\ 1 & 1 \end{pmatrix}, \qquad b = \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \quad 0 < \varepsilon \ll 1.$$

The exact solution is $x_1 \approx 1$, $x_2 \approx 1$. Without pivoting, $\ell_{21} = 1/\varepsilon$ is enormous, and after elimination

$$U = \begin{pmatrix} \varepsilon & 1 \\ 0 & 1 - 1/\varepsilon \end{pmatrix}.$$

In floating point, $1 - 1/\varepsilon$ rounds to $-1/\varepsilon$, so back substitution returns $x_2 \approx 1$ but $x_1 \approx 0$. Swapping the two rows before eliminating gives $\ell_{21} = \varepsilon$ and the same arithmetic recovers the correct solution.

**Partial pivoting.** At each step $k$, choose the row index $q = \arg\max_{k \le i \le n} \left| a_{ik}^{(k-1)} \right|$ and swap rows $k$ and $q$ before eliminating. This guarantees $|\ell_{ik}| \le 1$ and so bounds the growth of multipliers. Recording all the swaps in a permutation matrix $P$ gives the canonical statement

$$PA = LU. \tag{8}$$

The factorization step still costs $\frac{2}{3}n^3$ flops, plus $O(n^2)$ comparisons, and Theorem 4.1 now applies to *every* nonsingular $A$.

**Complete pivoting.** At each step one may also swap columns and bring the largest entry of the remaining submatrix to the pivot position. This gives better worst–case stability at the cost of $O(n^3)$ extra comparisons; partial pivoting is the standard default.

### 4.4 Cholesky factorization

A matrix $A \in \mathbb{R}^{n \times n}$ is *symmetric positive definite* (SPD) if $A = A^\top$ and $x^\top A x > 0$ for all $x \neq 0$. If $A$ is SPD, every leading principal submatrix is also SPD, so by Theorem 4.1 $A$ admits an $LU$ factorization without pivoting. Symmetry then forces $U = DL^\top$ with $D > 0$, and absorbing $\sqrt{D}$ into the triangular factor yields the *Cholesky factorization*

$$A = G G^\top, \tag{9}$$

where $G$ is lower triangular with positive diagonal. Cholesky costs $\frac{1}{3}n^3$ flops, half of $LU$, and requires no pivoting.

## 5 Vector and matrix norms

To talk about the size of an error, we need the size of a vector; to talk about the sensitivity of a linear system, we need the size of a matrix. Both notions are formalized by *norms*.

**Definition 5.1.** A function $\|\cdot\| : \mathbb{R}^n \to \mathbb{R}$ is a (vector) norm if

(i) $\|x\| \geq 0$, with equality iff $x = 0$;

(ii) $\|\alpha x\| = |\alpha| \, \|x\|$ for all $\alpha \in \mathbb{R}$;

(iii) $\|x + y\| \leq \|x\| + \|y\|$ (triangle inequality).

**The $p$–norms.** For $1 \leq p < \infty$, $\|x\|_p = \left(\sum_{i=1}^n |x_i|^p\right)^{1/p}$, and $\|x\|_\infty = \max_i |x_i|$. The triangle inequality for these is Minkowski's inequality. The three workhorse choices are

$$\|x\|_1 = \sum |x_i| \,, \qquad \|x\|_2 = \sqrt{\sum x_i^2}, \qquad \|x\|_\infty = \max_i |x_i| \,.$$

In $\mathbb{R}^n$, all norms are equivalent: for any two norms $\mu, \nu$ there exist constants $0 < \alpha \leq \beta$ with $\alpha \, \nu(x) \leq \mu(x) \leq \beta \, \nu(x)$. For example, $\|x\|_\infty \leq \|x\|_2 \leq \sqrt{n} \, \|x\|_\infty$. Convergence in one norm therefore implies convergence in any other, though the rates may differ by dimension–dependent constants.

**Matrix norms.** The *Frobenius norm* $\|A\|_F = \left(\sum_{i,j} a_{ij}^2\right)^{1/2}$ treats $A$ as a long vector. For analyzing linear systems, the *induced* (or operator) norms

$$\|A\| \;=\; \sup_{x \neq 0} \frac{\|Ax\|}{\|x\|} \;=\; \sup_{\|x\|=1} \|Ax\| \tag{10}$$

are more useful, where the norm on the right is the chosen vector norm. Induced norms satisfy $\|Ax\| \leq \|A\| \, \|x\|$ by definition and the submultiplicative property $\|AB\| \leq \|A\| \, \|B\|$.

**Theorem 5.2.** *For an $m \times n$ matrix $A$,*

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

$$\|A\|_\infty \;=\; \max_{1 \leq i \leq m} \sum_{j=1}^n |a_{ij}| \quad (\textit{maximum row sum}),$$

$$\|A\|_2 \;=\; \sqrt{\rho(A^\top A)} \;=\; \sigma_{\max}(A),$$

*where $\rho(\cdot)$ is the spectral radius (largest absolute eigenvalue) and $\sigma_{\max}$ is the largest singular value.*

The first two formulas follow by bounding $\|Ax\|_p$ above and exhibiting a vector $x$ that attains the bound. The third follows from $\|Ax\|_2^2 = x^\top A^\top A x$, which is maximized at the largest eigenvalue of $A^\top A$.

# 6 Conditioning of linear systems

We now ask how the solution of $Ax = b$ changes under perturbations $A \to A + \Delta A$, $b \to b + \Delta b$, with $A$ nonsingular.

**Definition 6.1.** The *condition number* of $A$ (with respect to a chosen norm) is

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

For induced norms, $\kappa(A) \geq 1$, with equality for orthogonal matrices in the 2–norm. In the 2–norm, $\kappa_2(A) = \sigma_{\max}(A)/\sigma_{\min}(A)$, and for SPD matrices this becomes $\lambda_{\max}/\lambda_{\min}$. A matrix is well–conditioned if $\kappa(A)$ is moderate and ill–conditioned if $\kappa(A) \gg 1$.

**Theorem 6.2** (Sensitivity)**.** *Let $A$ be nonsingular, $Ax = b$ with $b \neq 0$, and let $x + \Delta x$ solve $A(x + \Delta x) = b + \Delta b$. Then*

$$\frac{\|\Delta x\|}{\|x\|} \;\leq\; \kappa(A)\, \frac{\|\Delta b\|}{\|b\|}.$$

*More generally, if $(A + \Delta A)(x + \Delta x) = b + \Delta b$ and $\|\Delta A\| \, \left\|A^{-1}\right\| < 1$,*

$$\frac{\|\Delta x\|}{\|x\|} \;\leq\; \frac{\kappa(A)}{1 - \kappa(A)\, \|\Delta A\| \,/\, \|A\|} \left( \frac{\|\Delta b\|}{\|b\|} + \frac{\|\Delta A\|}{\|A\|} \right).$$

**Conditioning vs. stability.** Conditioning and stability are distinct concepts:

- **Conditioning** is a property of the *problem*: it quantifies how much the exact answer changes under perturbations of the data.
- **Numerical stability** is a property of the *algorithm*. An algorithm is *backward stable* if the computed answer is the exact answer to a slightly perturbed problem. Pivoting in Gaussian elimination is a device for achieving backward stability.

For a backward stable algorithm the forward error $\|\widehat{x} - x\| \,/\, \|x\|$ is bounded by $\kappa(A)$ times the unit roundoff.

# 7 Iterative solvers

Direct solvers cost $O(n^3)$ flops and tend to fill in sparse matrices. For large sparse systems an alternative is to construct a sequence $\{x^{(k)}\}$ converging to the true solution $x$ and stop once the residual is small enough. The hope is that we reach an acceptable accuracy in far fewer than $n$ steps.

### 7.1 Splittings and the basic recurrence

Write $A = M - N$ with $M$ "easy" to invert. Then $Ax = b$ becomes $Mx = Nx + b$ and suggests the iteration

$$M\,x^{(k+1)} \;=\; N\,x^{(k)} + b, \qquad x^{(k+1)} = M^{-1}N\,x^{(k)} + M^{-1}b. \tag{11}$$

With $T := M^{-1}N$ and $c := M^{-1}b$, this is $x^{(k+1)} = Tx^{(k)} + c$. Subtracting the fixed–point relation $x = Tx + c$ gives the error recurrence $e^{(k+1)} = T\,e^{(k)}$, so $\|e^{(k)}\| \le \|T\|^k \,\|e^{(0)}\|$ and the iteration converges as soon as $\|T\| < 1$ in some induced norm.

**Theorem 7.1.** *The iteration* (11) *converges from every starting vector if and only if* $\rho(T) < 1$*, where* $\rho(T)$ *is the spectral radius of* $T$*. Moreover* $\rho(T)$ *is the asymptotic rate of error decay.*

### 7.2 Jacobi, Gauss–Seidel, and SOR

Write $A = D + L_A + U_A$, where $D$ is the diagonal of $A$ and $L_A, U_A$ are its strict lower and upper triangles.

**Jacobi.** Take $M = D$, $N = -(L_A + U_A)$. Componentwise,

$$x_i^{(k+1)} \;=\; \frac{1}{a_{ii}}\Big(b_i - \sum_{j \neq i} a_{ij}\,x_j^{(k)}\Big),$$

with iteration matrix $T_J = -D^{-1}(L_A + U_A)$.

**Gauss–Seidel.** Take $M = D + L_A$. Componentwise,

$$x_i^{(k+1)} \;=\; \frac{1}{a_{ii}}\Big(b_i - \sum_{j<i} a_{ij}\,x_j^{(k+1)} - \sum_{j>i} a_{ij}\,x_j^{(k)}\Big),$$

with iteration matrix $T_{GS} = -(D + L_A)^{-1}U_A$. Each new component uses the most recently updated values.

**Successive Over–Relaxation (SOR).** SOR introduces a relaxation parameter $\omega$:

$$x_i^{(k+1)} \;=\; (1-\omega)\,x_i^{(k)} \;+\; \frac{\omega}{a_{ii}}\Big(b_i - \sum_{j<i} a_{ij}\,x_j^{(k+1)} - \sum_{j>i} a_{ij}\,x_j^{(k)}\Big).$$

$\omega = 1$ recovers Gauss–Seidel; $1 < \omega < 2$ can accelerate convergence for suitable problems.

# 8 Least squares and QR

We now allow $A \in \mathbb{R}^{m\times n}$ with $m > n$ (more equations than unknowns) and full column rank. The system $Ax = b$ generally has no solution. Instead we solve the *linear least squares* problem

$$\min_{x\in\mathbb{R}^n} \ \|Ax - b\|_2^2 . \tag{12}$$

Geometrically we are looking for the orthogonal projection of $b$ onto the column space of $A$.

## 8.1 Normal equations

Setting the gradient of the objective to zero gives the *normal equations*

$$A^\top A\, x \ = \ A^\top b. \tag{13}$$

If $A$ has full column rank then $A^\top A$ is SPD and invertible, so

$$x \ = \ (A^\top A)^{-1} A^\top b \ =: \ A^\dagger b,$$

where $A^\dagger$ is the (Moore–Penrose) *pseudo–inverse*. The system can be solved by:

1. form $B = A^\top A$ and $c = A^\top b$ $\quad (O(mn^2))$;
2. compute the Cholesky factorization $B = GG^\top$ $\quad (O(n^3))$;
3. solve $Gz = c$, then $G^\top x = z$ by triangular sweeps.

