CS111 - Scientific Computing Basics

Spring 2025

UCSB

Contents

1 Fundamentals of Linear Algebra

1.1 Vectors and Matrices

$\mathbb{R}$ The set of all real numbers. Real numbers form the backbone of most numerical computations and serve as the primary field in many applications across science and engineering.

$\mathbb{Z}$ The set of all integers. Integers are important for problems involving countable or discrete quantities and are a subset of the real numbers.

$\mathbb{C}$ The set of all complex numbers. Although our primary focus is on real scalars, complex numbers extend the idea of numbers and are essential in advanced topics such as signal processing, control theory, and quantum mechanics.

Scalars Elements of number spaces (e.g., $\mathbb{R}$, $\mathbb{C}$, $\mathbb{Z}$) are called **scalars**. They represent individual quantities and serve as the fundamental building blocks from which vectors and matrices are constructed.

Scalars can be viewed as the atoms of numerical computation. In advanced studies, such as functional analysis or numerical linear algebra, understanding the properties of scalars is crucial, particularly when extending these concepts to infinite-dimensional spaces or complex systems.

Vectors A **vector** is a one-dimensional array of scalars. Vectors can represent points, directions, or more abstract quantities within a vector space, and they are fundamental in representing data and linear relationships.

NOTE Vectors may be represented either as row vectors or column vectors. Although both forms contain the same information, the distinction becomes important in the context of linear transformations, dual spaces, and inner product spaces.

Examples

$$\begin{pmatrix} 1 & 2 & 3 \end{pmatrix}, \quad \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}, \quad \begin{pmatrix} 1+2i \\ 7i \\ 3 \end{pmatrix}, \ldots$$

Just as scalars belong to number systems like $\mathbb{R}$ or $\mathbb{C}$, vectors inhabit **vector spaces**. While these notes treat vector spaces in an informal manner, later chapters will provide a rigorous formulation of vector spaces and their properties.

$\mathbb{R}^n$ The set of all real vectors of size n. When we write $x \in \mathbb{R}^n$, it indicates that x is a vector with n real entries. Vectors in $\mathbb{R}^n$ are typically expressed in one of the

following forms:

$$\boldsymbol{x} \in \mathbb{R}^{n\times 1} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \quad \text{(column vector)}$$

$$\boldsymbol{x} \in \mathbb{R}^{1\times n} = \begin{pmatrix} x_1 & x_2 & \cdots & x_n \end{pmatrix}, \quad \text{(row vector)}$$

For these notes, the column vector form is adopted as the default representation.

NOTE In our discussion, we focus on finite-dimensional vector spaces. In more advanced topics, infinite-dimensional vector spaces—central to functional analysis—play a significant role in areas such as differential equations and quantum mechanics.

Matrices A **matrix** is a two-dimensional (or higher-dimensional) array of scalars. A matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ is typically written as

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}.$$

Matrices are used to represent linear transformations, systems of linear equations, and more complex data structures.

Examples Examples of two-dimensional matrices include:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}, \quad \begin{pmatrix} 1 & 2 \\ 4 & 5 \\ 6 & 7 \end{pmatrix}, \quad \text{etc.}$$

NOTE Matrices can also be interpreted as vectors whose entries are themselves vectors. For instance,

$$\mathbf{A} = \begin{pmatrix} \text{—} & \boldsymbol{r}_1 & \text{—} \\ & \vdots & \\ \text{—} & \boldsymbol{r}_n & \text{—} \end{pmatrix} = \begin{pmatrix} | & & | \\ \boldsymbol{c}_1 & \cdots & \boldsymbol{c}_n \\ | & & | \end{pmatrix},$$

where:

1. $r_i = \mathbf{A}[i]$ denotes the *row vector* corresponding to the i^{th} row of $\mathbf{A}$.
2. $c_i = \mathbf{A}[:, i]$ denotes the *column vector* corresponding to the i^{th} column of $\mathbf{A}$.

This viewpoint is particularly useful when exploring advanced topics such as block matrices, tensor decompositions, and matrix factorization methods.

1.2 Matrix Arithmetic

We are all familiar with scalar arithmetic, including operations like addition, subtraction, and multiplication, as well as fundamental properties such as associativity and distributivity. In the following sections, we extend these familiar concepts to vectors and matrices, exploring how these operations are defined and applied in higher dimensions.

1.2.1 Addition and Subtraction

Matrix addition is performed element-wise. Note that matrices can only be added (or subtracted) if they have the same dimensions.

Matrix Sum Let $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{m\times n}$ be matrices. Their sum is defined as

$$\mathbf{A}+\mathbf{B} = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix} + \begin{pmatrix} b_{11} & \cdots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{nn} \end{pmatrix} = \begin{pmatrix} (a+b)_{11} & \cdots & (a+b)_{1n} \\ \vdots & \ddots & \vdots \\ (a+b)_{n1} & \cdots & (a+b)_{nn} \end{pmatrix},$$

where each entry satisfies $(a+b)_{ij} = a_{ij} + b_{ij}$ for $1 \le i \le m$ and $1 \le j \le n$.

NOTE For two matrices $\mathbf{A} \in \mathbb{R}^{m\times n}$ and $\mathbf{B} \in \mathbb{R}^{p\times q}$, the sum $\mathbf{A}+\mathbf{B}$ is defined if and only if $m = p$ and $n = q$.

Commutativity and Associativity of Matrix Addition

Let

$$\mathbf{A} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \quad \text{and} \quad \mathbf{B} = \begin{pmatrix} e & f \\ g & h \end{pmatrix}.$$

Then,

$$\mathbf{A}+\mathbf{B} = \begin{pmatrix} a+e & b+f \\ c+g & d+h \end{pmatrix} \quad \text{and} \quad \mathbf{B}+\mathbf{A} = \begin{pmatrix} e+a & f+b \\ g+c & h+d \end{pmatrix}.$$

Since addition of real numbers is commutative, we have $\mathbf{A}+\mathbf{B} = \mathbf{B}+\mathbf{A}$. Furthermore, for any third matrix $\mathbf{C} = \begin{pmatrix} i & j \\ k & l \end{pmatrix}$,

$$(\mathbf{A}+\mathbf{B})+\mathbf{C} = \begin{pmatrix} (a+e)+i & (b+f)+j \\ (c+g)+k & (d+h)+l \end{pmatrix},$$

and

$$\mathbf{A}+(\mathbf{B}+\mathbf{C}) = \begin{pmatrix} a+(e+i) & b+(f+j) \\ c+(g+k) & d+(h+l) \end{pmatrix}.$$

By associativity of addition for real numbers, these two results are identical. Hence, matrix addition is both commutative and associative.

Scalar Multiplication

Let $\mathbf{A} \in \mathbb{R}^{m\times n}$ and $c \in \mathbb{R}$. Then scalar multiplication is defined as

$$c\mathbf{A} = c\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix} = \begin{pmatrix} ca_{11} & \cdots & ca_{1n} \\ \vdots & \ddots & \vdots \\ ca_{n1} & \cdots & ca_{nn} \end{pmatrix},$$

where each entry satisfies $(ca)_{ij} = c \cdot a_{ij}$.

Distributivity of Scalar Multiplication over Matrix Addition

Let $\alpha \in \mathbb{R}$ and

$$\mathbf{A} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} e & f \\ g & h \end{pmatrix}.$$

Then,

$$\alpha(\mathbf{A}+\mathbf{B}) = \alpha\begin{pmatrix} a+e & b+f \\ c+g & d+h \end{pmatrix} = \begin{pmatrix} \alpha(a+e) & \alpha(b+f) \\ \alpha(c+g) & \alpha(d+h) \end{pmatrix}.$$

Since scalar multiplication distributes over addition for real numbers, this equals

$$\begin{pmatrix} \alpha a+\alpha e & \alpha b+\alpha f \\ \alpha c+\alpha g & \alpha d+\alpha h \end{pmatrix} = \alpha\mathbf{A}+\alpha\mathbf{B}.$$

1.2.2 Dot Product

Dot Product

For vectors $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^n$, the dot product (or inner product) is defined as

$$\boldsymbol{a}\cdot\boldsymbol{b} = \boldsymbol{a}^t\boldsymbol{b} = \begin{pmatrix} a_1 & \cdots & a_n \end{pmatrix}\begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix} = \sum_{i=1}^{n} a_i b_i. \tag{1}$$

Matrix Product

Let $\mathbf{A} \in \mathbb{R}^{m\times n}$ and $\mathbf{B} \in \mathbb{R}^{n\times p}$. Their product $\mathbf{AB}$ is an $m \times p$ matrix defined by

$$(\mathbf{AB})_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj},$$

or equivalently, if we consider the rows of $\mathbf{A}$ as vectors $\boldsymbol{a}_i$ and the columns of $\mathbf{B}$ as vectors $\boldsymbol{b}_j$, then

$$\mathbf{AB} = \begin{pmatrix} \boldsymbol{a}_1\cdot\boldsymbol{b}_1 & \cdots & \boldsymbol{a}_1\cdot\boldsymbol{b}_p \\ \vdots & \ddots & \vdots \\ \boldsymbol{a}_m\cdot\boldsymbol{b}_1 & \cdots & \boldsymbol{a}_m\cdot\boldsymbol{b}_p \end{pmatrix}.$$

NOTE

Matrix multiplication is defined only when the number of columns of $\mathbf{A}$ equals the number of rows of $\mathbf{B}$. Moreover, it is associative and distributive over addition, but in general it is not commutative; that is, $\mathbf{AB} \neq \mathbf{BA}$ for most matrices.

Compatibility for Matrix Multiplication

Let $\mathbf{A}$ be a 3×4 matrix and $\mathbf{B}$ be a 4×2 matrix. Since the number of columns in $\mathbf{A}$ (4) equals the number of rows in $\mathbf{B}$ (4), the product $\mathbf{AB}$ is defined and will result in a 3×2 matrix. Conversely, if $\mathbf{B}$ were instead a 5×2 matrix, the product would be undefined.

Distributive Property of Matrix Multiplication

Let

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}, \quad \mathbf{C} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}.$$

Then,

$$\mathbf{A}(\mathbf{B} + \mathbf{C}) = \mathbf{A} \begin{pmatrix} b_{11} + c_{11} & b_{12} + c_{12} \\ b_{21} + c_{21} & b_{22} + c_{22} \end{pmatrix},$$

and by computing the entries using the definition of matrix multiplication, one can verify that

$$\mathbf{A}(\mathbf{B} + \mathbf{C}) = \mathbf{AB} + \mathbf{AC}.$$

Non-Commutativity of Matrix Multiplication

Consider the matrices

$$\mathbf{A} = \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix} \quad \text{and} \quad \mathbf{B} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

Then,

$$\mathbf{AB} = \begin{pmatrix} 1 \cdot 0 + 0 \cdot 1 & 1 \cdot 1 + 0 \cdot 0 \\ 0 \cdot 0 + 2 \cdot 1 & 0 \cdot 1 + 2 \cdot 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 0 \end{pmatrix},$$

and

$$\mathbf{BA} = \begin{pmatrix} 0 \cdot 1 + 1 \cdot 0 & 0 \cdot 0 + 1 \cdot 2 \\ 1 \cdot 1 + 0 \cdot 0 & 1 \cdot 0 + 0 \cdot 2 \end{pmatrix} = \begin{pmatrix} 0 & 2 \\ 1 & 0 \end{pmatrix}.$$

Since $\mathbf{AB} \neq \mathbf{BA}$ and clearly $\mathbf{A} \neq \mathbf{B}$, this example shows that matrix multiplication is not commutative.
Many properties of scalar arithmetic extend naturally to both vectors and matrices. In fact, matrices can be viewed as elements of a higher-dimensional vector space, and matrix multiplication can be interpreted as the composition of linear transformations. While the notations for vectors and matrices are similar, their roles in linear algebra differ:

- Vectors typically represent points or directions in space.
- Matrices often represent linear maps or transformations between vector spaces.

This conceptual distinction becomes crucial when discussing topics such as vector spaces, eigenvalues, and linear transformations.

1.2.3 Matrix Inverse

Matrix Inverse

For a square matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$, the inverse of $\mathbf{A}$, denoted $\mathbf{A}^{-1}$, is defined as the unique matrix satisfying

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I},$$

where $\mathbf{I}$ is the $n \times n$ identity matrix. A matrix that has an inverse is called *invertible* or *nonsingular*.

NOTE

A square matrix $\mathbf{A}$ is invertible if and only if $\det(\mathbf{A}) \neq 0$, which also implies that $\mathbf{A}$ has full rank.

Uniqueness of the Inverse

Suppose $\mathbf{A}$ is an invertible matrix and that $\mathbf{B}$ and $\mathbf{C}$ are both inverses of $\mathbf{A}$; that is,

$$\mathbf{AB} = \mathbf{BA} = \mathbf{I} \quad \text{and} \quad \mathbf{AC} = \mathbf{CA} = \mathbf{I}.$$

Then,

$$\mathbf{B} = \mathbf{BI} = \mathbf{B}(\mathbf{AC}) = (\mathbf{BA})\mathbf{C} = \mathbf{IC} = \mathbf{C}.$$

Thus, the inverse of $\mathbf{A}$ is unique.

2 Linear Systems of Equations

2.1 Linearity

Linearity is a fundamental concept in numerical linear algebra that underpins many of the algorithms and methods discussed in this course. At its core, linearity describes mathematical structures that behave in a predictable way under addition and scalar multiplication.

A mapping or operation L is considered linear if it satisfies two key properties:

- **Additivity**: $L(\mathbf{x} + \mathbf{y}) = L(\mathbf{x}) + L(\mathbf{y})$ for all inputs $\mathbf{x}$ and $\mathbf{y}$
- **Homogeneity**: $L(\alpha\mathbf{x}) = \alpha L(\mathbf{x})$ for all inputs $\mathbf{x}$ and all scalars α

These properties can be combined into the principle of superposition: $L(\alpha\mathbf{x} + \beta\mathbf{y}) = \alpha L(\mathbf{x}) + \beta L(\mathbf{y})$ for all inputs $\mathbf{x}$ and $\mathbf{y}$, and all scalars α and β.

Linear mappings preserve vector operations, which allows us to decompose complex problems into simpler ones and then combine the solutions. This property is what makes linear algebra so powerful for a wide range of applications, from solving systems of equations to approximating functions, analyzing data, and modeling physical phenomena.

In numerical linear algebra, we often represent linear mappings as matrices. The multiplication of a matrix $\mathbf{A}$ by a vector $\mathbf{x}$ is a linear operation, making matrices the perfect tool for expressing and analyzing linear transformations. The computational challenges that arise in numerical linear algebra—such as solving linear systems, finding eigenvalues, or computing matrix decompositions—all fundamentally involve manipulating these linear structures efficiently and accurately.

2.1.1 Linear Systems

Linear Systems

A **system of linear equations** is a collection of equations in which each equation is a linear combination of the unknown variables. Such a system can be written in the form

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \quad &\cdots \quad + a_{1n}x_n = b_1, \\ a_{21}x_1 + a_{22}x_2 + \quad &\cdots \quad + a_{2n}x_n = b_2, \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \quad &\cdots \quad + a_{mn}x_n = b_m, \end{aligned}$$

where each a_{ij} (with $1 \leq i \leq m$ and $1 \leq j \leq n$) is a known constant and $x_1, \ldots, x_n$ are the unknowns. In matrix-vector form, this system is succinctly expressed as

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

where

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, \quad \boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \quad \boldsymbol{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}.$$

The mapping $T : \mathbb{R}^n \to \mathbb{R}^m$ defined by $T(\boldsymbol{x}) = \mathbf{A}\boldsymbol{x}$ is a linear transformation. That is, for any vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$ and scalar $c \in \mathbb{R}$, we have:

$$T(\boldsymbol{x} + \boldsymbol{y}) = \mathbf{A}(\boldsymbol{x} + \boldsymbol{y}) = \mathbf{A}\boldsymbol{x} + \mathbf{A}\boldsymbol{y} = T(\boldsymbol{x}) + T(\boldsymbol{y}),$$

$$T(c\boldsymbol{x}) = \mathbf{A}(c\boldsymbol{x}) = c\,\mathbf{A}\boldsymbol{x} = c\,T(\boldsymbol{x}).$$

These properties, known as additivity and homogeneity, are the defining features of linear maps.

Existence and Uniqueness

For a square system (i.e., when $m = n$), the linear system

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

has a unique solution if and only if the coefficient matrix $\mathbf{A}$ is invertible, which is equivalent to $\det(\mathbf{A}) \neq 0$. If $\mathbf{A}$ is singular (i.e., noninvertible, $\det(\mathbf{A}) = 0$), then the system either has no solution or has infinitely many solutions. In the latter case, the set of solutions forms an affine subspace of $\mathbb{R}^n$.

Linear Combination

A **linear combination** of vectors $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$ is any vector $\boldsymbol{y}$ that can be written as

$$\boldsymbol{y} = \alpha_1 \boldsymbol{x}_1 + \alpha_2 \boldsymbol{x}_2 + \cdots + \alpha_n \boldsymbol{x}_n,$$

where $\alpha_1, \ldots, \alpha_n \in \mathbb{R}$.

NOTE View each vector as an ingredient; a linear combination mixes these ingredients in various proportions to produce new vectors.

Linear Dependence

A set of vectors $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$ is **linearly dependent** if there exist scalars $\alpha_1, \ldots, \alpha_n$, not all zero, such that

$$\alpha_1 \boldsymbol{x}_1 + \alpha_2 \boldsymbol{x}_2 + \cdots + \alpha_n \boldsymbol{x}_n = \mathbf{0}.$$

If the only solution is $\alpha_1 = \cdots = \alpha_n = 0$, the vectors are **linearly independent**.

NOTE Linear dependence indicates redundancy; one or more vectors in the set can be expressed as a combination of the others.

Basis

A **basis** for a vector space $\mathcal{V}$ is a set of vectors that is both linearly independent and spanning. Every vector in $\mathcal{V}$ can be uniquely expressed as a linear combination of the basis vectors. The number of basis vectors is the **dimension** of $\mathcal{V}$.

2.1.2 Rank

Rank

The **rank** of a matrix $\mathbf{A}$ is defined as the maximum number of linearly independent rows (or equivalently, columns) of $\mathbf{A}$. Equivalently, it is the dimension of the row space (or column space) of $\mathbf{A}$. For an $m \times n$ matrix, the rank is always less than or equal to $\min(m, n)$.

NOTE Intuitively, the rank measures the amount of "information" contained in the matrix. It indicates the maximum number of independent directions or components that the matrix can represent. For instance, if a matrix has a rank of r, then there are r rows (or columns) that contribute unique information, while the remaining rows (or columns) can be expressed as linear combinations of these.

NOTE

For an $m \times n$ matrix, since there are m rows and n columns, the rank cannot exceed either m or n. In particular, if $\mathbf{A}$ is square (i.e., $m = n$) and full rank (i.e., rank $= n$), then $\mathbf{A}$ is invertible.

Full Rank

A matrix is said to be **full rank** if its rank is as large as possible, that is, if $\text{rank}(\mathbf{A}) = \min(m, n)$. In the case of a square matrix, full rank implies invertibility.

Examples

Example: Determining the Rank of a Matrix

Consider the matrix

$$\mathbf{A} = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 1 & 1 & 1 \end{pmatrix}.$$

Observe that the second row is exactly twice the first row, suggesting a dependency. To determine the rank, we perform elementary row operations to obtain the row-echelon form:

Step 1: Leave the first row unchanged:

$$R_1 = (1, 2, 3).$$

Step 2: Replace the second row by subtracting $2R_1$:

$$R_2 \to R_2 - 2R_1 : \quad (2, 4, 6) - 2(1, 2, 3) = (0, 0, 0).$$

Step 3: Replace the third row by subtracting R_1:

$$R_3 \to R_3 - R_1 : \quad (1, 1, 1) - (1, 2, 3) = (0, -1, -2).$$

The matrix becomes:

$$\mathbf{A}' = \begin{pmatrix} 1 & 2 & 3 \\ 0 & 0 & 0 \\ 0 & -1 & -2 \end{pmatrix}.$$

Swap R_2 and R_3 to bring the nonzero row up:

$$\mathbf{A}'' = \begin{pmatrix} 1 & 2 & 3 \\ 0 & -1 & -2 \\ 0 & 0 & 0 \end{pmatrix}.$$

Finally, multiply the second row by -1:

$$\mathbf{A}''' = \begin{pmatrix} 1 & 2 & 3 \\ 0 & 1 & 2 \\ 0 & 0 & 0 \end{pmatrix}.$$

Since there are two nonzero rows (pivot rows), the rank of $\mathbf{A}$ is 2.

Rank-Nullity Theorem

For any $m \times n$ matrix $\mathbf{A}$,

$$\text{rank}(\mathbf{A}) + \dim(\mathbf{A}) = n,$$

where the nullity of $\mathbf{A}$ is the dimension of its null space. This relationship is fundamental in understanding the solution structure of the homogeneous system $\mathbf{A}\boldsymbol{x} = \mathbf{0}$.

NOTE In many applications such as data compression, signal processing, and machine learning, the rank of a matrix is used to identify redundancy in data. Low-rank approximations can help reduce dimensionality while preserving essential information.

Linear Span The **linear span** of a set of vectors $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$, denoted $\text{span}\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$, is the set of all linear combinations of these vectors:

$$\text{span}\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\} = \{\alpha_1\boldsymbol{x}_1 + \cdots + \alpha_n\boldsymbol{x}_n : \alpha_i \in \mathbb{R}\}.$$

This is the smallest subspace of the vector space that contains the given vectors.

Examples **Basis for span$\{(1,2),(3,4)\}$ in $\mathbb{R}^2$**

Consider the vectors $\boldsymbol{v}_1 = (1,2)$ and $\boldsymbol{v}_2 = (3,4)$. To check their linear independence, assume there exist scalars α and β such that

$$\alpha(1,2) + \beta(3,4) = (0,0).$$

This yields the system:

$$\alpha + 3\beta = 0, \quad 2\alpha + 4\beta = 0.$$

Multiplying the first equation by 2 gives $2\alpha + 6\beta = 0$. Subtracting the second equation, we have

$$(2\alpha + 6\beta) - (2\alpha + 4\beta) = 2\beta = 0 \quad \Rightarrow \quad \beta = 0.$$

Substituting back, $\alpha = 0$. Since the only solution is $\alpha = \beta = 0$, the vectors are linearly independent. Hence, $\{(1,2),(3,4)\}$ forms a basis for span$\{(1,2),(3,4)\}$, which in this case is all of $\mathbb{R}^2$.

2.2 LU Factorization

The LU factorization expresses a square matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ as the product

$$\mathbf{A} = \mathbf{L}\,\mathbf{U},$$

where $\mathbf{L}$ is a lower triangular matrix with ones on the diagonal (a unit lower triangular matrix) and $\mathbf{U}$ is an upper triangular matrix. This factorization simplifies solving linear systems $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ by allowing the solution to be obtained via forward substitution (for $\mathbf{L}\boldsymbol{y} = \boldsymbol{b}$) followed by backward substitution (for $\mathbf{U}\boldsymbol{x} = \boldsymbol{y}$).

2.2.1 LU Factoriazation and Gaussian Elimination

LU Factorization

An **LU Factorization** of a matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ is a decomposition of the form

$$\mathbf{A} = \mathbf{L}\,\mathbf{U},$$

with

$$\mathbf{L} = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ \ell_{21} & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ \ell_{n1} & \ell_{n2} & \cdots & 1 \end{pmatrix} \quad \text{and} \quad \mathbf{U} = \begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{nn} \end{pmatrix}.$$

Gaussian elimination

The procedure for LU factorization closely follows Gaussian elimination. For example, given

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix},$$

one begins by eliminating the entries below a_{11}. For $i = 2, \ldots, n$, set

$$\ell_{i1} = \frac{a_{i1}}{a_{11}},$$

and update row i by replacing it with

$$\text{row}_i \leftarrow \text{row}_i - \ell_{i1}\,\text{row}_1.$$

This results in zeros below a_{11} and new entries

$$a_{ij}^{(1)} = a_{ij} - \ell_{i1}\,a_{1j}, \quad j = 2, \ldots, n.$$

Repeating this process on the $(n-1)\times(n-1)$ submatrix produces the full LU decomposition.

Examples

To illustrate, consider the 3×3 matrix

$$\mathbf{A} = \begin{pmatrix} 2 & 3 & 1 \\ 4 & 7 & 3 \\ -2 & -3 & 1 \end{pmatrix}.$$

We wish to decompose $\mathbf{A}$ as $\mathbf{A} = \mathbf{L}\,\mathbf{U}$ with

$$\mathbf{L} = \begin{pmatrix} 1 & 0 & 0 \\ \ell_{21} & 1 & 0 \\ \ell_{31} & \ell_{32} & 1 \end{pmatrix}, \quad \mathbf{U} = \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{pmatrix}.$$

Begin by taking the first row of $\mathbf{A}$ as the first row of $\mathbf{U}$:

$$u_{11} = 2, \quad u_{12} = 3, \quad u_{13} = 1.$$

The multipliers for the first column are then

$$\ell_{21} = \frac{4}{2} = 2, \quad \ell_{31} = \frac{-2}{2} = -1.$$

After eliminating the first column, update the remaining rows. For row 2:

$$u_{22} = 7 - 2 \cdot 3 = 1, \quad u_{23} = 3 - 2 \cdot 1 = 1.$$

For row 3, the updated entries become

$$\tilde{a}_{32} = -3 - (-1) \cdot 3 = 0, \quad \tilde{a}_{33} = 1 - (-1) \cdot 1 = 2.$$

Thus, $\ell_{32} = 0$ and $u_{33} = 2$. Consequently, we have

$$\mathbf{L} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & 0 & 1 \end{pmatrix}, \quad \mathbf{U} = \begin{pmatrix} 2 & 3 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix}.$$

A direct multiplication verifies that $\mathbf{L}\,\mathbf{U} = \mathbf{A}$.

2.2.2 Pivoting and LUP Decomposition

Pivoting and LUP Decomposition

When the matrix $\mathbf{A}$ has zeros or small entries in prospective pivot positions, it is necessary to perform row interchanges to maintain numerical stability. These interchanges are recorded by a permutation matrix $\mathbf{P}$, leading to the factorization

$$\mathbf{PA} = \mathbf{L}\,\mathbf{U},$$

which is known as the *LUP Decomposition*. Here, $\mathbf{L}$ is a unit lower triangular matrix and $\mathbf{U}$ is an upper triangular matrix.

Partial pivoting selects the largest available pivot (in absolute value) by interchanging (permuting) rows. This minimizes the risk of division by a small number and reduces the growth of rounding errors. Although complete pivoting, which interchanges both rows and columns, can further improve stability, it is generally more computationally expensive.

The permutation matrix $\mathbf{P}$ is orthogonal in the sense that $\mathbf{P}^T\mathbf{P} = \mathbf{I}$. In fact, for any permutation matrix $\mathbf{P}$, one has $\mathbf{P}^{-1} = \mathbf{P}^T$, reflecting that $\mathbf{P}$ merely rearranges the rows of the identity matrix.

Examples

For example, consider the matrix

$$\mathbf{A} = \begin{pmatrix} 0 & 2 & 1 \\ 2 & 1 & 0 \\ 1 & 0 & 2 \end{pmatrix}.$$

Since the (1,1) entry is zero, we must permute the rows to secure a nonzero (and preferably large) pivot in the (1,1) position. Observing the first column, the largest absolute value is 2 (in row 2). Therefore, we swap row 1 with row 2 using the permutation matrix

$$\mathbf{P} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Then,

$$\mathbf{PA} = \begin{pmatrix} 2 & 1 & 0 \\ 0 & 2 & 1 \\ 1 & 0 & 2 \end{pmatrix}.$$

Next, we perform the LU factorization on $\mathbf{PA}$. Write

$$\mathbf{PA} = \mathbf{L}\,\mathbf{U},$$

with

$$\mathbf{L} = \begin{pmatrix} 1 & 0 & 0 \\ \ell_{21} & 1 & 0 \\ \ell_{31} & \ell_{32} & 1 \end{pmatrix} \quad \text{and} \quad \mathbf{U} = \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{pmatrix}.$$

Assign the first row of $\mathbf{U}$ from $\mathbf{PA}$:

$$u_{11} = 2, \quad u_{12} = 1, \quad u_{13} = 0.$$

Then, the multipliers for the first column are

$$\ell_{21} = \frac{0}{2} = 0, \quad \ell_{31} = \frac{1}{2} = 0.5.$$

Subtracting 0.5 times the first row from the third row updates it to

$$\begin{pmatrix} 1 & 0 & 2 \end{pmatrix} - 0.5 \begin{pmatrix} 2 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & -0.5 & 2 \end{pmatrix}.$$

For the 2×2 submatrix (rows 2 and 3), set

$$u_{22} = 2, \quad u_{23} = 1,$$

and compute the multiplier

$$\ell_{32} = \frac{-0.5}{2} = -0.25.$$

Finally, update the (3,3) entry:

$$u_{33} = 2 - (-0.25)(1) = 2.25.$$

Thus, we obtain

$$\mathbf{L} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0.5 & -0.25 & 1 \end{pmatrix}, \quad \mathbf{U} = \begin{pmatrix} 2 & 1 & 0 \\ 0 & 2 & 1 \\ 0 & 0 & 2.25 \end{pmatrix}.$$

A multiplication of $\mathbf{L}$ and $\mathbf{U}$ confirms that $\mathbf{L}\,\mathbf{U} = \mathbf{PA}$.

2.2.3 Solving Linear Systems using LU Factorization

Once a square matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ has been factorized as $\mathbf{A} = \mathbf{LU}$, solving the linear system $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ reduces to two simpler steps: forward substitution and backward substitution.

LU Solve Given the LU factorization, we solve the system $\mathbf{A}\boldsymbol{x} = \mathbf{LU}\boldsymbol{x} = \boldsymbol{b}$ by first solving

$$\mathbf{L}\boldsymbol{y} = \boldsymbol{b},$$

and then solving

$$\mathbf{U}\boldsymbol{x} = \boldsymbol{y}.$$

Since $\mathbf{L}$ is a unit lower triangular matrix, we use forward substitution. Specifically, for $i = 1, 2, \ldots, n$,

$$y_i = b_i - \sum_{j=1}^{i-1} \ell_{ij}\, y_j.$$

After obtaining $\boldsymbol{y}$, we solve the upper triangular system using backward substitution. For $i = n, n-1, \ldots, 1$,

$$x_i = \frac{1}{u_{ii}} \left(y_i - \sum_{j=i+1}^{n} u_{ij}\, x_j \right).$$

NOTE The forward substitution process requires roughly

$$\sum_{i=1}^{n} (i-1) = \frac{n(n-1)}{2} = O(n^2)$$

operations, and backward substitution has a similar computational cost. Thus, once the LU factorization is available, solving $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ takes $O(n^2)$ operations.

NOTE

It is important to note that while solving the system after the factorization is efficient, computing the LU factorization itself generally requires $O(n^3)$ operations. Therefore, if multiple right-hand sides $\boldsymbol{b}$ are to be solved with the same matrix $\mathbf{A}$, it is advantageous to factorize $\mathbf{A}$ once and reuse the factors $\mathbf{L}$ and $\mathbf{U}$ for each subsequent system (very common in iterative methods), reducing the per-system cost to $O(n^2)$.

2.3 Floating Point Numbers and Errors

Computing with real numbers on finite machines introduces inherent approximation errors. Understanding these approximations and their consequences is fundamental to scientific computing. In this section, we exmplore floating point representation, the types of errors that arise in numerical computations, and techniques for analyzing and managing these errors.

Floating Point Representation

Floating point numbers are represented in computers using a format similar to scientific notation. A normalized floating point number has the form

$$x = (-1)^s \times (1.f) \times 2^e,$$

where:

- s is the sign bit (0 for positive, 1 for negative)
- f is the fractional part (or significand/mantissa)
- e is the exponent
- The leading 1 before the decimal point is implicit in most formats (hidden bit)

The IEEE 754 standard defines several formats, including:

- **Single precision (32 bits)**: 1 sign bit, 8 exponent bits, 23 fraction bits
- **Double precision (64 bits)**: 1 sign bit, 11 exponent bits, 52 fraction bits

The exponent field uses a biased representation, with bias $= 2^{(k-1)} - 1$ where k is the number of exponent bits. This allows representation of both very small and very large numbers.

Examples

16-bit Floating Point Representation: et's examine a simplified 16-bit floating point format with 1 sign bit, 5 exponent bits, and 10 fraction bits. The bias for the exponent would be $2^{(5-1)} - 1 = 15$. To represent the number -3.25 in this format:

1. Convert to sign-magnitude form: sign = 1 (negative)

2. Convert to binary: $3.25_{10} = 11.01_2$
3. Normalize: $1.101_2 \times 2^1$
4. Determine the biased exponent: $1 + 15 = 16 = 10000_2$
5. Encode the significand: 1.101_2 gives fraction bits 1010000000
6. Final representation: 1 10000 1010000000

To count the total number of representable values in this format:

- Special values: ± 0, $\pm\infty$, and NaN configurations
- Normal numbers: $2 \times (2^5 - 2) \times 2^{10} = 2 \times 30 \times 1024 = 61,440$
- Subnormal numbers: $2 \times 2^{10} = 2048$

The total is approximately $63,488+$ NaN configurations.

Machine Epsilon ($\varepsilon_{\text{mach}}$)

Machine epsilon is a fundamental constant in floating point arithmetic, defined as the difference between 1 and the next representable floating point number. Equivalently, it is the smallest ε such that $1 + \varepsilon > 1$ in floating point arithmetic. For a floating point system with p bits in the significand (including the hidden bit), machine epsilon is:

$$\varepsilon_{\text{mach}} = 2^{-p+1}$$

Machine epsilon determines the relative precision of the floating point system. For any normalized floating point number x, the gap to the next representable number is approximately $\varepsilon_{\text{mach}} \cdot |x|$.

Examples

Calculating Machine Epsilon: For IEEE double precision (64 bits):

- 52 fraction bits + 1 hidden bit = 53 bits of precision
- $\varepsilon_{\text{mach}} = 2^{-53+1} = 2^{-52} \approx 2.22 \times 10^{-16}$

Try verifying this experimentally!

Unit Roundoff (u)

The unit roundoff u is half of machine epsilon:

$$u = \frac{\varepsilon_{\text{mach}}}{2} = 2^{-p}$$

Unit roundoff represents the maximum relative error introduced when rounding a real number to its nearest floating point representation. For any real number x within the representable range:

$$\frac{|\text{fl}(x) - x|}{|x|} \leq u$$

where $\text{fl}(x)$ denotes the floating point representation of x.

NOTE The numerical difference between machine epsilon and unit roundoff is subtle but important. Machine epsilon is the smallest number that, when added to 1, yields a different floating point number. Unit roundoff is the maximum relative error in representing any number. In IEEE arithmetic with round-to-nearest mode, $u = \varepsilon_{\text{mach}}/2$.

Absolute and Relative Error

When a real number x is approximated by a computed value $\hat{x}$, the **absolute error** is defined as

$$\delta = \hat{x} - x$$

The **relative error** is given by

$$\varepsilon = \frac{\hat{x} - x}{x} = \frac{\delta}{x}$$

so that the computed value can be expressed as

$$\hat{x} = x + \delta \quad \text{or} \quad \hat{x} = x(1 + \varepsilon)$$

Examples

Absolute vs. Relative Error: Consider approximating $\pi \approx 3.14159265359...$ with different values:

Approximation	Absolute Error	Relative Error
3.14	0.00159...	0.000507...
3.1	0.04159...	0.01324...
3	0.14159...	0.04507...

Now consider approximating 10^{-6} by 0.99×10^{-6}:

- Absolute error: $\delta = 0.99 \times 10^{-6} - 10^{-6} = -0.01 \times 10^{-6} = -10^{-8}$
- Relative error: $\varepsilon = \frac{-10^{-8}}{10^{-6}} = -0.01$ or -1%

This example illustrates why relative error is often more meaningful than absolute error: a small absolute error might represent a large relative error for small numbers, and vice versa for large numbers.