**Squaring of the condition number.** This approach has a numerical drawback: $\kappa_2(A^\top A) = \kappa_2(A)^2$. A mildly ill–conditioned $A$ produces a severely ill–conditioned $A^\top A$, and useful information may be lost when forming $A^\top A$ in finite precision. The remedy is to work directly with $A$ via an orthogonal factorization.

## 8.2 The QR factorization

**Theorem 8.1.** *Let $A \in \mathbb{R}^{m\times n}$ with $m \geq n$ and full column rank. Then there exist $Q \in \mathbb{R}^{m\times m}$ orthogonal ($Q^\top Q = I_m$) and $R \in \mathbb{R}^{n\times n}$ upper triangular and nonsingular such that*

$$A \ = \ Q \begin{pmatrix} R \\ 0 \end{pmatrix}.$$

*Equivalently, the first $n$ columns of $Q$ form an orthonormal basis of* $\mathrm{Col}(A)$*, and $A = Q_1 R$ with $Q_1 \in \mathbb{R}^{m\times n}$, ${Q_1}^\top Q_1 = I_n$ (the* economy $QR$*).*

Since orthogonal matrices preserve the Euclidean norm ($\|Qy\|_2 = \|y\|_2$), we have

$$\begin{aligned} \|Ax - b\|_2^2 &= \left\| Q \begin{pmatrix} R \\ 0 \end{pmatrix} x - b \right\|_2^2 = \left\| \begin{pmatrix} R \\ 0 \end{pmatrix} x - Q^\top b \right\|_2^2 \\ &= \|Rx - c\|_2^2 \ + \ \|r\|_2^2 , \end{aligned}$$

where $Q^\top b = \binom{c}{r}$ with $c \in \mathbb{R}^n$, $r \in \mathbb{R}^{m-n}$. The second term is independent of $x$, and the first is minimized (in fact made zero) by solving the upper triangular system

$$R\,x \;=\; c.$$

The minimal residual is then $\|r\|_2$. No squaring of the condition number occurs: $\kappa_2(R) = \kappa_2(A)$.

### 8.3 Building $Q$: Gram–Schmidt and Householder

**Classical Gram–Schmidt.** Orthogonalize the columns $a_1, \dots, a_n$ of $A$ one at a time:

$$q_1 = \frac{a_1}{\|a_1\|_2}; \qquad v = a_k - \sum_{j=1}^{k-1} \langle q_j,\, a_k \rangle\; q_j, \quad q_k = \frac{v}{\|v\|_2}.$$

The computed $q_k$ may lose orthogonality in finite precision. The *modified Gram–Schmidt* variant subtracts each new projection from the remaining columns as soon as $q_j$ is computed, and is more stable.

**Householder reflections.** For a nonzero vector $v$, the matrix

$$H \;=\; I - \frac{2}{v^\top v}\, v\, v^\top$$

is symmetric and orthogonal: it reflects across the hyperplane orthogonal to $v$. Given a column $z$, one can choose $v$ so that $Hz$ is a multiple of $e_1$. Applying such reflections to zero out the subdiagonal of successive columns produces a $QR$ factorization.

## 9 Eigenvalues, the power method, and the SVD

### 9.1 The eigenvalue problem

Given $A \in \mathbb{R}^{n \times n}$, find $\lambda \in \mathbb{C}$ and $0 \neq x \in \mathbb{C}^n$ with $Ax = \lambda x$. In many applications only the largest (or smallest) eigenvalue and its corresponding eigenvector are needed, and an iterative method that computes one eigenpair at a time is sufficient.

### 9.2 The power method

**Algorithm 9.1** (Power iteration)**.** Choose $v_0$ with $\|v_0\|_2 = 1$. For $k = 1, 2, \dots$:

$$\widetilde{v} \;=\; A\, v_{k-1}; \qquad v_k \;=\; \frac{\widetilde{v}}{\|\widetilde{v}\|_2}; \qquad \lambda^{(k)} \;=\; {v_k}^\top A v_k.$$

**Why it works.** Suppose $A$ has $n$ linearly independent eigenvectors $x_1, \dots, x_n$ with eigenvalues $\lambda_1, \dots, \lambda_n$ satisfying $|\lambda_1| > |\lambda_2| \geq \cdots$. Expand $v_0 = \sum \alpha_j x_j$. Then

$$A^k v_0 \;=\; \sum_j \alpha_j \lambda_j^k x_j \;=\; \lambda_1^k \Big( \alpha_1 x_1 + \sum_{j \geq 2} \alpha_j (\lambda_j/\lambda_1)^k x_j \Big).$$

Since $|\lambda_j/\lambda_1| < 1$ for $j \geq 2$, the bracketed sum decays geometrically, and after normalization $v_k$ aligns with $x_1$. Convergence is linear with rate $|\lambda_2/\lambda_1|$. The Rayleigh quotient $\lambda^{(k)} = {v_k}^\top A v_k$ then estimates $\lambda_1$.

## 9.3 Inverse iteration with shifts

Applying the power method to $B = (A - \mu I)^{-1}$ instead of $A$ yields eigenvalues $1/(\lambda_j - \mu)$, and the dominant one corresponds to the eigenvalue of $A$ closest to $\mu$. Choosing $\mu$ near a target eigenvalue accelerates convergence. Each iteration requires solving $(A - \mu I)\widetilde{v} = v_{k-1}$, which can reuse a single factorization of $A - \mu I$.

## 9.4 Singular value decomposition

**Theorem 9.2.** *Let $A \in \mathbb{R}^{m \times n}$. There exist orthogonal matrices $U \in \mathbb{R}^{m \times m}$, $V \in \mathbb{R}^{n \times n}$ and a diagonal "matrix" $\Sigma \in \mathbb{R}^{m \times n}$ with non–negative entries $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0 = \sigma_{r+1} = \cdots$ such that*

$$A \;=\; U \, \Sigma \, V^\top,$$

*where $r = \operatorname{rank}(A)$. The columns of $U$ are eigenvectors of $AA^\top$, the columns of $V$ are eigenvectors of $A^\top A$, and $\sigma_i^2$ are the corresponding eigenvalues.*

**Properties of the SVD.**

- Geometrically, $A$ acts on a vector by an orthogonal transformation ($V^\top$), an axis–aligned scaling ($\Sigma$), and another orthogonal transformation ($U$). The image of the unit sphere is an ellipsoid with semiaxes $\sigma_1, \dots, \sigma_r$.
- $\|A\|_2 = \sigma_1$ and, for invertible square $A$, $\left\|A^{-1}\right\|_2 = 1/\sigma_n$, so $\kappa_2(A) = \sigma_1/\sigma_n$.
- The truncated SVD $A_k = \sum_{i=1}^{k} \sigma_i \, u_i {v_i}^\top$ is the best rank–$k$ approximation of $A$ in both the 2–norm and the Frobenius norm, with $\|A - A_k\|_2 = \sigma_{k+1}$ (Eckart–Young theorem).
- The pseudo–inverse is $A^\dagger = V \Sigma^\dagger U^\top$, with $\Sigma^\dagger_{ii} = 1/\sigma_i$ for $\sigma_i > 0$ and 0 otherwise.

# Part IV
# Approximation of functions

## 10 Polynomial interpolation

**The setup.** Given $n+1$ data points $\{(x_i, f_i)\}_{i=0}^{n}$ with distinct $x_i$, we look for a function $p$ that *interpolates* the data, i.e. $p(x_i) = f_i$ for every $i$. The data $f_i$ may come from samples of an underlying function $f$, in which case we hope to use $p$ to approximate $f$ at points where we have not sampled.

We restrict to interpolants that are linear combinations of fixed basis functions $\varphi_0, \dots, \varphi_n$:

$$p(x) \;=\; \sum_{j=0}^{n} c_j\, \varphi_j(x).$$

The interpolation conditions become a linear system $Ac = f$, with $A_{ij} = \varphi_j(x_i)$. Whether $A$ is well–conditioned, and whether $c$ is easy to compute, depends entirely on the choice of basis.

### 10.1 The monomial basis

Take $\varphi_j(x) = x^j$. The system matrix becomes the *Vandermonde matrix*

$$V \;=\; \begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & & & & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{pmatrix},$$

with $\det V = \prod_{i<j}(x_j - x_i) \neq 0$. In practice $V$ is ill–conditioned, especially when the nodes cluster in a small interval, so the monomial basis is a poor numerical choice and we look for alternative bases.

### 10.2 Lagrange form

**Theorem 10.1.** *Given $n+1$ distinct points $x_0, \dots, x_n$ and any values $f_0, \dots, f_n$, there is a* unique *polynomial $p_n$ of degree at most $n$ with $p_n(x_i) = f_i$ for all $i$.*

A constructive proof uses the Lagrange basis. Define

$$L_j(x) \;=\; \prod_{i \neq j} \frac{x - x_i}{x_j - x_i}, \qquad j = 0, \dots, n. \tag{14}$$

Each $L_j$ is a polynomial of degree $n$, and by inspection $L_j(x_i) = \delta_{ij}$. Therefore

$$p_n(x) \;=\; \sum_{j=0}^{n} f_j\, L_j(x)$$

satisfies $p_n(x_i) = f_i$ automatically, no linear system to solve. Uniqueness follows because the difference of two interpolants of degree $\le n$ has $n+1$ roots and so must vanish.

The Lagrange form is conceptually simple but expensive: each evaluation costs $O(n^2)$ flops, and adding a new data point requires a complete recomputation of the basis. The barycentric formula uses precomputed weights $w_j = 1/\prod_{i\neq j}(x_j - x_i)$ to bring the cost of an evaluation down to $O(n)$.

### 10.3 Newton form and divided differences

The *Newton basis*

$$\varphi_j(x) \;=\; \prod_{i=0}^{j-1}(x - x_i)$$

gives a lower triangular system for the coefficients, which we solve by forward substitution. The result is the Newton form of the interpolant

$$p_n(x) \;=\; \sum_{j=0}^{n} f[x_0, \dots, x_j] \prod_{i=0}^{j-1}(x - x_i), \tag{15}$$

where the coefficients $f[x_0, \dots, x_j]$ are the *divided differences* of $f$, defined recursively by

$$f[x_i] = f(x_i), \qquad f[x_i, \dots, x_j] = \frac{f[x_{i+1}, \dots, x_j] - f[x_i, \dots, x_{j-1}]}{x_j - x_i}.$$

The $O(n^2)$ table of divided differences is built bottom–up; the diagonal entries are the coefficients in (15).

The Newton form has two advantages:

- **Adaptivity.** Adding a new data point appends one row to the divided–difference table without modifying the others; the new interpolant is the old one plus a single correction term.
- **Nested evaluation.** Like Horner's rule, the Newton form can be evaluated in $O(n)$: $p_n(x) = c_0 + (x - x_0)\big(c_1 + (x - x_1)(c_2 + \cdots)\big)$.

**Example 10.2.** For the data $(1,1), (2,3), (4,3)$ the divided differences are $f[1] = 1$, $f[1,2] = 2$, $f[2,4] = 0$, $f[1,2,4] = -2/3$, giving $p_2(x) = 1 + 2(x-1) - \tfrac{2}{3}(x-1)(x-2)$. Adding the point $(5,4)$ produces $f[4,5] = 1$, $f[2,4,5] = 1/3$, $f[1,2,4,5] = 1/4$, and the cubic interpolant is $p_3(x) = p_2(x) + \tfrac{1}{4}(x-1)(x-2)(x-4)$.

### 10.4 Interpolation error and the Runge phenomenon

**Theorem 10.3** (Interpolation error)**.** *Let $f \in C^{n+1}[a,b]$ and let $p_n$ interpolate $f$ at $n+1$ distinct nodes $x_0, \dots, x_n \in [a,b]$. Then for every $x \in [a,b]$ there exists $\xi = \xi(x) \in [a,b]$ with*

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

*Sketch.* Fix $x$, define $F(t) = f(t) - p_n(t) - \big(f(x) - p_n(x)\big)\,\frac{\prod_i (t-x_i)}{\prod_i (x-x_i)}$, note that $F$ has $n+2$ roots $(x_0, \dots, x_n, x)$ and apply Rolle's theorem $n+1$ times. $\square$

The error has two factors: the smoothness of $f$, encoded in $f^{(n+1)}$, and the placement of the nodes, encoded in the *nodal polynomial* $\prod(x - x_i)$. We have no control over the first, but we can choose the second.

**Runge's example.** For $f(x) = 1/(1+25x^2)$ on $[-1,1]$ with equally spaced nodes, the maximum interpolation error grows with $n$ rather than decays. The high derivatives $f^{(n+1)}$ grow faster than $(n+1)!$ can control. The remedy is to change the node distribution.

**Chebyshev nodes.** The Chebyshev nodes on $[-1,1]$,

$$x_i \;=\; \cos\!\Big(\frac{(2i+1)\pi}{2(n+1)}\Big), \qquad i = 0, \dots, n,$$

minimize $\max_{x\in[-1,1]} |\prod(x - x_i)|$ over all choices of $n+1$ nodes, and interpolation at Chebyshev nodes converges for $f \in C^1[-1,1]$.

## 10.5 Hermite interpolation

If we are given derivative values as well as function values, we can ask for an interpolant matching them all. The simplest case is two data points $t_0, t_1$ each with a value and a derivative – four conditions, four coefficients, hence a unique cubic, the *Hermite cubic*. In Newton's framework one simply duplicates the abscissae: the Newton table accepts repeated nodes provided one interprets divided differences with repeated arguments as derivatives, $f[t,t] = f'(t)$, etc. The general formula and recipe carry over verbatim.

## 10.6 Piecewise interpolation

A high–degree global polynomial is not always the right object. For data that is only piecewise smooth, or simply for stability and locality, we use piecewise polynomials of *low* degree on a partition $a = x_0 < x_1 < \cdots < x_n = b$.

**Piecewise linear.** On each interval $[x_i, x_{i+1}]$, take the linear interpolant of $f$ at the two endpoints. The result is a continuous function with corners. If $h = \max(x_{i+1} - x_i)$ and $f \in C^2$, $\max|f - s| \le \frac{h^2}{8}\,\max|f''|$.

**Cubic splines.** Use a cubic on each subinterval and require the global function to be $C^2$. This leaves two free conditions at the endpoints, fixed by "natural" or "clamped" boundary conditions. Cubic splines give accurate, low–oscillation interpolants and are widely used in practice.

# 11 Best $L^2$ approximation and orthogonal polynomials

Interpolation forces the approximant to match the data exactly. When the data are noisy, or when we want the best smooth approximation of a function in an averaged sense, we instead minimize an $L^2$ error over a chosen finite–dimensional space.

## 11.1 The continuous least squares problem

Fix an interval $[a, b]$ and basis functions $\varphi_0, \dots, \varphi_n$. We seek $v(x) = \sum c_j \varphi_j(x)$ minimizing

$$\|f - v\|_2^2 \;=\; \int_a^b \big(f(x) - v(x)\big)^2 \, dx. \tag{16}$$

Setting $\partial / \partial c_k = 0$ gives the (continuous) normal equations

$$\sum_j \widetilde{B}_{kj}\, c_j = \widetilde{b}_k, \qquad \widetilde{B}_{kj} = \int_a^b \varphi_j \varphi_k \, dx, \quad \widetilde{b}_k = \int_a^b f\, \varphi_k \, dx. \tag{17}$$

The matrix $\widetilde{B}$ is the *Gram matrix* of the basis.

**The Hilbert matrix.** With $\varphi_j(x) = x^j$ on $[0, 1]$, the Gram matrix is $\widetilde{B}_{jk} = 1/(j+k+1)$, the *Hilbert matrix*, whose condition number grows exponentially in $n$. The monomial basis is therefore unsuitable for the continuous least squares problem.

**Orthogonal bases.** If the basis is orthogonal under the chosen inner product, $\int_a^b \varphi_j \varphi_k \, dx = \gamma_j \, \delta_{jk}$, then $\widetilde{B}$ is diagonal and the system trivializes:

$$c_j \;=\; \frac{\int_a^b f\, \varphi_j \, dx}{\int_a^b \varphi_j^2 \, dx}.$$

## 11.2 Inner product spaces and orthogonality

Recall that an *inner product* $\langle \cdot, \cdot \rangle$ on a real vector space is bilinear, symmetric, and positive definite ($\langle f,\, f \rangle > 0$ for $f \neq 0$). On $\mathbb{R}^n$ the standard choice is $\langle x,\, y \rangle = x^{\top} y$. On $L^2([a, b])$ we use

$$\langle f,\, g \rangle \;=\; \int_a^b f(x) g(x) \, dx,$$

and more generally with a positive weight $w(x)$, $\langle f,\, g \rangle_w = \int_a^b w(x) f(x) g(x) \, dx$.

## 11.3 Classical orthogonal polynomial families

Different choices of $[a, b]$ and $w$ give different families.

**Legendre polynomials.** $[-1,1]$ with $w \equiv 1$:

$$P_0 = 1, \quad P_1 = x, \qquad (j+1)P_{j+1}(x) = (2j+1)\,x\,P_j(x) - j\,P_{j-1}(x).$$

Orthogonality: $\int_{-1}^{1} P_j P_k\,dx = \frac{2}{2j+1}\delta_{jk}$.

**Chebyshev polynomials of the first kind.** $[-1,1]$ with $w(x) = 1/\sqrt{1-x^2}$:

$$T_0 = 1, \quad T_1 = x, \qquad T_{j+1}(x) = 2x\,T_j(x) - T_{j-1}(x).$$

Closed form: $T_j(\cos\theta) = \cos(j\theta)$.

**Laguerre polynomials.** $[0,\infty)$ with $w(x) = e^{-x}$:

$$L_0 = 1, \quad L_1 = 1 - x, \qquad (j+1)L_{j+1}(x) = (2j+1-x)L_j(x) - j\,L_{j-1}(x).$$

**Hermite polynomials.** $(-\infty,\infty)$ with $w(x) = e^{-x^2}$:

$$H_0 = 1, \quad H_1 = 2x, \qquad H_{j+1}(x) = 2x\,H_j(x) - 2j\,H_{j-1}(x).$$

**The three–term recurrence is universal.** Every family of polynomials orthogonal with respect to a positive weight satisfies a recurrence of the form

$$\varphi_{j+1}(x) = (x - \beta_j)\,\varphi_j(x) - \gamma_j\,\varphi_{j-1}(x),$$

with $\beta_j, \gamma_j$ given by inner–product integrals. This is the Gram–Schmidt process specialized to the case of multiplying by $x$, and it underlies the practical computation of all the families above.

## 11.4 Chebyshev's min–max property

**Theorem 11.1.** *Among all monic polynomials of degree $n$, the rescaled Chebyshev polynomial $\widetilde{T}_n(x) := 2^{1-n}T_n(x)$ has the smallest maximum norm on $[-1,1]$:*

$$\max_{x\in[-1,1]} \left|\widetilde{T}_n(x)\right| \;=\; 2^{1-n},$$

*and any other monic polynomial of degree $n$ has strictly larger maximum.*

*Sketch.* $\widetilde{T}_n$ attains $\pm 2^{1-n}$ at $n+1$ extrema with alternating sign. Suppose another monic $P_n$ has smaller maximum. Then $\widetilde{T}_n - P_n$ has $n+1$ sign changes, hence $n$ roots; but its degree is at most $n-1$, contradiction. □

This is exactly why Chebyshev nodes are optimal for interpolation: they minimize the nodal polynomial in the error formula of Theorem 10.3.

# 12 Numerical differentiation

Suppose $f$ is given to us as a black box, or as a finite list of sampled values, and we want its derivative. We have already seen one finite–difference formula in (1); here we set things up more systematically and look at the resulting error.

## 12.1 Finite difference formulas via Taylor

**Forward and backward differences.**

$$D_h^+ f(x) \;=\; \frac{f(x+h)-f(x)}{h} \;=\; f'(x) + \tfrac{h}{2} f''(\xi),$$

$$D_h^- f(x) \;=\; \frac{f(x)-f(x-h)}{h} \;=\; f'(x) + O(h).$$

Both are first–order accurate ($O(h)$).

**Centered difference.** Subtract two Taylor expansions: $f(x+h) - f(x-h) = 2hf'(x) + \frac{h^3}{3} f'''(\xi) + \cdots$, which gives

$$D_h^c f(x) \;=\; \frac{f(x+h)-f(x-h)}{2h} \;=\; f'(x) + \tfrac{h^2}{6} f'''(\xi).$$

This is second–order accurate, $O(h^2)$.

**Second derivative.** Adding two Taylor expansions:

$$\frac{f(x+h)-2f(x)+f(x-h)}{h^2} \;=\; f''(x) + \tfrac{h^2}{12} f^{(4)}(\xi),$$

also $O(h^2)$.

## 12.2 General approach: differentiate the interpolant

For unevenly spaced or higher–order schemes, the cleanest recipe is:

1. Build the Lagrange (or Newton) interpolant $p_n(x) = \sum_j f(x_j) L_j(x)$ through the points where $f$ is known.
2. Differentiate *the polynomial* symbolically.
3. Evaluate $p_n^{(k)}$ at the point of interest.

The resulting formulas have weights $L_j^{(k)}(x_0)$ that are just numbers depending on the geometry of the nodes, not on $f$.

### 12.3 The truncation/roundoff trade–off (again)

For the centered difference,

$$\text{total error}(h) \;\lesssim\; \tfrac{M}{6}h^2 + \frac{\varepsilon_{\text{mach}}\,|f(x)|}{h},$$

which is minimized at $h_{\text{opt}} \sim \varepsilon_{\text{mach}}^{1/3}$ with smallest error of order $\varepsilon_{\text{mach}}^{2/3}$.

### 12.4 Richardson extrapolation

If $D_h f = f'(x) + C\,h^p + O(h^{p+1})$, then $D_{h/2}f = f'(x) + C(h/2)^p + O(h^{p+1})$, and the linear combination

$$\frac{2^p D_{h/2}f - D_h f}{2^p - 1} \;=\; f'(x) + O(h^{p+1})$$

*eliminates the leading error term.* Applied to the centered difference ($p = 2$), Richardson extrapolation produces an $O(h^4)$ formula at the price of one extra function evaluation per side. This idea reappears in Romberg integration for quadrature.

## 13 Numerical integration (quadrature)

We approximate $I(f) = \int_a^b f(x)\,dx$ by a finite sum

$$Q_n(f) \;=\; \sum_{j=0}^{n} w_j\, f(x_j),$$

with chosen *nodes* $x_j$ and *weights* $w_j$.

**Interpolatory rules.** A natural construction is to interpolate $f$ at the nodes and integrate the resulting polynomial:

$$I(f) \;\approx\; \int_a^b \sum_{j=0}^{n} f(x_j) L_j(x)\,dx \;=\; \sum_{j=0}^{n} f(x_j) \underbrace{\int_a^b L_j(x)\,dx}_{w_j}.$$

The weights $w_j$ depend only on the geometry of the nodes and are computed once.