Floating Point Operations and Error Propagation

In IEEE floating point arithmetic, operations like addition, subtraction, multiplication, and division are performed as if the exact result were calculated and then rounded to the nearest floating point number. For any binary operation $\circledast$ between floating point numbers x and y:

$$\text{fl}(x \circledast y) = (x \circledast y)(1 + \delta), \quad |\delta| \leq u$$

where u is the unit roundoff. This model helps us analyze how errors propagate through computations.

Examples

Error Propagation in Addition: Consider adding two floating point numbers a and b. The computed sum has the form:

$$\text{fl}(a+b) = (a+b)(1+\delta), \quad |\delta| \leq u$$

If $a \approx -b$ (nearly equal in magnitude but opposite in sign), cancellation occurs. For example, with 3-digit decimal arithmetic:

$$\begin{aligned} a &= 1.00 \\ b &= -0.995 \end{aligned}$$

The exact sum is 0.005, but in 3-digit arithmetic:

$$\begin{aligned} \text{fl}(a+b) &= \text{fl}(1.00 + (-0.995)) \\ &= \text{fl}(0.005) \\ &= 0.00500 \end{aligned}$$

If a and b both have small relative errors ($\hat{a} = a(1+\varepsilon_a)$ and $\hat{b} = b(1+\varepsilon_b)$ where $|\varepsilon_a|, |\varepsilon_b| \leq u$), the relative error in the computed sum can be much larger:

$$\frac{\text{fl}(\hat{a}+\hat{b}) - (a+b)}{a+b} \approx \frac{a\varepsilon_a + b\varepsilon_b}{a+b}$$

When $a \approx -b$, the denominator becomes very small, magnifying the error.

Catastrophic Cancellation

Catastrophic cancellation occurs when subtracting two nearly equal numbers. Although the absolute error in each number may be small, their difference can lead to a significant loss of significant digits.
If $\hat{x} = x(1+\varepsilon_x)$ and $\hat{y} = y(1+\varepsilon_y)$ where $x \approx y$, then:

$$\hat{x} - \hat{y} = (x-y) + x\varepsilon_x - y\varepsilon_y$$

The relative error becomes:

$$\frac{(\hat{x}-\hat{y}) - (x-y)}{x-y} = \frac{x\varepsilon_x - y\varepsilon_y}{x-y}$$

When $x \approx y$, the denominator is small, potentially making the relative error very large.

Examples

Catastrophic Cancellation Example:
Consider computing $f(x) = \sqrt{x+1} - \sqrt{x}$ for $x = 10^8$ using double precision.
Direct computation:

$$\begin{aligned} \sqrt{10^8+1} &\approx 10000.00000005 \\ \sqrt{10^8} &= 10000 \\ f(10^8) &\approx 10000.00000005 - 10000 = 5 \times 10^{-8} \end{aligned}$$

Due to rounding in floating point, we might only get:

$$f(10^8) \approx 0$$

However, algebraic manipulation gives an equivalent form:

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

Using this formula:

$$f(10^8) = \frac{1}{10000.00000005 + 10000} \approx \frac{1}{20000} = 5 \times 10^{-5}$$

The second approach avoids catastrophic cancellation and gives a more accurate result. This illustrates that mathematical reformulation can often mitigate numerical issues.

Special Values in IEEE Floating Point

IEEE 754 defines several special values to handle exceptional cases:

- **Zeros**: Both +0 and -0 are represented, though they compare as equal.
- **Subnormal numbers (denormalized)**: Numbers smaller than the smallest normalized number, represented with a leading zero instead of one. They provide gradual underflow but with reduced precision.
- **Infinities**: $+\infty$ and $-\infty$ represent overflow or division by zero.
- **NaN (Not a Number)**: Represents undefined operations like 0/0 or $\sqrt{-1}$. There are actually many different NaN values with different internal bit patterns.

These special values ensure that every floating point operation produces a well-defined result, making numerical algorithms more robust.

Examples

Subnormal Numbers Range and Precision: For IEEE double precision:

- Smallest positive normalized number: $2^{-1022} \approx 2.22 \times 10^{-308}$
- Smallest positive subnormal number: $2^{-1022} \times 2^{-52} = 2^{-1074} \approx 4.94 \times 10^{-324}$

Subnormal numbers fill the gap between 0 and the smallest normalized number, but with reduced precision. While a normalized double precision number has 53 bits of precision, a subnormal number might have significantly fewer, depending on its magnitude.

Examples

For example, the subnormal number 2^{-1073} has only 1 bit of precision, making its relative representation error much larger than for normalized numbers.

Floating Point Arithmetic Properties

Floating point arithmetic differs from real arithmetic in several important ways:

- **Non-associativity**: In general, $(a + b) + c \neq a + (b + c)$ due to rounding errors.
- **Non-distributivity**: Generally, $a \times (b+c) \neq a \times b + a \times c$ due to rounding errors.
- **Absorption**: When $|a| \gg |b|$, the computation of $a + b$ might result in just a if b is too small to affect the representation of a.
- **Inexact cancellation**: As discussed, subtracting nearly equal quantities can result in significant loss of precision.

Understanding these properties is crucial for designing stable numerical algorithms.

Examples

Non-associativity Example: Consider adding three numbers in different orders using 4-digit decimal arithmetic:

$$\begin{aligned}(1.000 \times 10^{10} + 1.000) + (-1.000 \times 10^{10}) &= 1.000 \times 10^{10} + (-1.000 \times 10^{10}) \\ &= 0.000\end{aligned}$$

But in the other order:

$$\begin{aligned}1.000 \times 10^{10} + (1.000 + (-1.000 \times 10^{10})) &= 1.000 \times 10^{10} + (-9.999 \times 10^{9}) \\ &= 1.000 \times 10^{9}\end{aligned}$$

The large difference in results demonstrates why the order of operations matters in floating point arithmetic, especially when combining values of significantly different magnitudes.

Error Analysis Strategies

When performing error analysis for numerical algorithms, several approaches are commonly used:

- **Forward error analysis**: Estimate how input errors and rounding errors propagate to affect the output.
- **Backward error analysis**: Find the smallest perturbation to the inputs that would make the computed result exact.
- **Interval arithmetic**: Carry interval bounds through the computation to guarantee containment of the true result.
- **Condition number analysis**: Determine how sensitive the problem is to small changes in the input data.

These analyses help assess the reliability and accuracy of numerical results.

Examples

Let $\hat{\boldsymbol{x}}$ denote the computed solution to the linear system $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$. Backward error analysis seeks to determine the perturbation $\mathbf{E}$ to $\mathbf{A}$ such that $\hat{\boldsymbol{x}}$ is the exact solution to the perturbed system:

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

To proceed, define the residual:

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

By rearranging we get:

$$\mathbf{A}\hat{\boldsymbol{x}} + \boldsymbol{r} = \boldsymbol{b},$$

which can be rewritten as:

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

where the perturbation matrix $\mathbf{E}$ is given explicitly by:

$$\mathbf{E} = -\frac{\boldsymbol{r}\hat{\boldsymbol{x}}^\top}{\|\hat{\boldsymbol{x}}\|^2}.$$

The *relative backward error* is defined as:

$$\frac{\|\mathbf{E}\|}{\|\mathbf{A}\|}.$$

In practice, this quantity is often much smaller than the *relative forward error*:

$$\frac{\|\hat{\boldsymbol{x}} - \boldsymbol{x}\|}{\|\boldsymbol{x}\|}.$$

This explains why algorithms that exhibit large forward errors may still be practically useful: they produce solutions that are exact for a nearby problem.

Floating point arithmetic forms the foundation of numerical computing, with its inherent limitations shaping the design and analysis of algorithms. By understanding floating point representation, error propagation, and special cases, we can develop numerically stable algorithms that produce reliable results even in the presence of unavoidable computational errors. Remember that mathematical reformulations, careful algorithm design, and appropriate error analysis are often more effective than simply increasing precision.

3 Eigenvalues and Matrix Properties

3.1 Eigenvalues and Eigenvectors

Eigenvalues and Eigenvectors

For a matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$, a scalar $\lambda \in \mathbb{C}$ is called an *eigenvalue* of $\mathbf{A}$ if there exists a nonzero vector $\boldsymbol{v} \in \mathbb{C}^n$ such that

$$\mathbf{A}\boldsymbol{v} = \lambda\boldsymbol{v}.$$

The nonzero vector $\boldsymbol{v}$ is called an *eigenvector* of $\mathbf{A}$ corresponding to the eigenvalue λ. The set of all eigenvalues of $\mathbf{A}$ is called the *spectrum* of $\mathbf{A}$, denoted by $\sigma(\mathbf{A})$.

Intuitively, an eigenvector is a vector whose direction remains unchanged when transformed by the matrix $\mathbf{A}$—it may be stretched, compressed, or reversed, but it stays on the same line. The eigenvalue tells us the factor by which the eigenvector is scaled. Eigenvalues and eigenvectors reveal the fundamental character of a linear transformation, showing the directions in which the transformation behaves most simply.

Characteristic Polynomial

For a matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$, the *characteristic polynomial* is defined as

$$p_{\mathbf{A}}(\lambda) = \det(\mathbf{A} - \lambda\mathbf{I}),$$

where $\mathbf{I}$ is the $n \times n$ identity matrix. The equation

$$\det(\mathbf{A} - \lambda\mathbf{I}) = 0$$

is called the *characteristic equation* of $\mathbf{A}$. The roots of the characteristic polynomial are precisely the eigenvalues of $\mathbf{A}$.

Examples

Consider the matrix

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

To find its eigenvalues, we compute the characteristic polynomial:

$$\begin{aligned} p_{\mathbf{A}}(\lambda) &= \det(\mathbf{A} - \lambda\mathbf{I}) && (2)\\ &= \det\begin{pmatrix} 3-\lambda & 1 \\ 1 & 3-\lambda \end{pmatrix} && (3)\\ &= (3-\lambda)(3-\lambda) - 1\cdot 1 && (4)\\ &= (3-\lambda)^2 - 1 && (5)\\ &= 9 - 6\lambda + \lambda^2 - 1 && (6)\\ &= \lambda^2 - 6\lambda + 8 && (7)\\ &= (\lambda-4)(\lambda-2) && (8) \end{aligned}$$

Setting $p_{\mathbf{A}}(\lambda) = 0$, we get the eigenvalues $\lambda_1 = 4$ and $\lambda_2 = 2$.
For $\lambda_1 = 4$, we find the corresponding eigenvector by solving $(\mathbf{A} - 4\mathbf{I})\boldsymbol{v} = \mathbf{0}$:

$$\begin{aligned} \begin{pmatrix} 3-4 & 1 \\ 1 & 3-4 \end{pmatrix}\begin{pmatrix} v_1 \\ v_2 \end{pmatrix} &= \begin{pmatrix} 0 \\ 0 \end{pmatrix} && (9)\\ \begin{pmatrix} -1 & 1 \\ 1 & -1 \end{pmatrix}\begin{pmatrix} v_1 \\ v_2 \end{pmatrix} &= \begin{pmatrix} 0 \\ 0 \end{pmatrix} && (10) \end{aligned}$$

This gives us $-v_1+v_2 = 0$, so $v_1 = v_2$. Any nonzero vector of the form $\boldsymbol{v}_1 = (t, t)^T$ for $t \neq 0$ is an eigenvector. A normalized eigenvector would be $\boldsymbol{v}_1 = \frac{1}{\sqrt{2}}(1, 1)^T$. Similarly, for $\lambda_2 = 2$, we solve $(\mathbf{A} - 2\mathbf{I})\boldsymbol{v} = \mathbf{0}$:

$$\begin{pmatrix} 3-2 & 1 \\ 1 & 3-2 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \tag{11}$$

$$\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \tag{12}$$

This gives us $v_1 + v_2 = 0$, so $v_1 = -v_2$. Any nonzero vector of the form $\boldsymbol{v}_2 = (t, -t)^T$ for $t \neq 0$ is an eigenvector. A normalized eigenvector would be $\boldsymbol{v}_2 = \frac{1}{\sqrt{2}}(1, -1)^T$.

Key Properties of Eigenvalues

For a matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$:

1. The sum of the eigenvalues equals the trace of $\mathbf{A}$: $\sum_{i=1}^{n} \lambda_i = \text{tr}(\mathbf{A})$.
2. The product of the eigenvalues equals the determinant of $\mathbf{A}$: $\prod_{i=1}^{n} \lambda_i = \det(\mathbf{A})$.
3. The eigenvalues of $\mathbf{A}^k$ are λ_i^k for $i = 1, 2, \ldots, n$.
4. The eigenvalues of $\mathbf{A}^{-1}$ (if $\mathbf{A}$ is invertible) are $\frac{1}{\lambda_i}$ for $i = 1, 2, \ldots, n$.
5. If $\mathbf{A}$ is triangular, its eigenvalues are exactly the diagonal entries.
6. Similar matrices (i.e., matrices $\mathbf{A}$ and $\mathbf{B}$ where $\mathbf{B} = \mathbf{V}^{-1}\mathbf{A}\mathbf{V}$ for some invertible $\mathbf{V}$) have the same eigenvalues.

Diagonalization

A matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ is *diagonalizable* if there exists an invertible matrix $\mathbf{V}$ and a diagonal matrix $\mathbf{\Lambda}$ such that

$$\mathbf{A} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^{-1}$$

where the diagonal entries of $\mathbf{\Lambda}$ are the eigenvalues of $\mathbf{A}$ and the columns of $\mathbf{V}$ are the corresponding eigenvectors.

NOTE

A matrix $\mathbf{A}$ is diagonalizable if and only if there exist n linearly independent eigenvectors, which occurs if and only if the geometric multiplicity of each eigenvalue equals its algebraic multiplicity.

The eigenvalue problem is one of the most fundamental in numerical linear algebra, with applications ranging from structural engineering (vibration analysis) to quantum mechanics (energy states), machine learning (principal component analysis), and dynamical systems (stability analysis). The efficiency and accuracy of eigenvalue algorithms have direct implications for these applications, making them a central focus in scientific computing.

3.2 Symmetric Positive Definite (SPD) Matrices

Symmetric Matrix

A matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ is called *symmetric* if

$$\mathbf{A} = \mathbf{A}^T,$$

meaning that the entry in the ith row and jth column is the same as the entry in the jth row and ith column for all $1 \leq i, j \leq n$.

Positive Definite Matrix

A symmetric matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ is *positive definite* (PD) if

$$\boldsymbol{x}^T \mathbf{A} \boldsymbol{x} > 0 \quad \text{for every nonzero } \boldsymbol{x} \in \mathbb{R}^n.$$

In computational terms, when we transform any nonzero vector $\boldsymbol{x}$ using the quadratic form $\boldsymbol{x}^T \mathbf{A} \boldsymbol{x}$, the result is always a positive number.

Metric Induced by an SPD Matrix

Let $\mathbf{A} \in \mathbb{R}^{n \times n}$ be an SPD matrix. Then, the function

$$d_{\mathbf{A}}(\boldsymbol{x}, \boldsymbol{y}) = \sqrt{(\boldsymbol{x} - \boldsymbol{y})^T \mathbf{A} (\boldsymbol{x} - \boldsymbol{y})}$$

defines a distance function on $\mathbb{R}^n$. This generalizes the standard Euclidean distance ((in this case, x$\mathbf{A} = \mathbf{I}$)) by incorporating the transformation defined by $\mathbf{A}$. It satisfies all the properties of a metric:

1. **Non-negativity:** $d_{\mathbf{A}}(\boldsymbol{x}, \boldsymbol{y}) \geq 0$ for all $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$.
2. **Zero distance means equality:** $d_{\mathbf{A}}(\boldsymbol{x}, \boldsymbol{y}) = 0$ if and only if $\boldsymbol{x} = \boldsymbol{y}$.
3. **Symmetry:** $d_{\mathbf{A}}(\boldsymbol{x}, \boldsymbol{y}) = d_{\mathbf{A}}(\boldsymbol{y}, \boldsymbol{x})$ for all $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$.
4. **Triangle Inequality:** $d_{\mathbf{A}}(\boldsymbol{x}, \boldsymbol{z}) \leq d_{\mathbf{A}}(\boldsymbol{x}, \boldsymbol{y}) + d_{\mathbf{A}}(\boldsymbol{y}, \boldsymbol{z})$ for all $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z} \in \mathbb{R}^n$.

This matrix-based distance function is particularly useful in scientific computing applications like clustering, optimization, and data analysis where standard Euclidean distance might not capture the right relationships between data points.

SPD Interpratation

An SPD matrix transforms vector spaces by rescaling their components in a way that preserves a consistent notion of size. In standard Euclidean space, the norm $\|\boldsymbol{x}\|$ returns a positive length for any nonzero vector $\boldsymbol{x}$. Similarly, an SPD matrix $\mathbf{A}$ creates a new norm

$$\|\boldsymbol{x}\|_{\mathbf{A}} = \sqrt{\boldsymbol{x}^T \mathbf{A} \boldsymbol{x}},$$

which remains strictly positive for all nonzero $\boldsymbol{x}$. This property ensures that the transformation never collapses vectors to zero length or produces negative squared lengths, which is crucial for numerical stability in computational algorithms.

NOTE

Another way to visualize SPD matrices is through their geometric interpretation. When you use an SPD matrix $\mathbf{A}$ to compute distances via $\boldsymbol{x}^T\mathbf{A}\boldsymbol{x}$, the set of all vectors with constant "size" forms an ellipsoid (a stretched or squashed sphere). This is important in scientific computing applications like principal component analysis (PCA), where we often need to understand how data varies in different directions. The axes of this ellipsoid correspond to the eigenvectors of $\mathbf{A}$, and their lengths are determined by the eigenvalues.