### 13.1 Newton–Cotes rules on $[a, b]$

**Midpoint rule ($n = 0$).** Single node $x_0 = (a + b)/2$, weight $w_0 = b - a$: $I_{\text{mid}} = (b - a)\, f\big(\frac{a+b}{2}\big)$. Exact for linear functions. Error $\frac{(b-a)^3}{24} f''(\xi)$.

**Trapezoidal rule ($n = 1$).** Nodes $a, b$, weights $(b-a)/2, (b-a)/2$: $I_{\text{trap}} = \frac{b-a}{2}\big(f(a) + f(b)\big)$. Exact for linear functions. Error $-\frac{(b-a)^3}{12} f''(\xi)$.

**Simpson's rule ($n = 2$).** Three equally spaced nodes: $I_{\mathrm{S}} = \frac{b-a}{6}\big(f(a) + 4f(\frac{a+b}{2}) + f(b)\big)$. Exact for cubic polynomials, with error $-\frac{(b-a)^5}{2880} f^{(4)}(\xi)$.

## 13.2 Composite rules

Applying a basic rule on a single interval $[a, b]$ becomes inaccurate as $b - a$ grows. Instead, partition $[a, b]$ into $r$ equal subintervals of length $h = (b - a)/r$ and apply the basic rule on each subinterval. This gives the *composite trapezoidal rule*

$$T_h(f) \;=\; \frac{h}{2}\Big(f(a) + 2\sum_{i=1}^{r-1} f(a + ih) + f(b)\Big),$$

*composite midpoint*, $M_h(f) = h\sum_{i=0}^{r-1} f(a + (i + \frac{1}{2})h)$, and *composite Simpson*, which use $r/2$ Simpson panels. For smooth $f$ the errors are

$$|I - T_h| \le \tfrac{(b-a)h^2}{12} \max\left|f''\right|, \quad |I - M_h| \le \tfrac{(b-a)h^2}{24} \max\left|f''\right|, \quad |I - S_h| \le \tfrac{(b-a)h^4}{180} \max\left|f^{(4)}\right|.$$

Halving $h$ improves accuracy by a factor of 4 for the trapezoidal and midpoint rules and by a factor of 16 for Simpson's rule.

## 13.3 Gaussian quadrature

Newton–Cotes rules use fixed equally spaced nodes. Treating the nodes as additional unknowns, $n+1$ nodes and $n+1$ weights give $2(n+1)$ parameters, suggesting that one can integrate polynomials of degree up to $2n + 1$ exactly. Gaussian quadrature realizes this bound.

**Theorem 13.1.** *Let $\{P_j\}$ be the orthogonal polynomials on $[-1, 1]$ with respect to weight 1 (i.e. Legendre polynomials). Take the nodes $x_0, \dots, x_n$ to be the roots of $P_{n+1}$ and the weights*

$$w_j \;=\; \int_{-1}^{1} L_j(x)\,dx \;=\; \frac{2}{(1 - x_j^2)\,[P'_{n+1}(x_j)]^2}.$$

*Then the rule $\sum w_j f(x_j)$ is exact for all polynomials of degree $\le 2n + 1$.*

*Sketch.* Write $f$ of degree $\le 2n + 1$ as $f = qP_{n+1} + r$ with $\deg q, \deg r \le n$. Then $\int qP_{n+1} = 0$ by orthogonality ($q$ has degree $\le n$). At the nodes $P_{n+1}(x_j) = 0$, so $f(x_j) = r(x_j)$, and the rule integrates $r$ exactly because it has only degree $n$ and the rule is interpolatory. □

For a general interval $[a, b]$, use the affine change of variables $t = \frac{b-a}{2}x + \frac{a+b}{2}$. Tabulated Gauss–Legendre nodes and weights are widely available for any reasonable $n$.

## 14 The discrete Fourier transform and the FFT

Trigonometric polynomials are the natural basis for periodic functions. Sampled at $m$ equally spaced points $x_k = 2\pi k/m$, the discrete Fourier transform (DFT) of a vector $y = (y_0, \dots, y_{m-1})$ is

$$\widehat{y}_j \;=\; \sum_{k=0}^{m-1} y_k\, \omega_m^{jk}, \qquad \omega_m \;=\; e^{-2\pi i/m},$$

and the inverse DFT is $y_k = \frac{1}{m}\sum_j \widehat{y}_j \omega_m^{-jk}$. A naive evaluation requires $O(m^2)$ complex multiplications. The Cooley–Tukey *fast Fourier transform* (FFT) computes the same vector in $O(m \log_2 m)$ flops.

### 14.1 Divide and conquer

Assume $m = 2\ell$ is even. Split the input vector into its even and odd subsequences and observe

$$\widehat{y}_j \;=\; \sum_{k=0}^{\ell-1} y_{2k}\, \omega_m^{2jk} + \sum_{k=0}^{\ell-1} y_{2k+1}\, \omega_m^{(2k+1)j}.$$

Now $\omega_m^2 = \omega_\ell$, so each of the two sums is itself a length–$\ell$ DFT, of the even part $y^e$ and the odd part $y^o$. Write $\widehat{y}^e, \widehat{y}^o$ for these. Then for $j = 0, \dots, \ell - 1$,

$$\widehat{y}_j \;=\; \widehat{y}^e_j + \omega_m^j\, \widehat{y}^o_j, \qquad \widehat{y}_{j+\ell} \;=\; \widehat{y}^e_j - \omega_m^j\, \widehat{y}^o_j,$$

where the second identity uses $\omega_m^\ell = -1$. These two formulas are the FFT *butterfly*: each pair $(\widehat{y}_j, \widehat{y}_{j+\ell})$ is computed from $(\widehat{y}^e_j, \widehat{y}^o_j)$ by one complex multiplication and one complex addition or subtraction.

If $C(m)$ denotes the cost of an $m$–point DFT, then $C(m) = 2\,C(m/2) + O(m)$, and recursion down to $m = 1$ gives $C(m) = O(m \log_2 m)$.

# Part V
# Numerical methods for ordinary differential equations

## 15 Initial value problems

We finish the course with numerical methods for the initial value problem

$$y'(t) \;=\; f(t, y(t)), \qquad t \in [a, b], \qquad y(a) = y_0. \tag{18}$$

$y$ may be scalar or vector valued; in the latter case $f\colon [a,b] \times \mathbb{R}^m \to \mathbb{R}^m$. We assume $f$ is Lipschitz in $y$, so the IVP has a unique solution on $[a, b]$. We discretize the time interval as $a = t_0 < t_1 < \cdots < t_N = b$ with uniform step $h = (b-a)/N$, $t_i = a + ih$, and we seek approximations $y_i \approx y(t_i)$.

### 15.1 Forward (explicit) Euler

The simplest way to discretize the IVP is to replace $y'$ at $t_i$ by the forward difference quotient:

$$y'(t_i) \;\approx\; \frac{y(t_{i+1}) - y(t_i)}{h}.$$

Combined with the ODE this gives the *forward Euler method*

$$y_{i+1} \;=\; y_i + h\, f(t_i, y_i), \qquad i = 0, 1, \ldots, N-1. \tag{19}$$

This is *explicit*: $y_{i+1}$ is given by an explicit formula in terms of known data.

**Local truncation error.** Plugging the exact solution into the scheme gives $d_i := \frac{y(t_{i+1})-y(t_i)}{h} - f(t_i, y(t_i))$. By Taylor's theorem, $d_i = \frac{h}{2}\, y''(\xi_i)$, so $\max_i |d_i| = O(h)$. The method has *local order* one.

**Global error.** The global error $e_i = y(t_i) - y_i$ satisfies

$$e_{i+1} = e_i + h\big(f(t_i, y(t_i)) - f(t_i, y_i)\big) - h\, d_i.$$

Using the Lipschitz condition $|f(t,y) - f(t,\tilde{y})| \le L\,|y - \tilde{y}|$ and $|d_i| \le \frac{h}{2}\max|y''| =: \frac{Mh}{2}$, one shows by induction

$$|e_i| \;\le\; \frac{Mh}{2L}\big(e^{L(t_i - a)} - 1\big) \;=\; O(h).$$

Forward Euler is therefore convergent of order one.

**Example 15.1.** For $y' = y$, $y(0) = 1$, the exact solution is $y(t) = e^t$ and the Euler iterate is $y_i = (1+h)^i$. At $t = 1$, $y_N = (1+h)^N = (1 + 1/N)^N \to e$ as $N \to \infty$, with $e - (1+h)^{1/h} = O(h)$.

## 16 Absolute stability and stiff equations

So far we have asked: as $h \to 0$, does the numerical solution approach the true one? This is the question of *convergence*. A different question is: for a *fixed* step $h$, does the numerical solution remain bounded when the true solution is itself bounded or decaying? This is the question of *absolute stability*.

### 16.1 The test equation

Apply forward Euler to the linear test equation

$$y' \;=\; \lambda\, y, \qquad y(0) = 1,$$

with $\lambda < 0$, whose true solution $e^{\lambda t}$ decays to zero. Forward Euler gives

$$y_{i+1} \;=\; (1 + h\lambda)\, y_i, \qquad y_i = (1 + h\lambda)^i.$$

For this to remain bounded (and indeed to decay) we need $|1 + h\lambda| \leq 1$, which forces

$$h \;\leq\; \frac{2}{|\lambda|}. \tag{20}$$

For example, if $\lambda = -10^4$, forward Euler requires $h \leq 2 \times 10^{-4}$ regardless of the smoothness of the solution. This is a stability constraint, not an accuracy one.

**Stiff systems.** A problem is called *stiff* when the absolute–stability restriction on $h$ is much more severe than what accuracy alone would require. Stiffness arises when the system has widely separated time scales, e.g. a fast transient coexisting with a slowly varying component.

### 16.2 Backward (implicit) Euler

Replace the forward by a backward difference at $t_{i+1}$:

$$y_{i+1} \;=\; y_i + h\, f(t_{i+1}, y_{i+1}). \tag{21}$$

$y_{i+1}$ now appears on both sides; in general (21) is a (possibly nonlinear) equation that must be solved at each time step (typically by Newton's method). For this reason backward Euler is called an *implicit* method.

**Stability.** For the test equation, $y_{i+1} = y_i + h\lambda y_{i+1}$ gives $y_{i+1} = y_i/(1 - h\lambda)$. For $\lambda < 0$, $1 - h\lambda > 1$ and so $|y_{i+1}| < |y_i|$ *for any* $h > 0$. Backward Euler is therefore *unconditionally stable* (or *A–stable*): the step size is dictated entirely by accuracy, not stability. This is why backward Euler – and the more sophisticated implicit methods that generalize it – is the right tool for stiff problems.

**Order.** A Taylor expansion shows that the local truncation error of backward Euler is also $O(h)$, so the method is first–order accurate.

**Trade–offs.** Implicit methods are more expensive per step than explicit ones because each step requires the solution of an algebraic equation, but they remove the absolute–stability constraint and allow much larger step sizes on stiff problems.

# Appendices: Homework problems

The following appendices collect the twelve homework assignments of the course as a single problem set, organized by week. Each problem is numbered within its homework (Problem 1.1, 1.2, ... for Homework 1, and so on). Problems originally drawn from Ascher & Greif, *A First Course in Numerical Methods*, or from Süli & Mayers, *An Introduction to Numerical Analysis*, are credited next to the problem heading.