Key Properties of SPD Matrices

Let $\mathbf{A} \in \mathbb{R}^{n\times n}$ be an SPD matrix. Then:

1. Every eigenvalue λ of $\mathbf{A}$ satisfies $\lambda > 0$.
2. $\mathbf{A}$ is invertible, meaning it has full rank and its determinant is non-zero.
3. The function $\langle \boldsymbol{x}, \boldsymbol{y}\rangle_{\mathbf{A}} = \boldsymbol{x}^T\mathbf{A}\boldsymbol{y}$ defines an inner product, which induces the norm $\|\boldsymbol{x}\|_{\mathbf{A}} = \sqrt{\boldsymbol{x}^T\mathbf{A}\boldsymbol{x}}$. This gives us a computationally consistent way to calculate distances in transformed spaces.

From a computational perspective, the condition $\boldsymbol{x}^T\mathbf{A}\boldsymbol{x} > 0$ for all nonzero $\boldsymbol{x}$ ensures that the matrix $\mathbf{A}$ never maps a vector to a zero result through the quadratic form, nor does it produce negative values. This is critical in numerical algorithms because it guarantees the stability and uniqueness of solutions. For example, when solving linear systems like $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ where $\mathbf{A}$ is SPD, efficient methods like Cholesky decomposition can be used, and the solution is guaranteed to exist and be unique.

NOTE SPD matrices are fundamental in scientific computing applications. For example:

- In machine learning, covariance matrices are SPD and used to understand data distributions
- In numerical optimization, the Hessian matrix of a convex function is SPD near the minimum
- In finite element methods, the stiffness matrix is typically SPD
- In numerical linear algebra, SPD matrices allow for specialized algorithms like Cholesky factorization, which is twice as efficient as LU decomposition
- In image processing, SPD matrices can represent local structure tensors that capture directional information

Their computational properties make algorithms more stable, efficient, and reliable.

3.2.1 Cholesky Decomposition

Cholesky Decomposition

For a symmetric positive definite (SPD) matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$, the Cholesky Decomposition expresses $\mathbf{A}$ as

$$\mathbf{A} = \mathbf{L}\mathbf{L}^T,$$

where $\mathbf{L}$ is a unique lower triangular matrix with strictly positive diagonal entries.

SPD matrices, which satisfy $\boldsymbol{x}^T\mathbf{A}\boldsymbol{x} > 0$ for all nonzero $\boldsymbol{x}$, are prevalent in optimization, statistics, and finite element methods. The Cholesky factorization is both computationally efficient and numerically stable since it avoids the need for pivoting.

Algorithm for Cholesky Decomposition

For an SPD matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$, the Cholesky factorization $\mathbf{A} = \mathbf{L}\mathbf{L}^T$ can be computed entry by entry as follows:
For each entry l_{ij} of $\mathbf{L}$ where $1 \le i, j \le n$:

$$l_{ij} = \begin{cases} 0, & \text{if } i < j \\ \sqrt{a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2}, & \text{if } i = j \\ \frac{1}{l_{jj}}\left(a_{ij} - \sum_{k=1}^{j-1} l_{ik}l_{jk}\right), & \text{if } i > j \end{cases} \tag{13}$$

This algorithm proceeds row by row, using previously computed elements of $\mathbf{L}$ to determine subsequent entries.

Derivation of the Cholesky Algorithm (Matrix Form)
To derive the Cholesky algorithm, we start with the relationship $\mathbf{A} = \mathbf{L}\mathbf{L}^T$ where $\mathbf{A}$ is an SPD matrix and $\mathbf{L}$ is lower triangular. Let's express both matrices explicitly:

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}, \quad \mathbf{L} = \begin{pmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{pmatrix}$$

The transpose of $\mathbf{L}$ is:

$$\mathbf{L}^T = \begin{pmatrix} l_{11} & l_{21} & \cdots & l_{n1} \\ 0 & l_{22} & \cdots & l_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & l_{nn} \end{pmatrix}$$

Now, the matrix product $\mathbf{L}\mathbf{L}^T$ gives us:

$$\mathbf{LL}^T = \begin{pmatrix} l_{11}^2 & l_{11}l_{21} & \cdots & l_{11}l_{n1} \\ l_{21}l_{11} & l_{21}^2 + l_{22}^2 & \cdots & l_{21}l_{n1} + l_{22}l_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1}l_{11} & l_{n1}l_{21} + l_{n2}l_{22} & \cdots & \sum_{k=1}^{n} l_{nk}^2 \end{pmatrix}$$

Equating this with $\mathbf{A}$, we can derive the formulas for different elements of $\mathbf{L}$:
Case 1: Diagonal elements (position (i, i))
From the diagonal entries of $\mathbf{A} = \mathbf{LL}^T$, we have:

$$a_{ii} = \sum_{k=1}^{i} l_{ik}^2 = l_{i1}^2 + l_{i2}^2 + \cdots + l_{ii}^2$$

Solving for l_{ii}:

$$l_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2}$$

Case 2: Below diagonal (position (i, j) where $i > j$)
From the below-diagonal entries of $\mathbf{A} = \mathbf{LL}^T$, we have:

$$a_{ij} = \sum_{k=1}^{j} l_{ik}l_{jk} = l_{i1}l_{j1} + l_{i2}l_{j2} + \cdots + l_{ij}l_{jj}$$

Since $l_{jk} = 0$ for $k > j$ (as $\mathbf{L}$ is lower triangular), the sum stops at $k = j$. Solving for l_{ij}:

$$l_{ij} = \frac{1}{l_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik}l_{jk} \right)$$

Case 3: Above diagonal (position (i, j) where $i < j$)
Since $\mathbf{A}$ is symmetric, $a_{ij} = a_{ji}$. The elements of $\mathbf{L}$ above the diagonal are defined to be zero: $l_{ij} = 0$ for $i < j$.
This gives us the complete algorithm for computing the Cholesky decomposition:

$$l_{ij} = \begin{cases} 0, & \text{if } i < j \\ \sqrt{a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2}, & \text{if } i = j \\ \frac{1}{l_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik}l_{jk} \right), & \text{if } i > j \end{cases}$$

The algorithm proceeds by computing $\mathbf{L}$ one column at a time, from left to right, and within each column from top to bottom.

Computational Complexity (Cholesky)

The Cholesky decomposition algorithm requires:

- Approximately $\frac{1}{3}n^3$ floating-point operations (flops) for an $n \times n$ matrix
- This is roughly half the cost of LU factorization ($\frac{2}{3}n^3$ flops), which is the standard method for non-symmetric matrices

- Storage requirement of $\frac{n(n+1)}{2}$ elements (only the lower triangular part is stored)

The efficiency comes from exploiting both the symmetry of $\mathbf{A}$ and its positive definiteness, which eliminates the need for pivoting strategies.

Examples Let's compute the Cholesky decomposition of:

$$\mathbf{A} = \begin{pmatrix} 4 & 2 \\ 2 & 3 \end{pmatrix}$$

Step 1: Compute the first column of $\mathbf{L}$.

$$l_{11} = \sqrt{a_{11}} = \sqrt{4} = 2 \tag{14}$$

$$l_{21} = \frac{a_{21}}{l_{11}} = \frac{2}{2} = 1 \tag{15}$$

Step 2: Compute the second column of $\mathbf{L}$.

$$l_{22} = \sqrt{a_{22} - l_{21}^2} = \sqrt{3 - 1^2} = \sqrt{2} \approx 1.414 \tag{16}$$

Therefore:

$$\mathbf{L} = \begin{pmatrix} 2 & 0 \\ 1 & \sqrt{2} \end{pmatrix}$$

Verification:

$$\mathbf{L}\mathbf{L}^T = \begin{pmatrix} 2 & 0 \\ 1 & \sqrt{2} \end{pmatrix} \begin{pmatrix} 2 & 1 \\ 0 & \sqrt{2} \end{pmatrix} = \begin{pmatrix} 4 & 2 \\ 2 & 3 \end{pmatrix} = \mathbf{A}$$

NOTE The uniqueness of the Cholesky Decomposition stems from the positivity of the leading principal minors of an SPD matrix, ensuring that each pivot in the elimination process is nonzero. For an SPD matrix $\mathbf{A}$, the leading principal minor of order k, denoted as $\det(\mathbf{A}_k)$, is the determinant of the submatrix formed by the first k rows and columns. These determinants are all positive, which ensures that the diagonal entries of $\mathbf{L}$ are well-defined and unique.

3.3 Orthogonal Matrices

Orthogonal Vectors Two vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$ are called *orthogonal* if their inner product is zero:

$$\boldsymbol{x}^T \boldsymbol{y} = 0$$

Geometrically, orthogonal vectors are perpendicular to each other. A set of vectors $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_k$ is called *mutually orthogonal* if every pair of distinct vectors in the set is orthogonal:

$$\boldsymbol{v}_i^T \boldsymbol{v}_j = 0 \quad \text{for all } i \neq j$$

Orthonormal Basis A set of vectors $\boldsymbol{q}_1, \boldsymbol{q}_2, \ldots, \boldsymbol{q}_n$ in $\mathbb{R}^n$ is called an *orthonormal basis* if:

1. The vectors are *mutually orthogonal*: $\boldsymbol{q}_i^T \boldsymbol{q}_j = 0$ for all $i \neq j$
2. Each vector has unit length: $|\boldsymbol{q}_i|_2 = 1$ for all $i = 1, 2, \ldots, n$
3. The vectors span $\mathbb{R}^n$ (which is automatic if we have n linearly independent vectors in $\mathbb{R}^n$)

Equivalently, a set of vectors $\boldsymbol{q}_1, \boldsymbol{q}_2, \ldots, \boldsymbol{q}_n$ forms an orthonormal basis if:

$$\boldsymbol{q}_i^T \boldsymbol{q}_j = \delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}$$

If we arrange these vectors as columns of a matrix $\mathbf{Q} = [\boldsymbol{q}_1, \boldsymbol{q}_2, \ldots, \boldsymbol{q}_n]$, then $\mathbf{Q}$ is an orthogonal matrix, satisfying $\mathbf{Q}^T\mathbf{Q} = \mathbf{Q}\mathbf{Q}^T = \mathbf{I}$.

NOTE Any vector $\boldsymbol{v} \in \mathbb{R}^n$ can be expressed uniquely as a linear combination of vectors from an orthonormal basis $\boldsymbol{q}_1, \boldsymbol{q}_2, \ldots, \boldsymbol{q}_n$:

$$\boldsymbol{v} = (\boldsymbol{q}_1^T \boldsymbol{v})\boldsymbol{q}_1 + (\boldsymbol{q}_2^T \boldsymbol{v})\boldsymbol{q}_2 + \ldots + (\boldsymbol{q}_n^T \boldsymbol{v})\boldsymbol{q}_n = \sum_{i=1}^{n} (\boldsymbol{q}_i^T \boldsymbol{v})\boldsymbol{q}_i$$

The coefficients $\boldsymbol{q}_i^T \boldsymbol{v}$ are the projections of $\boldsymbol{v}$ onto the basis vectors. This simple formula for the coefficients is a key computational advantage of orthonormal bases - *we don't need to solve a system of equations to find the coefficients.*

Orthogonal Matrix A square matrix $\mathbf{Q} \in \mathbb{R}^{n \times n}$ is called *orthogonal* if its columns form an orthonormal set in $\mathbb{R}^n$, which means: $\mathbf{Q}^T\mathbf{Q} = \mathbf{Q}\mathbf{Q}^T = \mathbf{I}_n$ Equivalently, $\mathbf{Q}^T = \mathbf{Q}^{-1}$, meaning the transpose of an orthogonal matrix is its inverse.

Orthogonal matrices represent rotations, reflections, or combinations of these operations in $\mathbb{R}^n$. They preserve vector lengths and angles between vectors, making them particularly valuable in scientific computing where maintaining geometric properties is important. When multiplying a vector by an orthogonal matrix, the result is a vector with the same length but potentially different direction.

Properties of Orthogonal Matrices Let $\mathbf{Q} \in \mathbb{R}^{n \times n}$ be an orthogonal matrix. Then:

1. The columns of $\mathbf{Q}$ form an orthonormal basis for $\mathbb{R}^n$.
2. The rows of $\mathbf{Q}$ also form an orthonormal basis for $\mathbb{R}^n$.
3. $\det(\mathbf{Q}) = \pm 1$. When $\det(\mathbf{Q}) = 1$, $\mathbf{Q}$ represents a rotation; when $\det(\mathbf{Q}) = -1$, it represents a reflection or rotoreflection.
4. $\|\mathbf{Q}\boldsymbol{x}\|_2 = \|\boldsymbol{x}\|_2$ for all $\boldsymbol{x} \in \mathbb{R}^n$, meaning orthogonal transformations preserve Euclidean lengths.

5. The condition number (with respect to the 2-norm) is $\kappa_2(\mathbf{Q}) = 1$, making orthogonal matrices optimally conditioned for numerical computations.

6. If $\mathbf{Q}_1$ and $\mathbf{Q}_2$ are orthogonal matrices, then their product $\mathbf{Q}_1\mathbf{Q}_2$ is also orthogonal.

3.3.1 QR Decomposition

QR Decomposition

For a matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ with $m \geq n$, the QR Decomposition factors $\mathbf{A}$ as

$$\mathbf{A} = \mathbf{QR},$$

where $\mathbf{Q} \in \mathbb{R}^{m\times m}$ is an orthogonal matrix (i.e., $\mathbf{Q}^T\mathbf{Q} = \mathbf{I}$) and $\mathbf{R} \in \mathbb{R}^{m\times n}$ is an upper triangular matrix with zeros below the main diagonal.

The QR factorization is particularly valuable in scientific computing because it transforms problems involving arbitrary matrices into problems with triangular matrices, which are much easier to solve. In addition, the orthogonality of $\mathbf{Q}$ ensures numerical stability in many applications, as orthogonal transformations preserve Euclidean norms and don't amplify errors.

Properties of QR Decomposition

The QR decomposition of a matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ with $m \geq n$ has several important properties:

1. If $\mathbf{A}$ has full column rank, then the first n columns of $\mathbf{Q}$ form an orthonormal basis for the column space of $\mathbf{A}$.

2. The diagonal elements of $\mathbf{R}$ satisfy $|r_{ii}| = \|\boldsymbol{q}_i^T\mathbf{A}\|_2$, where $\boldsymbol{q}_i$ is the ith column of $\mathbf{Q}$.

3. If $\mathbf{A}$ has full column rank, then the upper triangular matrix $\mathbf{R}$ has nonzero diagonal elements.

4. The computational complexity for the QR decomposition is approximately $2mn^2 - \frac{2}{3}n^3$ floating-point operations.

3.3.2 Grahm Schmidt

Orthogonal Complement

Let $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_k\}$ be a set of vectors in $\mathbb{R}^n$ that spans a space S. The *orthogonal complement* of S, denoted by $S^\perp$ (read as "S perp"), is the set of all vectors in $\mathbb{R}^n$ that are orthogonal to every vector in S:

$$S^\perp = \{\boldsymbol{w} \in \mathbb{R}^n : \boldsymbol{w}^T\boldsymbol{v}_i = 0 \text{ for all } i = 1, 2, \ldots, k\}$$

Important properties of the orthogonal complement include:

1. $S^\perp$ forms a valid space of vectors in $\mathbb{R}^n$.
2. If the vectors $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_k\}$ are linearly independent and $k < n$, then $S^\perp$ is non-empty.
3. If $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_k\}$ spans a space of dimension k, then $S^\perp$ has dimension $n - k$.
4. Any vector $\boldsymbol{x} \in \mathbb{R}^n$ can be uniquely written as $\boldsymbol{x} = \boldsymbol{x}_S + \boldsymbol{x}_{S^\perp}$ where $\boldsymbol{x}_S$ is in the space spanned by $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_k\}$ and $\boldsymbol{x}_{S^\perp}$ is in $S^\perp$.
5. If the columns of matrix $\mathbf{A}$ are $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_k\}$, then $S^\perp$ consists of all solutions to $\mathbf{A}^T\boldsymbol{x} = \mathbf{0}$.

Geometrically, $S^\perp$ consists of all vectors that are perpendicular to each vector in the set $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_k\}$.

Gram-Schmidt Process

The Gram-Schmidt process is an algorithm for converting a set of linearly independent vectors $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n\}$ into an orthogonal or orthonormal basis for the same subspace. The process works by sequentially projecting each vector onto the orthogonal complement of the subspace spanned by the previous orthogonalized vectors.

For a set of linearly independent vectors $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n\}$, the Gram-Schmidt process generates an orthogonal set $\{\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_n\}$ as follows:

$$\boldsymbol{u}_1 = \boldsymbol{v}_1 \tag{17}$$

$$\boldsymbol{u}_2 = \boldsymbol{v}_2 - \frac{\boldsymbol{u}_1^T\boldsymbol{v}_2}{\boldsymbol{u}_1^T\boldsymbol{u}_1}\boldsymbol{u}_1 \tag{18}$$

$$\boldsymbol{u}_3 = \boldsymbol{v}_3 - \frac{\boldsymbol{u}_1^T\boldsymbol{v}_3}{\boldsymbol{u}_1^T\boldsymbol{u}_1}\boldsymbol{u}_1 - \frac{\boldsymbol{u}_2^T\boldsymbol{v}_3}{\boldsymbol{u}_2^T\boldsymbol{u}_2}\boldsymbol{u}_2 \tag{19}$$

$$\vdots \tag{20}$$

$$\boldsymbol{u}_k = \boldsymbol{v}_k - \sum_{j=1}^{k-1} \frac{\boldsymbol{u}_j^T\boldsymbol{v}_k}{\boldsymbol{u}_j^T\boldsymbol{u}_j}\boldsymbol{u}_j \tag{21}$$

To obtain an orthonormal basis $\{\boldsymbol{q}_1, \boldsymbol{q}_2, \ldots, \boldsymbol{q}_n\}$, we normalize each orthogonal vector:

$$\boldsymbol{q}_i = \frac{\boldsymbol{u}_i}{\|\boldsymbol{u}_i\|_2} = \frac{\boldsymbol{u}_i}{\sqrt{\boldsymbol{u}_i^T\boldsymbol{u}_i}} \tag{22}$$

NOTE The Gram-Schmidt process can be interpreted geometrically:

- $\boldsymbol{u}_1$ is simply the first vector $\boldsymbol{v}_1$
- $\boldsymbol{u}_2$ is $\boldsymbol{v}_2$ minus its projection onto $\boldsymbol{u}_1$

- $\boldsymbol{u}_3$ is $\boldsymbol{v}_3$ minus its projections onto both $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$

Each $\boldsymbol{u}_k$ is the component of $\boldsymbol{v}_k$ that is orthogonal to the subspace spanned by $\{\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_{k-1}\}$.

Examples Gram-Schmidt Process Consider the vectors $\boldsymbol{v}_1 = [1, 1, 1]^T$ and $\boldsymbol{v}_2 = [0, 1, 2]^T$ in $\mathbb{R}^3$. Let's apply the Gram-Schmidt process:

- Step 1: $\boldsymbol{u}_1 = \boldsymbol{v}_1 = [1, 1, 1]^T$
- Step 2: We compute $\boldsymbol{u}_2$ as follows:

$$\boldsymbol{u}_2 = \boldsymbol{v}_2 - \frac{\boldsymbol{u}_1^T \boldsymbol{v}_2}{\boldsymbol{u}_1^T \boldsymbol{u}_1} \boldsymbol{u}_1 \tag{23}$$

$$= [0, 1, 2]^T - \frac{[1, 1, 1][0, 1, 2]^T}{[1, 1, 1][1, 1, 1]^T} [1, 1, 1]^T \tag{24}$$

$$= [0, 1, 2]^T - \frac{3}{3} [1, 1, 1]^T \tag{25}$$

$$= [0, 1, 2]^T - [1, 1, 1]^T \tag{26}$$

$$= [-1, 0, 1]^T \tag{27}$$

- To normalize:

$$\boldsymbol{q}_1 = \frac{\boldsymbol{u}_1}{\|\boldsymbol{u}_1\|_2} = \frac{[1, 1, 1]^T}{\sqrt{3}} = [\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}]^T \tag{28}$$

$$\boldsymbol{q}_2 = \frac{\boldsymbol{u}_2}{\|\boldsymbol{u}_2\|_2} = \frac{[-1, 0, 1]^T}{\sqrt{2}} = [-\frac{1}{\sqrt{2}}, 0, \frac{1}{\sqrt{2}}]^T \tag{29}$$

  Thus, we have transformed the linearly independent vectors $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ into the orthonormal vectors $\boldsymbol{q}_1$ and $\boldsymbol{q}_2$.

The Gram-Schmidt process provides a constructive proof of the existence of an orthonormal basis for any subspace of $\mathbb{R}^n$. In practice, however, the classical Gram-Schmidt algorithm can suffer from numerical instability due to accumulated rounding errors, especially for large sets of vectors. The modified Gram-Schmidt algorithm, which applies the orthogonalization against each previously computed orthogonal vector immediately, offers better numerical stability and is preferred in computational applications.

Modified Gram-Schmidt The Modified Gram-Schmidt algorithm is a numerically more stable version of the classical Gram-Schmidt process. Instead of computing all projections simultaneously for each new vector, it orthogonalizes against each previous vector sequentially:

$$\boldsymbol{v}_1^{(1)} = \boldsymbol{v}_1 \tag{30}$$

$$\boldsymbol{u}_1 = \boldsymbol{v}_1^{(1)} \tag{31}$$

$$\boldsymbol{v}_2^{(1)} = \boldsymbol{v}_2 \tag{32}$$

$$\boldsymbol{v}_2^{(2)} = \boldsymbol{v}_2^{(1)} - \frac{\boldsymbol{u}_1^T \boldsymbol{v}_2^{(1)}}{\boldsymbol{u}_1^T \boldsymbol{u}_1} \boldsymbol{u}_1 \tag{33}$$

$$\boldsymbol{u}_2 = \boldsymbol{v}_2^{(2)} \tag{34}$$

$$\vdots \tag{35}$$

$$\boldsymbol{v}_k^{(1)} = \boldsymbol{v}_k \tag{36}$$

$$\boldsymbol{v}_k^{(2)} = \boldsymbol{v}_k^{(1)} - \frac{\boldsymbol{u}_1^T \boldsymbol{v}_k^{(1)}}{\boldsymbol{u}_1^T \boldsymbol{u}_1} \boldsymbol{u}_1 \tag{37}$$

$$\boldsymbol{v}_k^{(3)} = \boldsymbol{v}_k^{(2)} - \frac{\boldsymbol{u}_2^T \boldsymbol{v}_k^{(2)}}{\boldsymbol{u}_2^T \boldsymbol{u}_2} \boldsymbol{u}_2 \tag{38}$$

$$\vdots \tag{39}$$

$$\boldsymbol{v}_k^{(k)} = \boldsymbol{v}_k^{(k-1)} - \frac{\boldsymbol{u}_{k-1}^T \boldsymbol{v}_k^{(k-1)}}{\boldsymbol{u}_{k-1}^T \boldsymbol{u}_{k-1}} \boldsymbol{u}_{k-1} \tag{40}$$

$$\boldsymbol{u}_k = \boldsymbol{v}_k^{(k)} \tag{41}$$

As before, to obtain an orthonormal basis, normalize each vector:

$$\boldsymbol{q}_i = \frac{\boldsymbol{u}_i}{\|\boldsymbol{u}_i\|_2} \tag{42}$$

NOTE The Gram-Schmidt process is closely related to the QR decomposition. If we arrange the original vectors $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n\}$ as columns of a matrix $\mathbf{A}$ and the orthonormal vectors $\{\boldsymbol{q}_1, \boldsymbol{q}_2, \ldots, \boldsymbol{q}_n\}$ as columns of a matrix $\mathbf{Q}$, then:

$$\mathbf{A} = \mathbf{QR}$$

where $\mathbf{R}$ is an upper triangular matrix with elements:

$$r_{ij} = \begin{cases} \boldsymbol{q}_i^T \boldsymbol{v}_j & \text{if } i \leq j \\ 0 & \text{if } i > j \end{cases}$$

The elements of $\mathbf{R}$ represent the projection coefficients computed during the Gram-Schmidt process.

Examples

Gram-Schmidt and QR Decomposition Consider the matrix $\mathbf{A} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 1 & 2 \end{bmatrix}$ whose columns are the vectors $\boldsymbol{v}_1 = [1, 1, 1]^T$ and $\boldsymbol{v}_2 = [0, 1, 2]^T$ from our previous example.
From the Gram-Schmidt process, we obtained:

$$\boldsymbol{q}_1 = [\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}]^T \tag{43}$$

$$\boldsymbol{q}_2 = [-\frac{1}{\sqrt{2}}, 0, \frac{1}{\sqrt{2}}]^T \tag{44}$$

The $\mathbf{R}$ matrix is given by:

$$r_{11} = \boldsymbol{q}_1^T \boldsymbol{v}_1 = \frac{1}{\sqrt{3}} \cdot 3 = \sqrt{3} \tag{45}$$

$$r_{12} = \boldsymbol{q}_1^T \boldsymbol{v}_2 = \frac{1}{\sqrt{3}} \cdot 3 = \sqrt{3} \tag{46}$$

$$r_{21} = 0 \quad \text{(by construction)} \tag{47}$$

$$r_{22} = \boldsymbol{q}_2^T \boldsymbol{v}_2^{(2)} = \boldsymbol{q}_2^T \boldsymbol{u}_2 = \frac{1}{\sqrt{2}} \cdot \sqrt{2} = \sqrt{2} \tag{48}$$

Therefore:

$$\mathbf{R} = \begin{bmatrix} \sqrt{3} & \sqrt{3} \\ 0 & \sqrt{2} \end{bmatrix}$$

And we can verify that $\mathbf{A} = \mathbf{QR}$:

$$\mathbf{QR} = \begin{bmatrix} \frac{1}{\sqrt{3}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{3}} & 0 \\ \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} \sqrt{3} & \sqrt{3} \\ 0 & \sqrt{2} \end{bmatrix} \tag{49}$$

$$= \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 1 & 2 \end{bmatrix} = \mathbf{A} \tag{50}$$

Applications of Gram-Schmidt

The Gram-Schmidt process and the related QR decomposition have numerous applications in numerical linear algebra and computational mathematics:

1. **Solving Linear Systems:** QR factorization provides a stable method for solving linear systems $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$.

2. **Least Squares Problems:** For overdetermined systems, the QR decomposition allows for efficient and stable computation of least squares solutions.

3. **Eigenvalue Algorithms:** QR decomposition forms the basis of the QR algorithm for computing eigenvalues.

4. **Coordinate Transformations:** Orthogonal bases simplify many calculations by allowing easy computation of coordinates and projections.

5. **Signal Processing:** Orthogonal projections are used for signal decomposition and filtering.

6. **Numerical PDEs:** Orthogonal basis functions are valuable in spectral methods for solving partial differential equations.

4 Iterative Methods for Linear Systems

4.1 Iterative Methods for Linear Systems

So far, we've examined direct methods for solving linear systems $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$. These methods compute the solution in a fixed number of operations, but for large systems, we need alternative approaches. Iterative methods provide an elegant solution by refining an initial guess through successive approximations.

Recap of Direct Methods

Before diving into iterative approaches, let's briefly recap the direct methods we've studied:

1. **Direct Inverse**: The analytical solution $\boldsymbol{x} = \mathbf{A}^{-1}\boldsymbol{b}$ requires explicitly computing $\mathbf{A}^{-1}$, which costs $O(n^3)$ operations and may suffer from numerical instability.

2. **Matrix Factorization**: Methods like LU, Cholesky, and QR decompose $\mathbf{A}$ into simpler matrices (e.g., $\mathbf{A} = \mathbf{LU}$). This factorization costs $O(n^3)$ operations, but subsequent solutions with new right-hand sides require only $O(n^2)$ operations.

NOTE

While these methods are effective for small to medium-sized problems, they become impractical for very large systems due to the high complexity, especially when $\mathbf{A}$ is sparse but its factors are not.

Iterative Methods

Iterative methods take a fundamentally different approach to solving linear systems. Instead of computing the solution directly, they:

1. Start with an initial guess $\boldsymbol{x}^{(0)}$

2. Generate a sequence of increasingly accurate approximations $\{\boldsymbol{x}^{(k)}\}_{k=0}^{\infty}$

3. Stop when a specified accuracy is achieved

We can think of iterative methods as navigating through the solution space. Starting from an initial position $\boldsymbol{x}^{(0)}$, we repeatedly follow a rule that moves us closer to the true solution $\boldsymbol{x}^*$. The rule is designed so that each step reduces our distance from the destination, eventually bringing us arbitrarily close to it.

The Iterative Principle

To understand the basic principle behind iterative methods, consider rearranging the linear system $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ into the form:

$$\boldsymbol{x} = \mathbf{T}\boldsymbol{x} + \boldsymbol{c}$$

where $\mathbf{T}$ is derived from $\mathbf{A}$, and $\boldsymbol{c}$ is derived from both $\mathbf{A}$ and $\boldsymbol{b}$.
This leads to the iterative scheme:

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

For this iteration to converge to the true solution $\boldsymbol{x}^*$, the matrix $\mathbf{T}$ must have specific properties—most importantly, its spectral radius must be less than 1. Different choices of $\mathbf{T}$ and $\boldsymbol{c}$ lead to different iterative methods. The challenge lies in choosing them to ensure:

- Convergence (the sequence actually reaches the solution)
- Rapid convergence (it gets there quickly)
- Computational efficiency (each iteration is inexpensive)

4.2 Jacobi Iteration

4.2.1 Jacobi Iteration

Jacobi Iteration Method

The Jacobi method is one of the simplest iterative techniques for solving a linear system $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ where $\mathbf{A} \in \mathbb{R}^{n\times n}$ and $\boldsymbol{b} \in \mathbb{R}^n$. The method decomposes the matrix $\mathbf{A}$ into its diagonal and off-diagonal components:

$$\mathbf{A} = \mathbf{D} + \mathbf{R}$$

where $\mathbf{D}$ is the diagonal matrix containing the diagonal entries of $\mathbf{A}$, and $\mathbf{R} = \mathbf{A} - \mathbf{D}$ contains all the off-diagonal entries.
The linear system can then be rewritten as:

$$\mathbf{D}\boldsymbol{x} + \mathbf{R}\boldsymbol{x} = \boldsymbol{b}$$

Assuming that all diagonal entries of $\mathbf{A}$ are non-zero (i.e., $a_{ii} \neq 0$ for all i), we can solve for $\boldsymbol{x}$:

$$\boldsymbol{x} = \mathbf{D}^{-1}(\boldsymbol{b} - \mathbf{R}\boldsymbol{x})$$

This equation forms the basis of the Jacobi iteration. Starting with an initial guess $\boldsymbol{x}^{(0)}$, we generate a sequence of approximations $\{\boldsymbol{x}^{(k)}\}_{k=0}^{\infty}$ using:

$$\boldsymbol{x}^{(k+1)} = \mathbf{D}^{-1}(\boldsymbol{b} - \mathbf{R}\boldsymbol{x}^{(k)})$$

In component form, this becomes:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right), \quad i = 1, 2, \ldots, n$$

This formula reveals the key characteristic of the Jacobi method: the computation of each component $x_i^{(k+1)}$ uses only the values from the previous iteration $\boldsymbol{x}^{(k)}$.

4.2.2 Convergence of Jacobi Iteration

Convergence of Jacobi Iteration

The Jacobi method does not always converge. The convergence behavior depends on the properties of the matrix $\mathbf{A}$. The iteration can be written in matrix form as:

$$\boldsymbol{x}^{(k+1)} = \mathbf{T}_J \boldsymbol{x}^{(k)} + \boldsymbol{c}$$

where $\mathbf{T}_J = -\mathbf{D}^{-1}\mathbf{R}$ is the Jacobi iteration matrix and $\boldsymbol{c} = \mathbf{D}^{-1}\boldsymbol{b}$.
The method converges for any initial guess $\boldsymbol{x}^{(0)}$ if and only if the spectral radius of $\mathbf{T}_J$ is less than 1:

$$\rho(\mathbf{T}_J) < 1$$

where $\rho(\mathbf{T}_J)$ is the maximum absolute value of the eigenvalues of $\mathbf{T}_J$.
Sufficient (but not necessary) conditions for convergence include:

- $\mathbf{A}$ is strictly diagonally dominant, i.e., $|a_{ii}| > \sum_{j \neq i} |a_{ij}|$ for all i
- $\mathbf{A}$ is symmetric positive definite

The rate of convergence is determined by $\rho(\mathbf{T}_J)$ – the closer this value is to 0, the faster the convergence.

Examples

Effect of Matrix Properties on Convergence Let's examine how different matrix properties affect the convergence of the Jacobi method.
Case 1: Strictly Diagonally Dominant Matrix

$$\mathbf{A}_1 = \begin{pmatrix} 5 & 1 & 1 \\ 1 & 5 & 1 \\ 1 & 1 & 5 \end{pmatrix}$$

For this matrix, $|a_{ii}| = 5 > 2 = \sum_{j \neq i} |a_{ij}|$ for all i. The Jacobi iteration matrix $\mathbf{T}_J$ has $\rho(\mathbf{T}_J) \approx 0.4$, ensuring fast convergence.
Case 2: Non-Diagonally Dominant Matrix

$$\mathbf{A}_2 = \begin{pmatrix} 2 & 1 & 3 \\ 1 & 3 & 1 \\ 2 & 2 & 2 \end{pmatrix}$$

For this matrix, the third row violates diagonal dominance. The Jacobi method still converges but more slowly, with $\rho(\mathbf{T}_J) \approx 0.8$.
Case 3: Non-Convergent Case

$$\mathbf{A}_3 = \begin{pmatrix} 1 & 3 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{pmatrix}$$

For this matrix, $\rho(\mathbf{T}_J) > 1$, and the Jacobi method diverges regardless of the initial guess.

These examples illustrate that the structure of $\mathbf{A}$ significantly impacts whether the Jacobi method converges and how quickly it does so.

Stopping Criteria

In practical implementations of the Jacobi method, we need to determine when to terminate the iteration. Common stopping criteria include:

1. **Residual-based**: Stop when the residual norm is small:
$$\|\mathbf{A}\boldsymbol{x}^{(k)} - \boldsymbol{b}\| < \epsilon_{\text{res}}$$
2. **Increment-based**: Stop when the change between successive iterations is small:
$$\|\boldsymbol{x}^{(k+1)} - \boldsymbol{x}^{(k)}\| < \epsilon_{\text{inc}}$$
3. **Relative Residual**: Stop when the relative residual is small:
$$\frac{\|\mathbf{A}\boldsymbol{x}^{(k)} - \boldsymbol{b}\|}{\|\boldsymbol{b}\|} < \epsilon_{\text{rel}}$$
4. **Maximum Iterations**: Stop after a predetermined number of iterations, regardless of convergence.

   The error after k iterations can be bounded by:
$$\|\boldsymbol{x}^{(k)} - \boldsymbol{x}^*\| \leq \frac{\rho(\mathbf{T}_J)^k}{1 - \rho(\mathbf{T}_J)}\|\boldsymbol{x}^{(1)} - \boldsymbol{x}^{(0)}\|$$
   where $\boldsymbol{x}^*$ is the exact solution. This bound shows that the error decreases geometrically at a rate determined by $\rho(\mathbf{T}_J)$.

4.3 Conjugate Gradient Method

The Conjugate Gradient (CG) method is an efficient iterative solver for symmetric positive definite linear systems. By combining steepest descent with a dynamically generated set of $\mathbf{A}$-conjugate search directions, it converges in at most n steps in exact arithmetic, while requiring only $O(n)$ storage and one matrix–vector product per iteration.

4.3.1 From Linear Systems to Optimization

Quadratic Form

For a symmetric positive definite matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ and a vector $\boldsymbol{b} \in \mathbb{R}^n$, solving the linear system $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ is equivalent to minimizing the quadratic function:

$$\phi(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T\mathbf{A}\boldsymbol{x} - \boldsymbol{x}^T\boldsymbol{b}$$

To see this, note that the gradient of ϕ is:

$$\nabla\phi(\boldsymbol{x}) = \mathbf{A}\boldsymbol{x} - \boldsymbol{b}$$

At the minimum of ϕ, we have $\nabla\phi(\boldsymbol{x}) = \mathbf{0}$, which gives us $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$. This optimization perspective is the foundation for developing the Conjugate Gradient method.