## Homework 1

**Problem A.1** (Ascher & Grief, Ex. 1.4(2))**.** Carry out derivation and calculations analogous to those in Example 1.2, using the expression

$$\frac{f(x_0+h)-f(x_0-h)}{2h}$$

for approximating the first derivative $f'(x_0)$.

Show that the error is $O(h^2)$. More precisely, the leading term of the error is $-\frac{1}{6}h^2 f'''(x_0)$ when $f'''(x_0) \neq 0$.

**Problem A.2** (Ascher & Grief, Ex. 1.4(3))**.** Carry out calculations similar to those in Example 1.3 using the approximation from Exercise 2. Observe similarities and differences by comparing your graph with that in Figure 1.3.

**Problem A.3** (Ascher & Grief, Ex. 1.4(4))**.** Following Example 1.5, assess the conditioning of the problem of evaluating

$$g(x) = \tanh(cx) = \frac{\exp(cx)-\exp(-cx)}{\exp(cx)+\exp(-cx)}$$

near $x = 0$ as the positive parameter $c$ grows.

**Problem A.4** (Ascher & Grief, Ex. 1.4(5))**.** Consider the problem presented in Example 1.6. There we saw a numerically unstable procedure for carrying out the task.

(a) Derive a formula for approximately computing these integrals based on evaluating $y_{n-1}$ given $y_n$.

(b) Show that for any given value $\varepsilon > 0$ and positive integer $n_0$, there exists an integer $n_1 \geq n_0$ such that taking $y_{n_1} = 0$ as a starting value will produce integral evaluations $y_n$ with an absolute error smaller than $\varepsilon$ for all $0 < n \leq n_0$.

(c) Explain why your algorithm is stable.

(d) Write a MATLAB/Python function that computes the value of $y_{20}$ within an absolute error of at most $10^{-5}$. Explain how you choose $n_1$ in this case.

**Problem A.5** (Ascher & Grief, Ex. 2.5(11))**.** Show that

$$\ln(x+\sqrt{x^2-1}) = -\ln(x-\sqrt{x^2-1}).$$

Which of the two formulas is more suitable for numerical computation? Explain why, and provide a numerical example in which the difference in accuracy is evident.

**Problem A.6** (Ascher & Grief, Ex. 2.5(12))**.** For the following expressions, state the numerical difficulties that may occur, and rewrite the formulas in a way that is more suitable for numerical computation:

(a) $\sqrt{x+\frac{1}{x}} - \sqrt{x-\frac{1}{x}}$, where $x \gg 1$.

(b) $\sqrt{\frac{1}{a^2}+\frac{1}{b^2}}$, where $a \approx 0$ and $b \approx 1$.

**Problem A.7** (Ascher & Grief, Ex. 2.5(14))**.** Consider the approximation to the first derivative

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}.$$

The truncation (or discretization) error for this formula is $\mathcal{O}(h)$. Suppose that the absolute error in evaluating the function $f$ is bounded by $\varepsilon$ and let us ignore the errors generated in basic arithmetic operations.

(a) Show that the total computational error (truncation and rounding combined) is bounded by

$$\frac{Mh}{2} + \frac{2\varepsilon}{h},$$

where $M$ is a bound on $|f''(x)|$.

(b) What is the value of $h$ for which the above bound is minimized?

(c) The rounding unit we employ is approximately equal to $10^{-16}$. Use this to explain the behavior of the graph in Example 1.3. Make sure to explain the shape of the graph as well as the value where the apparent minimum is attained.

(d) It is not difficult to show, using Taylor expansions, that $f'(x)$ can be approximated more accurately (in terms of truncation error) by

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}.$$

For this approximation, the truncation error is $\mathcal{O}(h^2)$. Generate a graph similar to Figure 1.3 (please generate only the solid line) for the same function and the same value of $x$, namely, for $\sin(1.2)$, and compare the two graphs. Explain the meaning of your results.

## Homework 2

**Problem B.1** (Ex. 1)**.** For $x > 0$ consider the equation

$$x + \ln x = 0$$

(a) Prove that there is exactly one root, $0 < x^\star < \infty$.

(b) Plot a graph of the function on the interval [0.1,1].

(c) As you can see from the graph, the root is between 0.5 and 0.6. Write codes for finding the root, using the following:

i. The bisection method, with the initial interval [0.5,0.6]. Explain why this choice of the initial interval is valid.

ii. A linearly convergent fixed point iteration, with $x_0 = 0.5$. Show that the conditions of the Fixed Point Theorem (for the function $g$ you have properly selected in the fixed point iteration) are satisfied.

iii. Newton's method, with $x_0 = 0.5$.

iv. The secant method, with $x_0 = 0.5$ and $x_1 = 0.6$.

For each of the methods:

- Use $|x_k - x_{k-1}| < 10^{-10}$ as a convergence criterion. - Print out the iterates and show the progress in the number of correct decimal digits throughout the iteration. - Explain the convergence behavior and how it matches theoretical expectations.

**Problem 2** Cube Root Computation

Write a MATLAB/Python script for computing the cube root of a number, $x = a^{1/3}$, with only basic arithmetic operations using Newton's method, by finding a root of the function $f(x) = x^3 - a$. Run your program for $a = 0, 2, 10$. For each of these cases, start with an initial guess reasonably close to the solution. As a stopping criterion, require the function value whose root you are searching to be smaller than $10^{-8}$. Print out the values of $x_k$ and $f(x_k)$ in each iteration. Comment on the convergence rates and explain how they match your expectations. (Note: you can re-use your Newton method code in previous problems)

**Problem 3** The function

$$f(x) = (x-1)^2 e^x$$

has a double root at $x = 1$.

(a) Derive Newton's iteration for this function. Show that the iteration is well-defined so long as $x_k \neq -1$ and that the convergence rate is expected to be similar to that of the bisection method (and certainly not quadratic).

(b) Implement Newton's method and observe its performance starting from $x_0 = 2$.

(c) How easy would it be to apply the bisection method? Explain.

**Problem 4**

(a) Derive a third order method for solving $f(x) = 0$ in a way similar to the derivation of Newton's method, using evaluations of $f(x_n)$, $f'(x_n)$, and $f''(x_n)$. The following remarks may be helpful in constructing the algorithm:

- Use the Taylor expansion with three terms plus a remainder term. - Show that in the course of derivation a quadratic equation arises, and therefore two distinct schemes can be derived.

(b) Show that the order of convergence (under the appropriate conditions) is cubic.

(c) Can you speculate what makes this method less popular than Newton's method, despite its cubic convergence? You can try some experiments in your answer if necessary.

## Homework 3

**Problem C.1** (Ex. 1)**.** Consider the problem

$$\begin{aligned} x_1 - x_2 + 3x_3 &= 2, \\ x_1 + x_2 &= 4, \\ 3x_1 - 2x_2 + x_3 &= 1. \end{aligned}$$

Carry out Gaussian elimination in its simplest form for this problem. What is the resulting upper triangular matrix? Proceed to find the solution by backward substitution.

**Problem C.2** (Ex. 2)**.** The Gauss–Jordan method used to solve a linear system can be described as follows. Augment $A$ by the right-hand-side vector $\mathbf{b}$ and proceed as in Gaussian elimination, except use the diagonal element $a_{kk}^{(k-1)}$ to eliminate not only $a_{ik}^{(k-1)}$ for $i = k+1, \ldots, n$ but also the elements $a_{ik}^{(k-1)}$ for $i = 1, \ldots, k-1$, i.e., all elements in the $k$th column other than the pivot. Upon reducing $(A|\mathbf{b})$ into

$$\begin{bmatrix} a_{11}^{(n-1)} & 0 & \cdots & 0 & b_1^{(n-1)} \\ 0 & a_{22}^{(n-1)} & \ddots & \vdots & b_2^{(n-1)} \\ \vdots & \ddots & \ddots & 0 & \vdots \\ 0 & \cdots & 0 & a_{nn}^{(n-1)} & b_n^{(n-1)} \end{bmatrix},$$

the solution is obtained by setting

$$x_k = \frac{b_k^{(n-1)}}{a_{kk}^{(n-1)}}, \quad k = 1, \ldots, n.$$

This procedure circumvents the backward substitution part necessary for the Gaussian elimination algorithm.

(a) Write a pseudocode for this Gauss–Jordan procedure; i.e., describe what are the operations in $k$-th step of the algorithm. You may assume that no pivoting (i.e., no row interchanging) is required.

(b) Show that the Gauss–Jordan method requires $n^3 + \mathcal{O}(n^2)$ floating point operations for one right-hand-side vector $\mathbf{b}$—roughly 50% more than what's needed for Gaussian elimination.

**Problem C.3** (Ex. 3)**.** Let $A$ and $T$ be two nonsingular, $n \times n$ real matrices. Furthermore, suppose we are given two matrices $L$ and $U$ such that $L$ is unit lower triangular, $U$ is upper triangular, and

$$TA = LU$$

Write an algorithm that will solve the problem

$$A\mathbf{x} = \mathbf{b}$$

for any given vector $\mathbf{b}$ in $\mathcal{O}(n^2)$ complexity. Explain briefly yet clearly why your algorithm requires only $\mathcal{O}(n^2)$ flops (you may assume without proof that solving an upper triangular or a lower triangular system requires only $\mathcal{O}(n^2)$ flops).

**Problem C.4** (Ex. 4)**.** Let

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

The matrix $A$ can be decomposed using partial pivoting as

$$PA = LU,$$

where $U$ is upper triangular, $L$ is unit lower triangular, and $P$ is a permutation matrix. Find the $4 \times 4$ matrices $U$, $L$, and $P$ (either by hand, or by writing a code, or by using a package in MATLAB or Python)

```
# or code goes here
```

## Homework 4

**Problem D.1** (Ex. 1)**.** The **Cholesky decomposition** is typically used for symmetric positive-definite matrices. For a tridiagonal matrix (a matrix where only the main diagonal and the diagonals immediately above and below it have non-zero entries), a specialized algorithm can efficiently perform the decomposition.

For a general symmetric tridiagonal matrix $A$, the Cholesky decomposition seeks to find a lower triangular matrix $L$ such that:

$$A = LL^T$$

Let the matrix $A$ be represented as:

$$A = \begin{pmatrix} a_1 & b_1 & 0 & \cdots & 0 \\ b_1 & a_2 & b_2 & \cdots & 0 \\ 0 & b_2 & a_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & a_n \end{pmatrix}$$

Here, $a_i$ represents the diagonal entries and $b_i$ represents the off-diagonal entries.

(a) Show that the Cholesky factor $L$ will also have a tridiagonal structure:

$$L = \begin{pmatrix} \ell_1 & 0 & 0 & \cdots & 0 \\ \ell_{21} & \ell_2 & 0 & \cdots & 0 \\ 0 & \ell_{32} & \ell_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \ell_n \end{pmatrix}$$

For simplicity, here to get full points, you only need to show this rigorously for $n = 4$.

(b) for general $n$, provide an algorithm to calculate $\ell_i$ and $\ell_{i,i-1}$ (write down the iterations of your algorithm). What is the complexity of the algorithm? How does it compare with the complexity of Cholesky factorization for a general positive definite matrix?