NOTE When $\mathbf{A}$ is symmetric positive definite, the quadratic function $\phi(\boldsymbol{x})$ has a unique global minimum, forming a paraboloid in $(n+1)$-dimensional space with elliptical contours in $\mathbb{R}^n$. The condition number of $\mathbf{A}$ determines how elongated these elliptical contours are—a high condition number creates a narrow, steep valley that causes difficulty for simple iterative methods.

Limitations of Stationary Methods

Before exploring the Conjugate Gradient method, let's understand the limitations of simpler approaches. Methods like Jacobi iteration belong to a class called *stationary iterative methods*, characterized by a fixed iteration matrix $\mathbf{T}$:

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

NOTE These methods suffer from several drawbacks: slow convergence that only reduces error by a constant factor per iteration, high sensitivity to the condition number with performance degrading as $\kappa(\mathbf{A})$ increases, fixed behavior that doesn't adapt based on information gained during iteration, and the typical requirement for far more than n iterations to solve an $n \times n$ system. These limitations motivated the development of more sophisticated approaches.

Steepest Descent

The method of steepest descent improves upon stationary methods by adaptively choosing both the direction and step size at each iteration:

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \alpha_k \boldsymbol{r}^{(k)}$$

where $\boldsymbol{r}^{(k)} = \boldsymbol{b} - \mathbf{A}\boldsymbol{x}^{(k)}$ (the residual) represents the negative gradient of ϕ at $\boldsymbol{x}^{(k)}$, and α_k is the optimal step size:

$$\alpha_k = \frac{(\boldsymbol{r}^{(k)})^T \boldsymbol{r}^{(k)}}{(\boldsymbol{r}^{(k)})^T \mathbf{A}\boldsymbol{r}^{(k)}}$$

While steepest descent adapts to the local geometry of the problem, it often exhibits "zigzagging" behavior in narrow valleys, making it inefficient for ill-conditioned systems.

4.3.2 A-Conjugate Directions

A-Conjugate Directions: The Key Insight

The key idea behind the Conjugate Gradient method is the concept of **A**-conjugate (or **A**-orthogonal) directions. A set of nonzero vectors $\{\boldsymbol{p}_i\}_{i=1}^n$ is **A**-conjugate if:

$$\boldsymbol{p}_i^T \mathbf{A} \boldsymbol{p}_j = 0, \quad \text{for all } i \neq j$$

These vectors have several crucial properties:

- They are linearly independent
- A set of n such vectors forms a basis for $\mathbb{R}^n$
- When minimizing along these directions, progress in one direction is never undone by moves in subsequent directions (due to linear independence)

NOTE

If we minimize $\phi(\boldsymbol{x})$ sequentially along a set of n **A**-conjugate directions, we will reach the exact minimum in at most n steps.

Optimization Along Conjugate Directions

If $\{\boldsymbol{p}_i\}_{i=0}^{n-1}$ is a set of **A**-conjugate vectors, the solution $\boldsymbol{x}^*$ to $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ can be expressed as:

$$\boldsymbol{x}^* = \boldsymbol{x}^{(0)} + \sum_{i=0}^{n-1} \alpha_i \boldsymbol{p}_i$$

where the optimal step sizes are:

$$\alpha_i = \frac{\boldsymbol{p}_i^T \boldsymbol{r}_i}{\boldsymbol{p}_i^T \mathbf{A} \boldsymbol{p}_i}$$

with $\boldsymbol{r}_i = \boldsymbol{b} - \mathbf{A}\boldsymbol{x}^{(i)}$.
What makes this approach powerful is that each step size α_i can be determined independently of the others. When using **A**-conjugate directions, the components of our optimization problem decouple, allowing for exact minimization in exactly n steps.

Iteratively Computing A-Conjugate Directions

The primary challenge is efficiently generating **A**-conjugate directions. While the eigenvectors of **A** would form an **A**-conjugate set, computing them is typically too expensive. The key innovation of the Conjugate Gradient method is a technique to generate these directions iteratively without storing them all. Each new direction is formed as:

$$\boldsymbol{p}_{k+1} = \boldsymbol{r}_{k+1} + \beta_k \boldsymbol{p}_k$$

where β_k is chosen to ensure $\boldsymbol{p}_{k+1}$ is **A**-conjugate to all previous directions:

$$\beta_k = \frac{\boldsymbol{r}_{k+1}^T \boldsymbol{r}_{k+1}}{\boldsymbol{r}_k^T \boldsymbol{r}_k}$$

This formula guarantees that each new direction is $\mathbf{A}$-conjugate to all previous directions, not just the most recent one. This property is a consequence of how the residuals and search directions evolve during the algorithm.

The Complete Conjugate Gradient Algorithm

Putting everything together, the Conjugate Gradient algorithm becomes:
Initialize:

- Choose initial guess $\boldsymbol{x}^{(0)}$
- Compute initial residual $\boldsymbol{r}^{(0)} = \boldsymbol{b} - \mathbf{A}\boldsymbol{x}^{(0)}$
- Set initial search direction $\boldsymbol{p}^{(0)} = \boldsymbol{r}^{(0)}$

For $k = 0, 1, 2, \ldots$ **until convergence:**

1. Compute optimal step size: $\alpha_k = \frac{\boldsymbol{r}^{(k)T}\boldsymbol{r}^{(k)}}{\boldsymbol{p}^{(k)T}\mathbf{A}\boldsymbol{p}^{(k)}}$
2. Update solution: $\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \alpha_k \boldsymbol{p}^{(k)}$
3. Update residual: $\boldsymbol{r}^{(k+1)} = \boldsymbol{r}^{(k)} - \alpha_k \mathbf{A}\boldsymbol{p}^{(k)}$
4. Check for convergence (if $\|\boldsymbol{r}^{(k+1)}\|$ is sufficiently small, stop)
5. Compute direction update coefficient: $\beta_k = \frac{\boldsymbol{r}^{(k+1)T}\boldsymbol{r}^{(k+1)}}{\boldsymbol{r}^{(k)T}\boldsymbol{r}^{(k)}}$
6. Update search direction: $\boldsymbol{p}^{(k+1)} = \boldsymbol{r}^{(k+1)} + \beta_k \boldsymbol{p}^{(k)}$

This algorithm has the following properties:

- It requires storing only a constant number of vectors
- In exact arithmetic, it converges in at most n steps
- Each iteration costs $O(n^2)$ operations (due to matrix-vector product)
- For sparse matrices, meaning matrices that are mostly populated with zeros, this cost can be much lower (using sparse data structures).

4.3.3 Convergence of Conjugate Gradient Method

Conjugate Gradient Termination

Let $\mathbf{A} \in \mathbb{R}^{n \times n}$ be symmetric positive definite. Then the Conjugate Gradient method applied to solve $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ will terminate with the exact solution in at most n iterations.
The proof for this theorem rests on two key properties that can be established by induction:

1. The residuals $\boldsymbol{r}^{(0)}, \boldsymbol{r}^{(1)}, \ldots, \boldsymbol{r}^{(k)}$ are mutually orthogonal
2. The search directions $\boldsymbol{p}^{(0)}, \boldsymbol{p}^{(1)}, \ldots, \boldsymbol{p}^{(k)}$ are mutually $\mathbf{A}$-conjugate

Since at most n mutually $\mathbf{A}$-conjugate directions can exist in $\mathbb{R}^n$, the algorithm must terminate in at most n steps, having explored the entire space.

Examples Convergence Behavior Comparison Consider a 100×100 symmetric positive definite matrix with eigenvalues clustered in two groups: 90 eigenvalues near 1 and 10 eigenvalues near 100.
The convergence rates for different methods on this system are:

- Jacobi iteration: ≈ 0.99 (thousands of iterations needed)
- Steepest descent: ≈ 0.98 (still very slow)
- Conjugate Gradient: ≈ 0.82 (dramatically faster)

The error in CG after k iterations is bounded by:

$$\|\boldsymbol{x}^{(k)} - \boldsymbol{x}^*\|_{\mathbf{A}} \leq 2 \left(\frac{\sqrt{\kappa(\mathbf{A})} - 1}{\sqrt{\kappa(\mathbf{A})} + 1} \right)^k \|\boldsymbol{x}^{(0)} - \boldsymbol{x}^*\|_{\mathbf{A}}$$

Furthermore, because of the eigenvalue clustering, CG would likely converge in just over 10 iterations rather than the theoretical maximum of 100.

Preconditioning: Further Acceleration While CG significantly outperforms simpler methods, its performance still degrades as $\kappa(\mathbf{A})$ increases. Preconditioning addresses this by transforming the original system into an equivalent one with better conditioning. Given a symmetric positive definite matrix $\mathbf{M}$ that approximates $\mathbf{A}$ but is easier to invert, we solve the equivalent system:

$$\mathbf{M}^{-1/2}\mathbf{A}\mathbf{M}^{-1/2}\boldsymbol{y} = \mathbf{M}^{-1/2}\boldsymbol{b}, \quad \text{where } \boldsymbol{y} = \mathbf{M}^{1/2}\boldsymbol{x}$$

The ideal preconditioner:

- Approximates $\mathbf{A}$ well, so $\mathbf{M}^{-1}\mathbf{A}$ has eigenvalues clustered near 1
- Is easy to invert
- Preserves the symmetry and positive-definiteness of the system

With $\mathbf{M} = \text{diag}(\mathbf{A})$, Preconditioned CG effectively combines Jacobi's diagonal scaling with CG's superior convergence properties.

4.4 Stability and Condition Numer

In an idealized mathematical setting, a square and invertible matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ admits a closed-form solution for the linear system $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$, given by

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

This expression is exact and demonstrates the theoretical uniqueness of the solution provided that $\mathbf{A}^{-1}$ exists.

Limitations of Analytical Methods

Computing $\mathbf{A}^{-1}$ directly is rarely practical for several reasons:

1. **Computational Expense**: Explicitly forming $\mathbf{A}^{-1}$ requires $O(n^3)$ operations, which becomes prohibitive for large systems.
2. **Numerical Instability**: Direct inversion can amplify round-off errors, especially when $\mathbf{A}$ is nearly singular.
3. **Inefficiency for Multiple Systems**: When solving multiple systems with the same coefficient matrix but different right-hand sides, storing and applying $\mathbf{A}^{-1}$ is less efficient than using factorization methods.
4. **Memory Requirements**: Storing the full inverse requires $O(n^2)$ memory, which may be unnecessary since we only need the solution vector $\boldsymbol{x}$.

Matrix Factorization Approach

So far, we have looked at methods that instead of computing $\mathbf{A}^{-1}$ explicitly, we factorize the matrix $\mathbf{A}$ into simpler components (LU, Cholesky etc.) that are easier to work with. Each triangular solve requires only $O(n^2)$ operations, making this approach substantially more efficient for solving multiple systems with the same coefficient matrix.

The key advantage of matrix factorization methods like LU decomposition is that the expensive $O(n^3)$ factorization step needs to be performed only once for a given matrix $\mathbf{A}$. After that, each new right-hand side $\boldsymbol{b}$ can be handled with just $O(n^2)$ operations.

4.4.1 Numerical Stability

Numerical Stability

Numerical stability refers to how well a computational algorithm controls the accumulation and propagation of round-off errors. An algorithm is considered numerically stable if small changes in the input data produce correspondingly small changes in the output.
In the context of solving linear systems:

- A stable algorithm should not amplify errors beyond what would be expected based on the conditioning of the problem
- Different algorithms for the same mathematical problem can have significantly different stability properties
- Stability is a property of the algorithm, not the problem itself

For example, Gaussian elimination with partial pivoting is generally stable for solving linear systems, while the explicit computation of $\mathbf{A}^{-1}$ followed by matrix-vector multiplication is often less stable.

4.4.2 Condition Number

Condition Number

The condition number of $\mathbf{A}$ with respect to a chosen matrix norm $\|\cdot\|$ is defined as

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

This scalar quantifies the sensitivity of the solution $\boldsymbol{x}$ to perturbations in $\mathbf{A}$ or $\boldsymbol{b}$. A large $\kappa(\mathbf{A})$ indicates that the system is *ill-conditioned*, meaning that even minute errors in the data may be significantly amplified in the computed solution.

For symmetric matrices, the condition number can be computed using the eigenvalues:

$$\kappa_2(\mathbf{A}) = \frac{|\lambda_{\max}|}{|\lambda_{\min}|}$$

where $\lambda_{\max}$ and $\lambda_{\min}$ are the eigenvalues with the largest and smallest absolute values, respectively.

The condition number provides an upper bound on how much perturbations in the input data can be amplified in the solution. Unlike numerical stability, which is a property of the algorithm, the condition number is an intrinsic property of the matrix itself.

NOTE

For example, consider the diagonal matrix

$$\mathbf{A} = \begin{pmatrix} 1 & 0 \\ 0 & 10^{-6} \end{pmatrix}.$$

Its eigenvalues are exactly 1 and 10^{-6} (since it's a diagonal matrix), yielding $\kappa_2(\mathbf{A}) = 10^6$. Such a high condition number suggests that small perturbations in the input data could be significantly amplified in the solution.

This sensitivity to perturbations affects all solution methods, whether direct or iterative. However, different factorization techniques (like QR versus LU) may have different numerical stability properties when applied to ill-conditioned problems.

Examples

Impact of Condition Number on Solution Sensitivity: Consider solving a linear system $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ where:

$$\mathbf{A} = \begin{pmatrix} 1 & 0 \\ 0 & 10^{-6} \end{pmatrix} \quad \text{and} \quad \boldsymbol{b} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

This is a diagonal matrix with eigenvalues $\lambda_1 = 1$ and $\lambda_2 = 10^{-6}$, so $\kappa_2(\mathbf{A}) = 10^6$.

The exact solution is $\boldsymbol{x} = \begin{pmatrix} 1 \\ 10^6 \end{pmatrix}$.

Note how the small eigenvalue $\lambda_2 = 10^{-6}$ corresponds to a very large entry in the solution vector. This illustrates why ill-conditioned matrices can lead to

numerical challenges - even small perturbations in the direction corresponding to the smallest eigenvalue can cause large changes in the solution.
Even with stable algorithms like LU decomposition with pivoting, the fundamental sensitivity of the problem remains, potentially leading to significant solution errors when the input data contains even slight inaccuracies.

4.4.3 Relationship Between Stability and Conditioning

Relationship Between Stability and Conditioning

It's important to distinguish between numerical stability and conditioning:

- **Condition Number**: Measures the inherent sensitivity of the mathematical problem to perturbations in the input data. It is a property of the matrix $\mathbf{A}$, not the algorithm used to solve the system.
- **Numerical Stability**: Refers to how well an algorithm controls the propagation of errors during computation. It is a property of the algorithm, not the mathematical problem.

A well-conditioned problem (low $\kappa(\mathbf{A})$) solved with an unstable algorithm may yield inaccurate results. Similarly, an ill-conditioned problem (high $\kappa(\mathbf{A})$) solved with a stable algorithm will still be sensitive to input perturbations.
The ideal scenario is to:

1. Reformulate ill-conditioned problems when possible to improve conditioning
2. Use numerically stable algorithms regardless of the condition number
3. Apply higher precision arithmetic for severely ill-conditioned problems

Iterative Methods and Conditioning

For very large systems where even computing and storing factors like $\mathbf{L}$ and $\mathbf{U}$ becomes impractical, iterative methods provide an alternative approach. These methods generate a sequence of approximations $\{\boldsymbol{x}^{(k)}\}_{k=0}^{\infty}$ starting from an initial guess $\boldsymbol{x}^{(0)}$. Under suitable conditions, this sequence converges to the true solution $\boldsymbol{x}$.

Condition Number and Convergence

The convergence rate of iterative methods is strongly influenced by the condition number:

- Well-conditioned systems (small $\kappa(\mathbf{A})$) generally converge quickly
- Ill-conditioned systems (large $\kappa(\mathbf{A})$) may converge very slowly or fail to converge in practice

To address this challenge, preconditioning techniques are often employed. A preconditioner transforms the original system into an equivalent system with a more favorable condition number, accelerating convergence while maintaining the same solution.

Stability in Matrix Factorizations

Different matrix factorization methods exhibit varying degrees of numerical stability:

- **LU Decomposition**: In its basic form without pivoting, LU can be unstable. With partial pivoting (row exchanges), it becomes much more stable for most matrices. Complete pivoting (row and column exchanges) offers even better stability but at a higher computational cost.
- **Cholesky Decomposition**: For symmetric positive definite matrices, Cholesky is both efficient and numerically stable. No pivoting is required since diagonal elements are guaranteed to be positive.
- **QR Decomposition**: Generally more stable than LU decomposition, especially for ill-conditioned problems, but comes at a higher computational cost. The orthogonality of the **Q** factor helps maintain numerical stability.
- **SVD (Singular Value Decomposition)**: The most stable decomposition, capable of handling even rank-deficient matrices, but also the most computationally intensive.

The stability of these factorizations relates directly to how they handle the sensitive directions associated with small eigenvalues or singular values.

5 Advanced Matrix Decompositions

5.1 Singular Value Decomposition and the Four Fundamental Subspaces

The SVD is often regarded as the culmination of linear algebra concepts. It formalizes the intuition that any matrix transformation can be decomposed into rotations and scalings. The orthogonal matrices $\mathbf{U}$ and $\mathbf{V}$ represent rotations (or reflections) in the input and output spaces, while the diagonal matrix $\mathbf{\Sigma}$ represents stretching or compression along orthogonal axes.

Singular Value Decomposition (SVD)

For any matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, the **Singular Value Decomposition (SVD)** expresses $\mathbf{A}$ as

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

where $\mathbf{U} \in \mathbb{R}^{m \times m}$ and $\mathbf{V} \in \mathbb{R}^{n \times n}$ are orthogonal matrices, and $\mathbf{\Sigma} \in \mathbb{R}^{m \times n}$ is a diagonal (or more precisely, a rectangular diagonal) matrix whose diagonal entries $\sigma_1 \geq \sigma_2 \geq \cdots \geq 0$ are called the *singular values* of $\mathbf{A}$.

NOTE

At its core, any matrix operation can be understood geometrically as a composition of three fundamental actions: rotation, scaling, and another rotation. When a matrix $\mathbf{A}$ acts on a vector $\boldsymbol{x}$, it can be viewed as:

1. First, rotating the vector (changing its direction)
2. Then, scaling the vector along specific axes (changing its length in different directions)
3. Finally, rotating again to achieve the final orientation

This geometric interpretation is powerful because it allows us to visualize the action of complex transformations. For example, the matrix $\mathbf{A} = \begin{pmatrix} 3 & 1 \\ 1 & 2 \end{pmatrix}$ doesn't obviously reveal its geometric effect, but when decomposed into rotations and scalings, we can clearly see it consists of a rotation by approximately 22.5 degrees, followed by scaling along new axes by factors of 3.41 and 1.59, and then another rotation.

The Four Fundamental Subspaces

Given a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, the four fundamental subspaces are:

1. **Column Space (Range):** $\mathcal{R}(\mathbf{A}) = \{\mathbf{A}\boldsymbol{x} : \boldsymbol{x} \in \mathbb{R}^n\} \subseteq \mathbb{R}^m$. This subspace consists of all vectors that can be expressed as $\mathbf{A}\boldsymbol{x}$.
2. **Null Space (Kernel):** $\mathcal{N}(\mathbf{A}) = \{\boldsymbol{x} \in \mathbb{R}^n : \mathbf{A}\boldsymbol{x} = \mathbf{0}\}$. This subspace comprises all solutions to the homogeneous equation.
3. **Row Space:** $\mathcal{R}(\mathbf{A}^T) = \{\mathbf{A}^T\boldsymbol{y} : \boldsymbol{y} \in \mathbb{R}^m\} \subseteq \mathbb{R}^n$. It is the span of the rows of $\mathbf{A}$ and is the orthogonal complement of $\mathcal{N}(\mathbf{A})$ in $\mathbb{R}^n$.

4. **Left Null Space:** $\mathcal{N}(\mathbf{A}^T) = \{\boldsymbol{y} \in \mathbb{R}^m : \mathbf{A}^T\boldsymbol{y} = \mathbf{0}\}$. This is the set of all vectors in $\mathbb{R}^m$ that are orthogonal to every column of $\mathbf{A}$.

NOTE

Link to SVD: In the SVD $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$, the columns of $\mathbf{U}$ and $\mathbf{V}$ provide orthonormal bases for the four fundamental subspaces. The SVD expresses the matrix $\mathbf{A}$ in terms of these subspaces as follows:

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T = \begin{bmatrix} | & | & | & | & & | \\ | & | & | & | & & | \\ \boldsymbol{u}_1 & \boldsymbol{u}_2 & \cdots & \boldsymbol{u}_r & \cdots & \boldsymbol{u}_m \\ | & | & | & | & & | \\ | & | & | & | & & | \end{bmatrix} \begin{bmatrix} \sigma_1 & & & & & \\ & \sigma_2 & & & & \\ & & \ddots & & & \\ & & & \sigma_r & & \\ & & & & \ddots & \\ & & & & & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{v}_1^T \\ \boldsymbol{v}_2^T \\ \vdots \\ \boldsymbol{v}_r^T \\ \vdots \\ \boldsymbol{v}_n^T \end{bmatrix}$$

Where r is the rank of matrix $\mathbf{A}$. The four fundamental subspaces can be directly identified from this decomposition:

- **Column Space** $\mathcal{R}(\mathbf{A})$: Spanned by $\{\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_r\}$ (first r columns of $\mathbf{U}$)
- **Left Null Space** $\mathcal{N}(\mathbf{A}^T)$: Spanned by $\{\boldsymbol{u}_{r+1}, \boldsymbol{u}_{r+2}, \ldots, \boldsymbol{u}_m\}$ (last $m - r$ columns of $\mathbf{U}$)
- **Row Space** $\mathcal{R}(\mathbf{A}^T)$: Spanned by $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_r\}$ (first r columns of $\mathbf{V}$)
- **Null Space** $\mathcal{N}(\mathbf{A})$: Spanned by $\{\boldsymbol{v}_{r+1}, \boldsymbol{v}_{r+2}, \ldots, \boldsymbol{v}_n\}$ (last $n - r$ columns of $\mathbf{V}$)

This elegant connection reveals why the SVD is so powerful: it simultaneously provides orthonormal bases for all four fundamental subspaces associated with the matrix.

Zero Singular Values and Subspaces

The number of zero singular values in $\mathbf{\Sigma}$ directly corresponds to the dimension of the null space $\mathcal{N}(\mathbf{A})$. If we have $n - r$ zero singular values, then:

$$\dim(\mathcal{N}(\mathbf{A})) = n - r$$

Similarly, the number of non-zero singular values equals the dimension of the range space:

$$\dim(\mathcal{R}(\mathbf{A})) = r$$

This establishes the fundamental relationship between the rank of $\mathbf{A}$, the dimension of its null space, and the singular values:

$$\text{rank}(\mathbf{A}) + \dim(\mathcal{N}(\mathbf{A})) = n$$

where n is the number of columns in $\mathbf{A}$.

NOTE The zero singular values provide critical information about the linear transformation represented by $\mathbf{A}$. Each zero singular value signifies a direction in the input space (given by the corresponding right singular vector $\boldsymbol{v}_i$) that gets mapped to the zero vector. These directions collectively form the null space of $\mathbf{A}$.
Conversely, the positive singular values correspond to directions that are preserved by the transformation, though possibly scaled. The smallest non-zero singular value σ_r indicates the direction along which the transformation compresses vectors the most, making it crucial for understanding the numerical stability of problems involving $\mathbf{A}$.

Examples Consider a matrix $\mathbf{A} \in \mathbb{R}^{3\times 4}$ with rank 2. Its SVD would yield:

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T = \begin{bmatrix} \boldsymbol{u}_1 & \boldsymbol{u}_2 & \boldsymbol{u}_3 \end{bmatrix} \begin{bmatrix} \sigma_1 & 0 & 0 & 0 \\ 0 & \sigma_2 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{v}_1^T \\ \boldsymbol{v}_2^T \\ \boldsymbol{v}_3^T \\ \boldsymbol{v}_4^T \end{bmatrix}$$

Here, $\sigma_1 > 0$ and $\sigma_2 > 0$, while all other singular values are zero. The column space $\mathcal{R}(\mathbf{A})$ is two-dimensional, spanned by $\{\boldsymbol{u}_1, \boldsymbol{u}_2\}$. The null space $\mathcal{N}(\mathbf{A})$ is also two-dimensional, spanned by $\{\boldsymbol{v}_3, \boldsymbol{v}_4\}$. Any vector in the null space, when multiplied by $\mathbf{A}$, produces the zero vector because the corresponding singular values are zero.

Link to Solutions of Linear Equations Consider the linear system $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$. The existence and uniqueness of solutions depend on the relationship between $\boldsymbol{b}$ and the fundamental subspaces:

- A solution exists if and only if $\boldsymbol{b} \in \mathcal{R}(\mathbf{A})$.
- When $\boldsymbol{b} \in \mathcal{R}(\mathbf{A})$, the general solution can be expressed as the sum of a particular solution and any vector in the null space $\mathcal{N}(\mathbf{A})$.
- In the context of least squares, when $\boldsymbol{b}$ is not in $\mathcal{R}(\mathbf{A})$, one seeks a solution $\hat{\boldsymbol{x}}$ such that the residual $\|\mathbf{A}\hat{\boldsymbol{x}} - \boldsymbol{b}\|$ is minimized. The SVD provides a natural framework to compute the minimum-norm solution via the pseudoinverse.

NOTE The SVD not only decomposes the matrix into its fundamental components but also organizes its action into rotations (via $\mathbf{U}$ and $\mathbf{V}$) and scalings (via $\mathbf{\Sigma}$). This clear separation allows us to understand how errors propagate in the solution of linear systems and how the geometry of $\mathbf{A}$ influences the solvability and stability of $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$.

5.2 Rank Truncation via the SVD

Rank Truncation

Let r be an integer with $0 \leq r \leq p$. The **rank-r approximation** of $\mathbf{A}$ is defined as

$$\mathbf{A}_r = \sum_{i=1}^{r} \sigma_i \boldsymbol{u}_i \boldsymbol{v}_i^T,$$

or equivalently, if we partition $\mathbf{U}$, $\boldsymbol{\Sigma}$, and $\mathbf{V}$ as

$$\mathbf{U} = \begin{bmatrix} \mathbf{U}_r & \mathbf{U}_{r,\perp} \end{bmatrix}, \quad \boldsymbol{\Sigma} = \begin{bmatrix} \boldsymbol{\Sigma}_r & 0 \\ 0 & \boldsymbol{\Sigma}_{r,\perp} \end{bmatrix}, \quad \mathbf{V} = \begin{bmatrix} \mathbf{V}_r & \mathbf{V}_{r,\perp} \end{bmatrix},$$

then

$$\mathbf{A}_r = \mathbf{U}_r \boldsymbol{\Sigma}_r \mathbf{V}_r^T.$$

Here, $\boldsymbol{\Sigma}_r \in \mathbb{R}^{r \times r}$ contains the largest r singular values, while $\mathbf{U}_r$ and $\mathbf{V}_r$ contain the corresponding singular vectors. In this form, $\mathbf{U}_r$ consists of the first r columns of $\mathbf{U}$, and $\mathbf{V}_r$ consists of the first r columns of $\mathbf{V}$.

NOTE

Eckart-Young Theorem: This truncated SVD, $\mathbf{A}_r$, is the best approximation to $\mathbf{A}$ among all matrices of rank at most r. Specifically, for any matrix $\mathbf{B}$ of rank at most r,

$$\|\mathbf{A} - \mathbf{A}_r\|_2 \leq \|\mathbf{A} - \mathbf{B}\|_2 \quad \text{and} \quad \|\mathbf{A} - \mathbf{A}_r\|_F \leq \|\mathbf{A} - \mathbf{B}\|_F.$$

Moreover, the errors are given by

$$\|\mathbf{A} - \mathbf{A}_r\|_2 = \sigma_{r+1}, \quad \text{and} \quad \|\mathbf{A} - \mathbf{A}_r\|_F = \sqrt{\sum_{i=r+1}^{p} \sigma_i^2}.$$

To see the derivation, observe that the SVD of $\mathbf{A}$ gives

$$\mathbf{A} = \sum_{i=1}^{p} \sigma_i \boldsymbol{u}_i \boldsymbol{v}_i^T.$$

Truncating the series at r terms results in

$$\mathbf{A}_r = \sum_{i=1}^{r} \sigma_i \boldsymbol{u}_i \boldsymbol{v}_i^T,$$

and the remainder is

$$\mathbf{A} - \mathbf{A}_r = \sum_{i=r+1}^{p} \sigma_i \boldsymbol{u}_i \boldsymbol{v}_i^T.$$

Since the spectral norm (or 2-norm) is given by the largest singular value, it follows that

$$\|\mathbf{A} - \mathbf{A}_r\|_2 = \sigma_{r+1}.$$

Any other rank-r approximation $\mathbf{B}$ must incur an error at least as large as this, establishing the optimality of $\mathbf{A}_r$.

Examples

Let's examine a practical application of rank truncation in data compression. Consider a grayscale image represented as a matrix $\mathbf{A} \in \mathbb{R}^{1000\times1000}$ with singular values that decay rapidly after the first 50 values.
After computing the SVD, we can construct a rank-50 approximation $\mathbf{A}_{50}$. The storage requirements are dramatically reduced:

- Original matrix: $1000 \times 1000 = 10^6$ elements
- Rank-50 representation: $1000 \times 50 + 50 + 50 \times 1000 = 10^5 + 50$ elements

This represents approximately a 10x compression ratio while preserving the most important features of the image. The error bounds tell us that:

- Worst-case pixel error: $\|\mathbf{A} - \mathbf{A}_{50}\|_2 = \sigma_{51}$
- Average error across all pixels: $\|\mathbf{A} - \mathbf{A}_{50}\|_F = \sqrt{\sum_{i=51}^{1000} \sigma_i^2}$

If $\sigma_{51} \ll \sigma_{50}$, the compression preserves image quality well, demonstrating how the SVD can identify and retain the most significant components of the data.

NOTE

The optimal approximation property of truncated SVD makes it an invaluable tool across various applications, including image processing, signal denoising, latent semantic analysis in text processing, and recommender systems. The error bounds provided by the Eckart-Young theorem allow practitioners to make informed decisions about the trade-off between approximation quality and computational complexity. In practical terms, examining the distribution of singular values can reveal the intrinsic dimensionality of the data, with a sharp drop in singular values indicating a natural cutoff point for rank truncation.

Relationship Between SVD and Eigendecomposition

For any matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$, the singular values and vectors are related to the eigenvalues and eigenvectors of $\mathbf{A}^T\mathbf{A}$ and $\mathbf{A}\mathbf{A}^T$ as follows:

- The right singular vectors $\boldsymbol{v}_i$ are eigenvectors of $\mathbf{A}^T\mathbf{A}$: $\mathbf{A}^T\mathbf{A}\boldsymbol{v}_i = \sigma_i^2\boldsymbol{v}_i$
- The left singular vectors $\boldsymbol{u}_i$ are eigenvectors of $\mathbf{A}\mathbf{A}^T$: $\mathbf{A}\mathbf{A}^T\boldsymbol{u}_i = \sigma_i^2\boldsymbol{u}_i$
- The singular values σ_i are the square roots of the eigenvalues of both $\mathbf{A}^T\mathbf{A}$ and $\mathbf{A}\mathbf{A}^T$

This connection means that zero singular values correspond directly to zero eigenvalues of $\mathbf{A}^T\mathbf{A}$ and $\mathbf{A}\mathbf{A}^T$.

NOTE The pseudoinverse, derived from the SVD as $\mathbf{A}^+ = \mathbf{V}\boldsymbol{\Sigma}^+\mathbf{U}^T$ (where $\boldsymbol{\Sigma}^+$ has reciprocals of non-zero singular values on its diagonal), provides a unified framework for solving both overdetermined and underdetermined systems. For overdetermined systems, $\hat{\boldsymbol{x}} = \mathbf{A}^+\boldsymbol{b}$ minimizes $\|\mathbf{A}\boldsymbol{x} - \boldsymbol{b}\|_2$, while for underdetermined systems, it gives the solution with the smallest $\|\boldsymbol{x}\|_2$ among all vectors satisfying $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$.

5.3 Condition Number and SVD

Condition Number via SVD

The condition number of a matrix $\mathbf{A}$, denoted $\kappa(\mathbf{A})$, measures the sensitivity of the solution of the linear system $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ to perturbations in the input data. Given the SVD $\mathbf{A} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T$, the condition number is defined as:

$$\kappa(\mathbf{A}) = \frac{\sigma_1}{\sigma_r}$$

where σ_1 is the largest singular value and σ_r is the smallest non-zero singular value (assuming $\mathbf{A}$ has rank r).

To derive this definition from first principles, recall that the condition number measures how much a relative change in the input can affect the relative change in the output. For a linear system $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$, consider a perturbation $\delta\boldsymbol{b}$ in $\boldsymbol{b}$ leading to a change $\delta\boldsymbol{x}$ in the solution. The relative error magnification is bounded by:

$$\frac{\|\delta\boldsymbol{x}\|/\|\boldsymbol{x}\|}{\|\delta\boldsymbol{b}\|/\|\boldsymbol{b}\|} \leq \kappa(\mathbf{A})$$

Using the SVD $\mathbf{A} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T$, we can write:

$$\boldsymbol{x} = \mathbf{A}^{-1}\boldsymbol{b} = \mathbf{V}\boldsymbol{\Sigma}^{-1}\mathbf{U}^T\boldsymbol{b}$$

The worst-case amplification occurs when $\boldsymbol{b}$ is aligned with the left singular vector $\boldsymbol{u}_r$ (corresponding to the smallest singular value) and the resulting $\boldsymbol{x}$ is aligned with the right singular vector $\boldsymbol{v}_r$. In this case:

$$\boldsymbol{x} = \frac{1}{\sigma_r}\boldsymbol{v}_r$$

Conversely, the least amplification occurs when $\boldsymbol{b}$ aligns with $\boldsymbol{u}_1$ (the largest singular value):

$$\boldsymbol{x} = \frac{1}{\sigma_1}\boldsymbol{v}_1$$

The ratio between these extreme cases gives the condition number:

$$\kappa(\mathbf{A}) = \frac{\sigma_1}{\sigma_r}$$

This reveals why matrices with very small singular values relative to their largest singular value are ill-conditioned.

Examples Consider a linear system $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ where:

$$\mathbf{A} = \begin{bmatrix} 1 & 1 \\ 1 & 1.001 \end{bmatrix}$$

Computing the SVD yields:

$$\mathbf{U} = \begin{bmatrix} -0.7071 & -0.7071 \\ -0.7071 & 0.7071 \end{bmatrix}, \quad \mathbf{\Sigma} = \begin{bmatrix} 2.001 & 0 \\ 0 & 0.0005 \end{bmatrix}, \quad \mathbf{V} = \begin{bmatrix} -0.7071 & -0.7071 \\ -0.7071 & 0.7071 \end{bmatrix}$$

The condition number is $\kappa(\mathbf{A}) = \frac{\sigma_1}{\sigma_2} = \frac{2.001}{0.0005} \approx 4000$.