**Problem D.2** (Ex. 2)**.** Let us explore matrix norms and condition numbers.

(a) For the following matrix given by

$A = \begin{bmatrix} 1 & -2 \\ 3 & -1 \end{bmatrix}$,

calculate $\|A\|_1$, $\|A\|_2$, $\|A\|_\infty$ as well as the condition numbers for each norm by hand (either analytically or by using codes). Is $A$ well or ill-conditioned?

(b) Recall the formulas of 1-norm, and $\infty$-norm of matrices in the lectures. If you assume that taking the absolute value and determining the maximum does not contribute to the overall computational cost, how many flops (floating point operations) are needed to calculate $\|A\|_1$ and $\|A\|_\infty$ for $A \in \mathbb{R}^{n\times n}$? By what factor will the calculation time increase when you double the size of matrix size?

(c) Now implement a simple code that calculates $\|A\|_1$ and $\|A\|_\infty$ for a matrix of any size $n \geq 1$. Using system sizes of $n_1 = 100$, $n_{k+1} = 2n_k, k = 1, \ldots, 7$, determine how long your code takes to calculate $\|A\|_1$ and $\|A\|_\infty$ for a matrix $A \in \mathbb{R}^{n_i \times n_i}$ with random entries and report the results. Can you confirm the estimate from (b)?

(d) Python has the build-in function norm to calculate matrix norms. Calculate for the system sizes in (c) $\|A\|_1$ and $\|A\|_\infty$ using both your implementation and Python's norm function, determine for each $n_i$ how long each code takes and plot the results in one graph. Compare.

```
# or code goes here

#### random matrix generated by A = np.random.rand(n, n)
```

**Problem D.3** (Ex. 3)**.** Let $A, B \in \mathbb{R}^{n\times n}$ and let the matrix norm $\|\cdot\|$ be induced by a vector norm $\|\cdot\|$.

(a) Show that $\|AB\| \leq \|A\|\|B\|$.

(b) For the identity matrix $I \in \mathbb{R}^{n\times n}$, show that $\|I\| = 1$.

(c) For $A$ invertible, show that $\kappa(A) \geq 1$, where $\kappa(A)$ is the condition number of that matrix $A$ corresponding to the norm $\|\cdot\|$. You can use the above two properties with $B := A^{-1}$ for your argument.

(d) Argue that the Frobenius matrix norm $\|A\|_F := \left(\sum_{i,j=1}^n a_{ij}^2\right)^{1/2}$ cannot be induced by a suitable vector norm. Hint: use (b).

**Problem D.4** (Ex. 4)**.** Estimates for vector and matrix norms.

(a) Show that, for any $v \in \mathbb{R}^n$, we have

$\|v\|_\infty \leq \|v\|_2 \leq \sqrt{n}\|v\|_\infty$ and $\|v\|_2^2 \leq \|v\|_1\|v\|_\infty$

In each case, give an example of a nonzero $v$ for which equality is obtained.

(b) Using the problem above, show that for $A \in \mathbb{R}^{m\times n}$,

$\|A\|_\infty \leq \sqrt{n}\|A\|_2$ and $\|A\|_2 \leq \sqrt{m}\|A\|_\infty$

In each case, give an example of a nonzero matrix $A$ for which equality is obtained.

**Problem D.5** (Ex. 5)**.** Let $A \in \mathbb{R}^{n\times n}$ be invertible. Let $b \in \mathbb{R}^n 0, b \neq 0$, and $Ax = b, Ax' = b'$ and denote the perturbations by $\Delta b = b' - b$ and $\Delta x = x' - x$. Show that the inequality obtained in our lecture is sharp. That is, find vectors $b, \Delta b$ for which

$\frac{\|\Delta x\|_2}{\|x\|_2} = \kappa_2(A)\frac{\|\Delta b\|}{\|b\|_2}$
(Hint: consider the eigenvectors of $A^TA$, or singular vectors of $A$.)

## Homework 5

**Problem E.1** (Ex. 1)**.** Consider the $2 \times 2$ matrix

$$A = \begin{pmatrix} 2 & -1 \\ -1 & 2 \end{pmatrix},$$

and suppose we are required to solve $A\mathbf{x} = \mathbf{b}$.

(a) Write down explicitly the iteration matrices corresponding to the Jacobi, Gauss–Seidel, and SOR schemes.

(b) Find the spectral radius of the Jacobi and Gauss–Seidel iteration matrices and the asymptotic rates (which are $-\log$(spectral radius)) of convergence for these two schemes.

(c) Calculate the spectral radius of the SOR iteration matrix. Plot a graph of the spectral radius of the SOR iteration matrix vs. the relaxation param eter $\omega$ for $0 \leq \omega \leq 2$.

(d) Find the optimal SOR parameter, $\omega^*$. What is the spectral radius of the corresponding iteration matrix?

(e) Create a plot showing the convergence behavior for Jacobi, Gauss–Seidel, and the optimal SOR schemes for $b = [1; 1]$

**Problem E.2** (Ex. 2)**.** Let $\alpha$ be a scalar, and consider the iterative scheme for solving $Ax = b$:

$$x_{k+1} = x_k + \alpha(b - Ax_k).$$

(a) If $A = M - N$ is the splitting associated with this method, state what $M$ and the iteration matrix $T$ are.

(b) Suppose $A$ is symmetric positive definite and its eigenvalues are $\lambda_1 > \lambda_2 > \cdots > \lambda_n > 0$.

i. Derive a condition on $\alpha$ that guarantees convergence of the scheme to the solution $x$ for any initial guess.

ii. Show that the best value for the step size in terms of maximizing the speed of convergence is $\alpha = \frac{2}{\lambda_1+\lambda_n}$. Find the spectral radius of the iteration matrix in this case, and express it in terms of the 2-norm condition number of $A$. When the condition number increases, how does the convergence rate change?

**Problem E.3** (Ex. 3)**.** Work out the least squares solution when

$$A = \begin{bmatrix} 2 & -1 \\ 0 & 1 \\ -2 & 2 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ -5 \\ 6 \end{bmatrix}$$

What is pseudoinverse $A^+$ of the matrix $A$?

**Problem E.4** (Ex. 4)**.** Prove that if $A$ is a nonsingular square matrix, then $A^+ = A^{-1}$.

2. Let $x = A^+b$. Prove that the vector $Ax$ is the vector in the column space (i.e., range) of $A$ that is closest to $b$, in the sense of the 2-norm.

## Homework 6

**Problem F.1** (Ex. 1)**.** Prove that the Householder matrix $H = I - 2vv^T$ for some $\|v\|_2 = 1$ is an orthogonal matrix. What are the eigenvalues of $H$?

(2) Use Householder matrices to transform

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 2 \\ -2 & 0 \end{bmatrix}$$

to a upper triangular matrix. Write down the corresponding vector $v$ in the definition of the Householder matrix and the resulting QR decomposition of the matrix $A$. Use this QR decomposition to solve the least square problem

$$\min_x \|Ax - b\|_2$$

where

$$b = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}$$

You can use computer to help you obtain a numerical answer (or you can provide an analytic answer by hand)

```
### codes go here
```

**Problem F.2** (Ex. 2)**.** Often in practice, an approximation of the form

$$u(t) = \gamma_1 e^{\gamma_2 t}$$

is sought for a data fitting problem, where $\gamma_1$ and $\gamma_2$ are constants. Assume given data $(t_1, z_1), (t_2, z_2), \ldots, (t_m, z_m)$, where $z_i > 0$, $i = 1, 2, \ldots, m$, and $m > 0$.

(a) Explain in one brief sentence why the techniques we learned in least squares cannot be directly applied to find this $u(t)$.

(b) Considering instead

$$v(t) = \ln u(t) = (\ln \gamma_1) + \gamma_2 t,$$

it makes sense to define $b_i = \ln z_i$, $i = 1, 2, \ldots, m$, and then find coefficients $x_1$ and $x_2$ such that $v(t) = x_1 + x_1 t$ is the best least squares fit for the data

$$(t_1, b_1), (t_2, b_2), \ldots, (t_m, b_m).$$

Using this method, find $u(t)$ for the data

| $i$ | 1 | 2 | 3 |
|---|---|---|---|
| $t_i$ | 0.0 | 1.0 | 2.0 |
| $z_i$ | $e^{0.1}$ | $e^{0.9}$ | $e^2$ |

Plot the figure of the function $u(t)$ and mark the above data

**Problem 3** [Eigenvalues Review] Prove the following statements, using the basic definition of eigenvalues and eigenvectors, or give a counterexample showing the statement is not true. Assume $A \in \mathbb{R}^{n\times n}, n \geq 1$.

(a) If $\lambda$ is an eigenvalue of $A$ and $\alpha \in \mathbb{R}$, then $\lambda + \alpha$ is an eigenvalue of $A + \alpha I$, where $I$ is the identity matrix.

(b) If $\lambda$ is an eigenvalue of $A$ and $\alpha \in \mathbb{R}$, then $\alpha\lambda$ is an eigenvalue of $\alpha A$.

(c) If $\lambda$ is an eigenvalue of $A$, then for any positive integer $k$, $\lambda^k$ is an eigenvalue of $A^k$.

(d) If $B$ is "similar" to $A$, which means that there is a nonsingular matrix $S$ such that $B = SAS^{-1}$, then if $\lambda$ is an eigenvalue of $A$, it is also an eigenvalue of $B$. How do the eigenvectors of $B$ relate to the eigenvectors of $A$?

(e) Every matrix with $n \geq 2$ has at least two distinct eigenvalues, say $\lambda$ and $\mu$, with $\lambda \neq \mu$.

(f) Every real matrix has a real eigenvalue.

(g) If $A$ is singular, then it has an eigenvalue equal to zero.

(h) If all the eigenvalues of a matrix $A$ are zero, then $A = 0$.

**Problem 4** Power Method and Inverse Iteration.

(a) Implement the Power Method for an arbitrary matrix $A \in \mathbb{R}^{n\times n}$ and an initial vector $x_0 \in \mathbb{R}^n$.

(b) Use your code to find an eigenvector of

$$A = \begin{bmatrix} -2 & 1 & 4 \\ 1 & 1 & 1 \\ 4 & 1 & -2 \end{bmatrix},$$

starting with $x_0 = (1, 2, -1)^T$ and $x_0 = (1, 2, 1)^T$. Report the first 5 iterates for each of the two initial vectors. Then use Python's built-in functions to examine the eigenvalues and eigenvectors of $A$. Where do the sequences converge to? Why do the limits not seem to be the same?

(c) Implement the Inverse Power Method for an arbitrary matrix $A \in \mathbb{R}^{n\times n}$, an initial vector $x_0 \in \mathbb{R}^n$ and an initial eigenvalue guess $\theta \in \mathbb{R}$.

(d) Use your code from (c) to calculate all eigenvectors of $A$. You may pick appropriate values for $\theta$ and the initial vector as you wish (obviously not the eigenvectors themselves). Always report the first 5 iterates and explain where the sequence converges to and why.

```
## codes (you can add more cells if there are multiple parts in your solutions)
```

## Homework 7

**Problem G.1** (Ex. 1)**.** [Space of polynomials $P_n$] Let $P_n$ be the space of functions defined on $[-1, 1]$ that can be described by polynomials of degree less of equal to $n$ with coefficients in $\mathbb{R}$. $P_n$ is a linear space in the sense of linear algebra, in particular, for $p, q \in P_n$ and $a \in \mathbb{R}$, also $p + q$ and $ap$ are in $P_n$. Since the monomials $1, x, x^2, \ldots, x^n$ are a basis for $P_n$, the dimension of that space is $n + 1$.

(a) Show that for pairwise distinct points $x_0, x_1, \ldots, x_n \in [-1, 1]$, the Lagrange polynomials $L_k(x)$ are in $P_n$, and that they are linearly independent, that is, for a linear combination of the zero polynomial with Lagrange polynomials with coefficients $\alpha_k$, i.e.,

$$\sum_{k=0}^{n} \alpha_k L_k(x) = 0 \text{ (the zero polynomial)}$$

necessarily follows that $\alpha_0 = \alpha_1 = \ldots = \alpha_n = 0$. Note that this implies that the $(n + 1)$ Lagrange polynomials also form a basis of $P_n$.

(b) Since both the monomials and the Lagrange polynomials are a basis of $P_n$, each $p \in P_n$ can be written as linear combination of monomials as well as Lagrange polynomials, i.e.,

$$p(x) = \sum_{k=0}^{n} \alpha_k L_k(x) = \sum_{k=0}^{n} \beta_k x^k,$$

with appropriate coefficients $\alpha_k, \beta_k \in \mathbb{R}$. As you know from basic matrix theory, there exists a basis transformation matrix that converts the coefficients $\alpha = (\alpha_0, \ldots, \alpha_n)^T$ to the coefficients $\beta = (\beta_0, \ldots, \beta_n)^T$. Show that this basis transformation matrix is given by the so-called Vandermonde matrix $V \in \mathbb{R}^{n+1 \times n+1}$ given by

$$V = \begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} & x_1^n \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^{n-1} & x_n^n \end{pmatrix},$$

i.e., the relation between $\alpha$ and $\beta$ in (1) is given by $\alpha = V\beta$. An easy way to see this is to choose appropriate $x$ in equation (1).

(c) Note that since $V$ transforms one basis into another basis, it must be an invertible matrix. Let us compute the condition number of $V$ numerically. Compute the 2-based condition number $\kappa_2(V)$ for $n = 5, 10, 20, 30$ with uniformly spaced nodes $x_i = -1+(2i)/n, i = 0, \ldots, n$. Based on the condition numbers, can this basis transformation be performed accurately?

```
### codes go here
```

**Problem G.2** (Ex. 2)**.** [Polynomial interpolation versus least squares fitting]
Given

| i | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| X | 0.0 | 0.5 | 1.0 | 1.5 | 2.0 | 2.5 |
| Y | 0.0 | 0.20 | 0.27 | 0.30 | 0.32 | 0.33 |

(a) Write down the least squares problem associated to finding the cubic best fit polynomial

$$Y = ax^3 + bx^2 + cx + d$$

using (i) all six points, (ii) only the data for $i = 0, 1, 2, 3, 4$, and (iii) $i = 0, 1, 2, 3$. In each case solve the system and plot both the data points and the polynomial. Why is case (iii) not a least squares problem?

(b) What is the degree of the polynomial you would have to use so that the solution interpolates (i.e., goes through) all six data points? Perform the interpolation and plot the figure.

(c) Write the answer in (b) in terms of linear combinations of Lagrange polynomials. Plot the figure of these Lagrange polynomials and output the coefficients.

```
### codes go here
```

## Homework 8

**Problem H.1** (Ex. 1)**.** [Polynomial interpolation and error estimation] Let us interpolate the function $f : [0, 1] \to \mathbb{R}$ defined by $f(x) = \exp(3x)$ using the nodes $x_i = i/2$, $i = 0, 1, 2$ by a quadratic polynomial $p_2 \in P_2$.

(a) Use Newton's basis and compute (numerically) the corresponding coefficients of Newton's basis $c_j \in \mathbb{R}$. Plot $p_2$ and $f$ in the same graph.

(b) Compare the exact interpolation error $E_f(x) := f(x) - p_2(x)$ at $x = 3/4$ with the estimate $|E_f(x)| \leq \frac{M_{n+1}}{(n+1)!}|\pi_{n+1}(x)|$, where $M_{n+1} = \max_{z\in[0,1]} |f^{(n+1)}(z)|$, $f^{(n+1)}$ is the $(n + 1)$st derivative of $f$, and $\pi_{n+1}(x) = (x - x_0)(x - x_1)(x - x_2)$.

```
### codes go here
```

**Problem H.2** (Ex. 2)**.** [Polynomial interpolation]

Consider linear interpolation of $f(x) = x^3$ at $x_0 = 0$ and $x_1 = 1$.

a) For given $x$, find the value of $\xi = \xi(x)$ for which $f(x) - p_1(x) = \frac{f''(\xi)}{2}(x - x_0)(x - x_1)$.

b) Repeat for $f(x) = (2x - 1)^4$.

**Problem H.3** (Ex. 3)**.** Interpolate the Runge function $f(x) = 1/(1+25x^2), -1 \leq x \leq 1$ at Chebyshev points for $n$ from 10 to 170 in increments of 10. Calculate the maximum interpolation error on the uniform evaluation mesh $x = -1 : 0.001 : 1$ and plot the error vs. polynomial degree (with y-axis in log scale). What will happen if we use uniform points rather than Chebyshev points for interpolation?

```
### codes go here
```

**Problem H.4** (Ex. 4)**.** We know in the class that under the Newton basis, the interpolation polynomial has the form

$$p_n(x) = c_0 + c_1 (x - x_0) + \cdots + c_n (x - x_0)(x - x_1) \cdots (x - x_{n-1}).$$

where $c_n = f[x_0, x_1, \ldots, x_n]$ is the $n$-th divided difference.

In the class, we proved the formula for $n \leq 2$.

Prove the formula is correct for $n = 3$ (using the recursive definition of the divided difference). That is, prove the coefficient $c_3 = f[x_0, x_1, \ldots, x_3]$

(bonus) Prove the formula for general $n$.

# Homework 9

**Problem I.1** (Ex. 1)**.** (a) Write a script that interpolates $f(x) = \cosh(x) = \frac{e^x + e^{-x}}{2}$ with an osculating polynomial that matches both $f(x)$ and $f'(x)$ at abscissae $x_0 = 1$ and $x_1 = 3$. Generate a plot (with logarithmic vertical axis) comparing $f(x)$ and the interpolating polynomial and another plot showing the error in your interpolant over this interval.

(b) Modify the code to generate another interpolant that matches $f(x)$ and $f'(x)$ at the abscissae $x_0 = 1$, $x_1 = 2$, and $x_2 = 3$. Generate two more plots: comparison of function to interpolant, and error. Compare the quality of this new polynomial to the quality of the polynomial of part (a).

The comments for the plots can be very simple.

```
### codes go here
```

**Problem I.2** (Ex. 2)**.** Consider the piecewise constant interpolation of the function $f(x) = \sin(x)$, $0 \leq x \leq 381$, at points $x_i = ih$, where $h = 0.1$. Thus, our interpolant satisfies $v(x) = \sin(ih)$, $(i - 0.5)h \leq x < (i + 0.5)h$, for $i = 0, 1, \ldots, 3810$.

(a) Find a bound for the error in this interpolation.

(b) Based on (a), how many leading digits in $v(x)$ are guaranteed to agree with those of $f(x)$, for any $0 \leq x \leq 381$?

**Problem I.3** (Ex. 3)**.** Construct the second degree polynomial $q_2(t)$ that approximates $g(t) = \sin(\pi t)$ on the interval $[0, 1]$ by minimizing $\int_0^1 [g(t) - q_2(t)]^2 dt$. Some useful integrals:

$\int_0^1 (6t^2 - 6t + 1)^2 dt = \frac{1}{5}$, $\int_0^1 \sin(\pi t) dt = \frac{2}{\pi}$,

$\int_0^1 t \sin(\pi t) dt = \frac{1}{\pi}$, $\int_0^1 t^2 \sin(\pi t) dt = \frac{\pi^2 - 4}{\pi^3}$.

**Problem I.4** (Ex. 4)**.** The Legendre polynomials satisfy

$\int_{-1}^1 \phi_j(x)\phi_k(x) dx = 0$ for $j \neq k$

$\int_{-1}^1 \phi_j(x)\phi_k(x) dx = \frac{2}{2j+1}$ for $j = k$

Suppose that the best fit problem is given on the interval $[a, b]$.

Show that with the transformation $t = \frac{1}{2}[(b-a)x + (a+b)]$ and a slight change of notation, we have

$\int_a^b \phi_j(t)\phi_k(t)dt = 0$ for $j \neq k$

$\int_a^b \phi_j(t)\phi_k(t)dt = \frac{b-a}{2j+1}$ for $j = k$

## Homework 10

**Problem J.1** (Ex. 1** [weighted continuous least squares)**.** ] Consider

$$\min_{\mathbf{c}} \psi(\mathbf{c}) = \int_a^b w(x)(f(x) - v(x))^2 dx,$$

where

$$v(x) = \sum_{j=0}^{n} c_j \phi_j(x).$$

(1) Derive the normal equation that $c_j, 0 \leq j \leq n$ satisfies. You need to write down the detailed steps for the derivation; simply writing down the final equation won't get credits.

(2) When $a = -1, b = 1, w(x) = \frac{1}{\sqrt{1-x^2}}$, what is the name of the corresponding orthogonal polynomial with respect such weight function? What is advantage of using the orthogonal polynomial as basis functions compared to monomial basis functions?

(3) The orthogonal polynomials in (2) take the form

$$\phi_j(x) = \cos(j \arccos x)$$

Prove

$$\int_{-1}^{1} w(x)\phi_j(x)\phi_k(x)dx = \begin{cases} 0, & j \neq k, \\ \frac{\pi}{2}, & j = k. \end{cases}$$

**Problem J.2** (Ex. 2** [Three term recursion)**.** Let $\phi_0(x), \phi_1(x), \phi_2(x), \ldots$ be a sequence of orthogonal polynomials on an interval $[a, b]$ with respect to a positive weight function $w(x)$. Show that these orthogonal polynomials satisfy the following three term recursion]

$$\phi_{k+1}(x) = c_{k+1}x\phi_k(x) + c_k\phi_k(x) + c_{k-1}\phi_{k-1}(x), \quad k \geq 1.$$

for some constants $c_{k+1}, c_k, c_{k-1}$

**Problem J.3** (Ex. 3)**.** Let $\phi_0(x), \phi_1(x), \phi_2(x), \ldots$ be a sequence of orthogonal polynomials on an interval $[a, b]$ with respect to a positive weight function $w(x)$. Let $x_1, \ldots, x_n$ be the $n$ zeros of $\phi_n(x)$; it is known that these roots are real and $a < x_1 < \cdots < x_n < b$.

Show that the Lagrange polynomials of degree $n-1$ based on these points are orthogonal to each other, so we can write

$$\int_a^b w(x)L_j(x)L_k(x)dx = 0, \quad j \neq k,$$

where

$$L_j(x) = \prod_{k=1, k\neq j}^{n} \frac{x - x_k}{x_j - x_k}, \quad 1 \leq j \leq n.$$

**Problem J.4** (Ex. 4)**.** Given function $f(x) = x^2 - \pi^2$ for $x \in [-\pi, \pi]$. Consider the Fourier series approximation

$$v(x) = \frac{a_0}{2} + a_l \cos(lx) + \sum_{k=1}^{l-1} \left(a_k \cos(kx) + b_k \sin(kx)\right).$$

What are the coefficient $a_k, b_k$? Write down the for them; you do not need to calculate the exact answer.

Write a code to compute $v(x)$ and plot both $v(x)$ and $f(x)$ in one figure, for $l = 5, 10$. Do the same thing for the function

$$g(x) = \begin{cases} 1, & -\pi/2 \leq x \leq \pi/2, \\ 0, & \text{otherwise.} \end{cases}$$

where $x \in [-\pi, \pi]$. What do you observe?

Here you can either use numerical integration package to solve $a_k, b_k$ approximately or derive the exact answer analytically for $a_k, b_k$.

```
# code
```

## Homework 11

**Problem K.1** (Ex. 1** [Fast Fourier Transform for Signal Processing)**.** ] This problem asks you to use 1D FFT to compress signals. Please read through the given code and answer the following questions.

(1) What is the signal considered here? Write down its mathematical formula.

(2) Describe how the Fourier coefficients, or the frequency spectrum, of the signal, look like.

(3) How does the code compress the signal, given the parameter "num_freq"?

(4) Modify the code to use 10 frequency components for compression. What do you observe compared to using 5 frequency components?

```
import numpy as np
import matplotlib.pyplot as plt

def generate_signal(t):
    """
    Generate a test signal with multiple frequency components and noise

    Args:
```

```
        t: Time array
    Returns:
        tuple: (clean_signal, noisy_signal)
    """
    # Generate pure signal components
    signal = (
        5 * np.sin(2 * np.pi * 10 * t) +  # 10 Hz
        2 * np.sin(2 * np.pi * 25 * t) +  # 25 Hz
        1 * np.sin(2 * np.pi * 50 * t)    # 50 Hz
    )

    # Add noise
    noise = 0.5 * np.random.randn(len(t))
    noisy_signal = signal + noise

    return signal, noisy_signal

def analyze_fft(signal, sample_rate):
    """
    Perform FFT analysis on the signal

    Args:
        signal: Input signal array
        sample_rate: Sampling rate in Hz
    Returns:
        tuple: (frequencies, magnitudes, fft_result)
    """
    # Calculate FFT
    fft_result = np.fft.fft(signal)
    freqs = np.fft.fftfreq(len(signal), 1/sample_rate)

    # Get positive frequencies
    pos_mask = freqs >= 0
    freqs = freqs[pos_mask]
    fft_mag = np.abs(fft_result)[pos_mask]

    return freqs, fft_mag, fft_result

def compress_signal(signal, n_components):
    """
    Compress signal by keeping only N largest FFT components

    Args:
        signal: Input signal array
```

```
        n_components: Number of frequency components to keep
    Returns:
        tuple: (compressed_signal, compression_ratio)
    """
    # Calculate FFT
    fft = np.fft.fft(signal)

    # Get magnitudes and indices of highest components
    magnitudes = np.abs(fft)
    sorted_indices = np.argsort(magnitudes)[::-1]

    # Create mask for top N components
    mask = np.zeros_like(fft, dtype=bool)
    mask[sorted_indices[:n_components]] = True

    # Apply mask and inverse FFT
    fft_compressed = fft * mask
    compressed = np.real(np.fft.ifft(fft_compressed))

    # Calculate compression ratio
    compression_ratio = len(signal) / (n_components * 2)  # *2 for complex numbers

    return compressed, compression_ratio

def plot_analysis(t, original, noisy, freqs, fft_mag):
    """Plot original signal and its frequency spectrum"""
    fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10, 8))

    # Time domain
    ax1.plot(t, original, 'b-', label='Clean Signal')
    ax1.plot(t, noisy, 'r-', alpha=0.5, label='Noisy Signal')
    ax1.set_xlabel('Time (s)')
    ax1.set_ylabel('Amplitude')
    ax1.set_title('Time Domain Signal')
    ax1.legend()
    ax1.grid(True)

    # Frequency domain
    ax2.plot(freqs, fft_mag)
    ax2.set_xlabel('Frequency (Hz)')
    ax2.set_ylabel('Magnitude')
    ax2.set_title('Frequency Spectrum')
    ax2.grid(True)
```

```
    plt.tight_layout()
    return fig

def plot_compression_comparison(t, original, compressed, n_freq):
    """Plot original vs compressed signal"""
    plt.figure(figsize=(10, 4))
    plt.plot(t, original, 'b-', label='Original')
    plt.plot(t, compressed, 'r--', label=f'Compressed ({n_freq} freq)')
    plt.xlabel('Time (s)')
    plt.ylabel('Amplitude')
    plt.legend()
    plt.grid(True)
    plt.tight_layout()


# Example usage

# Generate signal
duration = 1.0  # seconds
sample_rate = 1000  # Hz
t = np.linspace(0, duration, int(duration * sample_rate))

# Generate signals
clean_signal, noisy_signal = generate_signal(t)

# Analyze FFT
freqs, fft_mag, fft_result = analyze_fft(noisy_signal, sample_rate)
fig = plot_analysis(t, clean_signal, noisy_signal, freqs, fft_mag)
plt.show()

# Plot example compression
num_freq = 5
compressed, _ = compress_signal(noisy_signal, num_freq)
plot_compression_comparison(t, clean_signal, compressed, num_freq)
plt.show()
```

**Problem K.2** (Ex. 2** [Fast Fourier Transform for Image Processing). ] This problem asks you to use 2D FFT to compress signals. Please read through the given code, understand what the code is doing, and apply the code to another image you like. Comment on the output of the code.

```
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
```

```
def load_image(path):
    """Load an image and convert to grayscale."""
    img = Image.open(path).convert('L')
    return np.array(img)

def compress_image(image, threshold_percentage):
    """
    Compress image using FFT.

    Args:
        image: Input image as numpy array
        threshold_percentage: Percentage of coefficients to keep (0-100)

    Returns:
        tuple: (compressed_image, original_fft, compressed_fft, compression_ratio, psnr)
    """
    # Apply 2D FFT
    fft = np.fft.fft2(image)
    fft_shifted = np.fft.fftshift(fft)

    # Get magnitudes of FFT coefficients
    magnitudes = np.abs(fft_shifted)

    # Calculate threshold based on percentage
    threshold = np.percentile(magnitudes, 100 - threshold_percentage)

    # Create mask and apply threshold
    mask = magnitudes > threshold
    fft_compressed = fft_shifted * mask

    # Count non-zero coefficients for compression ratio
    original_size = image.size
    compressed_size = np.count_nonzero(mask)
    compression_ratio = original_size / compressed_size

    # Inverse FFT
    ifft_shifted = np.fft.ifftshift(fft_compressed)
    image_compressed = np.real(np.fft.ifft2(ifft_shifted))

    # Calculate PSNR
    mse = np.mean((image - image_compressed) ** 2)
    max_pixel = np.max(image)
```

```
    psnr = 20 * np.log10(max_pixel) - 10 * np.log10(mse)

    return image_compressed, fft_shifted, fft_compressed, compression_ratio, psnr

def plot_compression_results(original, compressed, fft_original, fft_compressed,
                             threshold_percentage, compression_ratio, psnr):
    """Plot original and compressed images with their FFT spectrums."""
    fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(12, 10))

    # Original image
    ax1.imshow(original, cmap='gray')
    ax1.set_title('Original Image')
    ax1.axis('off')

    # Compressed image
    ax2.imshow(compressed, cmap='gray')
    ax2.set_title(f'Compressed Image\nThreshold: {threshold_percentage}%\n'
                  f'Compression Ratio: {compression_ratio:.2f}x\nPSNR: {psnr:.2f} dB')
    ax2.axis('off')

    # Original FFT spectrum
    fft_log = np.log(np.abs(fft_original) + 1)
    ax3.imshow(fft_log, cmap='viridis')
    ax3.set_title('Original FFT Spectrum (log scale)')
    ax3.axis('off')

    # Compressed FFT spectrum
    fft_comp_log = np.log(np.abs(fft_compressed) + 1)
    ax4.imshow(fft_comp_log, cmap='viridis')
    ax4.set_title('Compressed FFT Spectrum (log scale)')
    ax4.axis('off')

    plt.tight_layout()
    return fig

def analyze_compression_levels(image, thresholds=[5, 10, 30]):
    """Analyze compression at different threshold levels."""
    results = []

    for threshold in thresholds:
        compressed, fft_orig, fft_comp, ratio, psnr = compress_image(image, threshold)
        results.append({
            'threshold': threshold,
            'compression_ratio': ratio,
```

```
            'psnr': psnr,
            'compressed_image': compressed,
            'fft_original': fft_orig,
            'fft_compressed': fft_comp
        })

        # Plot results for this threshold
        fig = plot_compression_results(image, compressed, fft_orig, fft_comp,
                                       threshold, ratio, psnr)
        plt.show()
        plt.close(fig)

    return results

# Example usage
# Load image
image = load_image('NYUpicture.jpg')

print("Analyzing compression at different threshold levels...")
results = analyze_compression_levels(image)

# Print summary
print("\nCompression Summary:")
print("Threshold | Compression Ratio | PSNR (dB)")
print("-" * 45)
for r in results:
    print(f"{r['threshold']:9d} | {r['compression_ratio']:16.2f} | {r['psnr']:8.2f}")
```

**Problem K.3** (the class, we studied the following two numerical differentiation formula, Ex. 3** [Numerical Differentiation)**.** ] for three points $x_{-1}$, $x_0 = x_{-1} + h_0$, and $x_1 = x_0 + h_1$

$$f'(x_0) \approx \frac{f(x_1) - f(x_{-1})}{h_0 + h_1}$$

and

$$f'(x_0) \approx \frac{h_1 - h_0}{h_0 h_1} f(x_0) + \frac{1}{h_0 + h_1} \left( \frac{h_0}{h_1} f(x_1) - \frac{h_1}{h_0} f(x_{-1}) \right)$$

Here $h_0 \neq h_1$ and the second formula is derived using Lagrange interpolation.

Show, using Taylor's expansion, that the first formula leads to error $O(h)$ while the second leads to error $O(h^2)$, which match the numerical experiments presented in the class.

**Problem 4** [Numerical Integration] Compare the numerical integration of $f(x) = 5e^x$ over the interval [0,1] using the composite trapezoidal rule and Gaussian quadrature

method. Implement both methods in Python and create a convergence plot showing error versus number of points used (from 2 points to 5 points).

What is the convergence rate of the composite trapezoidal rule (written in the form of $O(h^r)$. You need to identify $r$). Here you can calculate the truth of the integral analytically and compare your numerical solution with it to obtain the accuracy.

Implement the methods also for $f(x) = \sqrt{x}$

```
# code
# Hint: you can get Gauss-Legendre quadrature points and weights in [-1,1]
# through x, w = np.polynomial.legendre.leggauss(n)
```

## Homework 12

**Problem L.1. Problem 1** Solve the ODE $\frac{dy}{dt} = 5y$ for $t \in [0, 1]$ and $y(0) = 1$ using forward Euler and backward Euler respectively. The true solution is $y(t) = \exp(5t)$.

Plot figures of the numerical solutions and the true solution. Using different stepsizes to solve the problem. Plot figures showing the convergence of the errors at $t = 1$ as stepsize becomes smaller. What is the convergence rate?

Suggestions: use stepsize smaller than 0.01 to help visualize convergence.

**Problem 2** Consider the ODE $\frac{dy}{dt} = f(t, y)$ and the numerical method

$$\frac{y_{i+1} - y_i}{h} = \frac{1}{2}(f(t_i, y_i) + f(t_{i+1}, y_{i+1}))$$

What is the order of accuracy of this method?

Then, suppose $f(t, y) = \lambda y$. Discuss the absolute stability property of the method for the cases $\lambda > 0$ and $\lambda < 0$