Now suppose we have $\boldsymbol{b} = \begin{bmatrix} 2 \\ 2.001 \end{bmatrix}$ and a slightly perturbed $\tilde{\boldsymbol{b}} = \begin{bmatrix} 2 \\ 2.002 \end{bmatrix}$. To understand the effect of this perturbation, we express these vectors in the basis of left singular vectors:

$$\boldsymbol{b} = \mathbf{U}^T\boldsymbol{b} = \begin{bmatrix} -2.8248 \\ 0.0007 \end{bmatrix}, \quad \tilde{\boldsymbol{b}} = \mathbf{U}^T\tilde{\boldsymbol{b}} = \begin{bmatrix} -2.8255 \\ 0.0014 \end{bmatrix}$$

The solution can be computed as $\boldsymbol{x} = \mathbf{V}\mathbf{\Sigma}^{-1}\mathbf{U}^T\boldsymbol{b}$. The key observation is that components corresponding to small singular values get amplified by the factor $1/\sigma_i$:

$$\boldsymbol{x} = \mathbf{V}\begin{bmatrix} \frac{-2.8248}{2.001} \\ \frac{0.0007}{0.0005} \end{bmatrix} = \mathbf{V}\begin{bmatrix} -1.41 \\ 1.40 \end{bmatrix} \approx \begin{bmatrix} 1.00 \\ 1.00 \end{bmatrix}$$

$$\tilde{\boldsymbol{x}} = \mathbf{V}\begin{bmatrix} \frac{-2.8255}{2.001} \\ \frac{0.0014}{0.0005} \end{bmatrix} = \mathbf{V}\begin{bmatrix} -1.41 \\ 2.80 \end{bmatrix} \approx \begin{bmatrix} -0.01 \\ 1.99 \end{bmatrix}$$

We see that a tiny change in the input ($\|\delta\boldsymbol{b}\|/\|\boldsymbol{b}\| \approx 0.0005$) creates a large relative change in the output ($\|\delta\boldsymbol{x}\|/\|\boldsymbol{x}\| \approx 0.70$), which is consistent with our condition number. The component of the perturbation in the direction of $\boldsymbol{u}_2$ (corresponding to the smallest singular value σ_2) gets amplified by $1/\sigma_2 = 1/0.0005 = 2000$.

This SVD analysis reveals precisely why the system is sensitive to perturbations, and in which directions these perturbations are most amplified.

SVD and Regularization For ill-conditioned problems, regularization techniques leverage the SVD to create more stable solutions. A key approach is the truncated SVD, where we replace $\mathbf{A}$ with a low-rank approximation $\mathbf{A}_k$ by setting all singular values below a threshold to zero. The SVD solution becomes:

$$\boldsymbol{x}_k = \sum_{i=1}^{k} \frac{\boldsymbol{u}_i^T \boldsymbol{b}}{\sigma_i} \boldsymbol{v}_i$$

This effectively ignores directions corresponding to small singular values, reducing sensitivity to perturbations at the cost of introducing a controlled amount of bias. The condition number of this regularized system is $\kappa(\mathbf{A}_k) = \sigma_1/\sigma_k$, which is much smaller than the original.

NOTE The singular value spectrum of a matrix reveals much about the problem's inherent difficulty. A rapidly decaying spectrum (many small singular values) indicates an ill-conditioned problem where information is being compressed in certain directions. In such cases, the SVD provides not just a diagnostic tool but also the means for regularization through methods like truncated SVD, ridge regularization, and other approaches that modify the singular values to balance accuracy against stability.

5.4 Matrix Norms

Matrix norms are functions that assign a scalar "size" or "magnitude" to a matrix. They are fundamental tools in linear algebra and numerical analysis, used for measuring the error in matrix computations (e.g., in low-rank approximations or solutions to linear systems), analyzing the convergence of iterative algorithms, and understanding the sensitivity of problems to perturbations (e.g., condition numbers).

Matrix Norm A function $\|\cdot\| : \mathbb{R}^{m\times n} \to \mathbb{R}$ is a matrix norm if for all matrices $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{m\times n}$ and any scalar $c \in \mathbb{R}$, it satisfies the following properties:

1. Non-negativity: $\|\mathbf{A}\| \geq 0$
2. Definiteness: $\|\mathbf{A}\| = 0 \iff \mathbf{A} = \mathbf{0}$ (the zero matrix)
3. Absolute homogeneity: $\|c\mathbf{A}\| = |c|\|\mathbf{A}\|$
4. Triangle inequality (or subadditivity): $\|\mathbf{A} + \mathbf{B}\| \leq \|\mathbf{A}\| + \|\mathbf{B}\|$

For square matrices ($\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n\times n}$), an additional desirable property for many matrix norms is submultiplicativity:

5. Submultiplicativity: $\|\mathbf{AB}\| \leq \|\mathbf{A}\|\|\mathbf{B}\|$

This property is crucial when dealing with products of matrices, such as in iterative methods or error propagation.

Induced (Operator) Norms Induced norms, also known as operator norms, are defined in terms of vector norms. Given a vector norm $\|\cdot\|_v$ on $\mathbb{R}^n$ (domain) and a vector norm $\|\cdot\|_u$ on

$\mathbb{R}^m$ (codomain), the matrix norm for $\mathbf{A} \in \mathbb{R}^{m \times n}$ induced by these vector norms is:

$$\|\mathbf{A}\|_{u,v} = \max_{\mathbf{x} \neq \mathbf{0}} \frac{\|\mathbf{A}\mathbf{x}\|_u}{\|\mathbf{x}\|_v} = \max_{\|\mathbf{x}\|_v = 1} \|\mathbf{A}\mathbf{x}\|_u$$

Effectively, $\|\mathbf{A}\|_{u,v}$ measures the maximum "stretching factor" that $\mathbf{A}$ applies to any non-zero vector $\mathbf{x}$ (when measured by the respective vector norms). All induced norms are submultiplicative (assuming compatible dimensions and the same vector norm for the inner product when chaining matrices). Common induced norms use p-norms for vectors:

- **L_1-Norm (Maximum Absolute Column Sum):** Induced by the vector L_1-norm ($\|\mathbf{x}\|_1 = \sum_i |x_i|$).

$$\|\mathbf{A}\|_1 = \max_{1 \leq j \leq n} \sum_{i=1}^{m} |a_{ij}|$$

- **L_∞-Norm (Maximum Absolute Row Sum):** Induced by the vector L_∞-norm ($\|\mathbf{x}\|_\infty = \max_i |x_i|$).

$$\|\mathbf{A}\|_\infty = \max_{1 \leq i \leq m} \sum_{j=1}^{n} |a_{ij}|$$

- **L_2-Norm (Spectral Norm):** Induced by the vector L_2-norm (Euclidean norm, $\|\mathbf{x}\|_2 = \sqrt{\sum_i x_i^2}$).

$$\|\mathbf{A}\|_2 = \sigma_{\max}(\mathbf{A}) = \sqrt{\lambda_{\max}(\mathbf{A}^T\mathbf{A})}$$

  Here, $\sigma_{\max}(\mathbf{A})$ is the largest singular value of $\mathbf{A}$, and $\lambda_{\max}(\mathbf{A}^T\mathbf{A})$ is the largest eigenvalue of the positive semi-definite matrix $\mathbf{A}^T\mathbf{A}$. This norm represents the maximum factor by which $\mathbf{A}$ can stretch a vector in the Euclidean sense.

Entry-wise Norms

Entry-wise norms treat the matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ as a vector in $\mathbb{R}^{mn}$ containing all its entries, and then apply a vector norm to this long vector.

- **Frobenius Norm ($\|\cdot\|_F$):** This is analogous to the Euclidean (L_2) norm for vectors.

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

  The Frobenius norm can also be expressed using the trace of $\mathbf{A}^T\mathbf{A}$ or the singular values $\sigma_k(\mathbf{A})$ of $\mathbf{A}$:

$$\|\mathbf{A}\|_F = \sqrt{\mathrm{Tr}(\mathbf{A}^T\mathbf{A})} = \sqrt{\mathrm{Tr}(\mathbf{A}\mathbf{A}^T)} = \sqrt{\sum_{k=1}^{\min(m,n)} \sigma_k^2(\mathbf{A})}$$

The Frobenius norm is submultiplicative and is frequently used in numerical linear algebra, particularly in problems involving low-rank approximation (like PCA).

- **Max Norm ($\|\cdot\|_{\max}$):** (Less common in theoretical analysis but sometimes used)
$$\|\mathbf{A}\|_{\max} = \max_{i,j} |a_{ij}|$$
This is a valid matrix norm but is not submultiplicative.

Examples Matrix Norm Calculations Let $\mathbf{A} = \begin{pmatrix} 1 & -2 \\ -3 & 4 \end{pmatrix}$.

- **L_1-Norm:** $\|\mathbf{A}\|_1 = \max(|1| + |-3|, |-2| + |4|) = \max(1+3, 2+4) = \max(4, 6) = 6$.
- **L_∞-Norm:** $\|\mathbf{A}\|_\infty = \max(|1| + |-2|, |-3| + |4|) = \max(1+2, 3+4) = \max(3, 7) = 7$.
- **Frobenius Norm:** $\|\mathbf{A}\|_F = \sqrt{1^2 + (-2)^2 + (-3)^2 + 4^2} = \sqrt{1+4+9+16} = \sqrt{30} \approx 5.477$.
- **L_2-Norm (Spectral Norm):** We need eigenvalues of $\mathbf{A}^T\mathbf{A}$.
$$\mathbf{A}^T\mathbf{A} = \begin{pmatrix} 1 & -3 \\ -2 & 4 \end{pmatrix} \begin{pmatrix} 1 & -2 \\ -3 & 4 \end{pmatrix} = \begin{pmatrix} 10 & -14 \\ -14 & 20 \end{pmatrix}$$
The characteristic equation is $\det(\mathbf{A}^T\mathbf{A} - \lambda\mathbf{I}) = (10-\lambda)(20-\lambda) - (-14)^2 = \lambda^2 - 30\lambda + 4 = 0$. The eigenvalues are $\lambda = \frac{30 \pm \sqrt{30^2 - 4(1)(4)}}{2} = \frac{30 \pm \sqrt{900-16}}{2} = \frac{30 \pm \sqrt{884}}{2} = 15 \pm \sqrt{221}$. So, $\lambda_{\max}(\mathbf{A}^T\mathbf{A}) = 15 + \sqrt{221}$. Therefore, $\|\mathbf{A}\|_2 = \sqrt{15 + \sqrt{221}} \approx \sqrt{15 + 14.866} \approx \sqrt{29.866} \approx 5.465$.

- **Equivalence of Norms:** All matrix norms on the finite-dimensional space $\mathbb{R}^{m \times n}$ are equivalent. This means that for any two norms $\|\cdot\|_\alpha$ and $\|\cdot\|_\beta$, there exist positive constants c_1, c_2 such that $c_1\|\mathbf{A}\|_\beta \leq \|\mathbf{A}\|_\alpha \leq c_2\|\mathbf{A}\|_\beta$ for all $\mathbf{A} \in \mathbb{R}^{m \times n}$. This implies that if a sequence of matrices converges in one norm, it converges in all norms.
- **Consistency with Vector Norms:** A matrix norm $\|\cdot\|$ is said to be consistent with a vector norm $\|\cdot\|_v$ if $\|\mathbf{Ax}\|_v \leq \|\mathbf{A}\|\|\mathbf{x}\|_v$ for all $\mathbf{A}, \mathbf{x}$. Induced norms are, by definition, consistent with the vector norms that induce them. The Frobenius norm is consistent with the vector L_2-norm.

- **Condition Number:** Matrix norms are used to define the condition number of a square invertible matrix $\mathbf{A}$ as $\kappa(\mathbf{A}) = \|\mathbf{A}\|\|\mathbf{A}^{-1}\|$. The condition number measures the sensitivity of the solution of $\mathbf{Ax} = \mathbf{b}$ to perturbations in $\mathbf{A}$ or $\mathbf{b}$.
- **Applications:** The Frobenius norm is prominently featured in the Eckart-Young-Mirsky theorem for optimal low-rank matrix approximation (relevant to PCA error). The spectral norm is crucial in stability analysis of dynamical systems and convergence rates of iterative methods.

5.5 Overdetermined Systems and Least Squares

So far, we have focused on solving square systems of linear equations $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ where $\mathbf{A} \in \mathbb{R}^{n\times n}$. In many real-world applications, however, we encounter overdetermined systems where we have more equations than unknowns, leading to systems that typically have no exact solution. The method of least squares provides a powerful approach to finding the "best" approximate solution to such systems.

5.5.1 Overdetermined Systems

Overdetermined Systems and Least Squares

An overdetermined linear system has the form

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

where $\mathbf{A} \in \mathbb{R}^{m\times n}$ with $m > n$ and $\boldsymbol{b} \in \mathbb{R}^m$. Since $\mathbf{A}$ maps from $\mathbb{R}^n$ to $\mathbb{R}^m$, its range is at most n-dimensional. When $m > n$ and $\boldsymbol{b}$ is a general vector in $\mathbb{R}^m$, it typically lies outside the range of $\mathbf{A}$, meaning no exact solution exists.

5.5.2 Least Squares

Least Squares

The **least squares problem** addresses this by finding the vector $\hat{\boldsymbol{x}}$ that minimizes the sum of squared residuals:

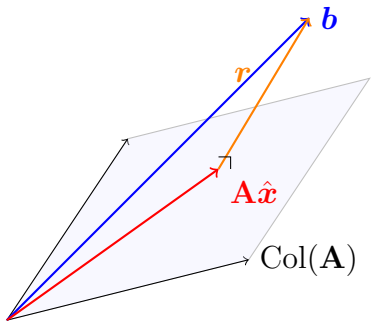
$$\hat{\boldsymbol{x}} = \arg\min_{\boldsymbol{x}\in\mathbb{R}^n} \|\mathbf{A}\boldsymbol{x} - \boldsymbol{b}\|_2^2$$

This formulation seeks the $\boldsymbol{x}$ that makes $\mathbf{A}\boldsymbol{x}$ as close as possible to $\boldsymbol{b}$ in the Euclidean norm sense.

NOTE

Overdetermined systems arise naturally in data fitting, statistical regression, signal processing, and many scientific applications where we have more measurements (equations) than parameters (unknowns). For example, when fitting a line to a set of data points, we typically have many more points than the two parameters needed to define a line.

Geometrically, the least squares solution represents the orthogonal projection of $\boldsymbol{b}$ onto the column space of $\mathbf{A}$. When $\mathbf{A}\hat{\boldsymbol{x}}$ is this projection, the residual vector $\boldsymbol{r} = \boldsymbol{b} - \mathbf{A}\hat{\boldsymbol{x}}$ is orthogonal to the column space, making it the shortest possible residual vector.



Deriving the Normal Equations

To find the minimizer of $f(\boldsymbol{x}) = \|\mathbf{A}\boldsymbol{x} - \boldsymbol{b}\|_2^2$, we can expand this expression:

$$f(\boldsymbol{x}) = (\mathbf{A}\boldsymbol{x} - \boldsymbol{b})^T(\mathbf{A}\boldsymbol{x} - \boldsymbol{b}) = \boldsymbol{x}^T\mathbf{A}^T\mathbf{A}\boldsymbol{x} - 2\boldsymbol{b}^T\mathbf{A}\boldsymbol{x} + \boldsymbol{b}^T\boldsymbol{b}$$

Since $\boldsymbol{b}^T\boldsymbol{b}$ is constant with respect to $\boldsymbol{x}$, we can focus on minimizing:

$$g(\boldsymbol{x}) = \boldsymbol{x}^T\mathbf{A}^T\mathbf{A}\boldsymbol{x} - 2\boldsymbol{b}^T\mathbf{A}\boldsymbol{x}$$

Setting the gradient to zero (a necessary condition for a minimum):

$$\nabla_{\boldsymbol{x}} g(\boldsymbol{x}) = 2\mathbf{A}^T\mathbf{A}\boldsymbol{x} - 2\mathbf{A}^T\boldsymbol{b} = \mathbf{0}$$

This leads to the **normal equations**:

$$\mathbf{A}^T\mathbf{A}\boldsymbol{x} = \mathbf{A}^T\boldsymbol{b}$$

When $\mathbf{A}$ has full column rank (i.e., its columns are linearly independent), $\mathbf{A}^T\mathbf{A}$ is invertible, and the unique least squares solution is:

$$\hat{\boldsymbol{x}} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\boldsymbol{b}$$

NOTE

The term "normal equations" comes from the fact that the residual $\boldsymbol{r} = \boldsymbol{b} - \mathbf{A}\hat{\boldsymbol{x}}$ is normal (perpendicular) to the column space of $\mathbf{A}$. This orthogonality principle is fundamental to least squares theory and follows directly from the normal equations:

$$\mathbf{A}^T(\boldsymbol{b} - \mathbf{A}\hat{\boldsymbol{x}}) = \mathbf{0}$$

This equation confirms that each column of $\mathbf{A}$ is orthogonal to the residual vector.

Examples

Consider fitting a straight line $y = mx + b$ to the data points $(1, 2)$, $(2, 3)$, $(3, 5)$, and $(4, 4)$.
We can rewrite the line equation in matrix form:

$$\begin{pmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ 1 & 4 \end{pmatrix} \begin{pmatrix} b \\ m \end{pmatrix} = \begin{pmatrix} 2 \\ 3 \\ 5 \\ 4 \end{pmatrix}$$

This is an overdetermined system with $\mathbf{A} \in \mathbb{R}^{4\times 2}$ and $\boldsymbol{b} \in \mathbb{R}^4$. Computing the normal equations:

$$\mathbf{A}^T\mathbf{A} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ 1 & 4 \end{pmatrix} = \begin{pmatrix} 4 & 10 \\ 10 & 30 \end{pmatrix}$$

$$\mathbf{A}^T\boldsymbol{b} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 \end{pmatrix} \begin{pmatrix} 2 \\ 3 \\ 5 \\ 4 \end{pmatrix} = \begin{pmatrix} 14 \\ 41 \end{pmatrix}$$

Solving $(\mathbf{A}^T\mathbf{A})\hat{\boldsymbol{x}} = \mathbf{A}^T\boldsymbol{b}$:

$$\begin{pmatrix} 4 & 10 \\ 10 & 30 \end{pmatrix} \begin{pmatrix} b \\ m \end{pmatrix} = \begin{pmatrix} 14 \\ 41 \end{pmatrix}$$

We get $\hat{b} = 1.3$ and $\hat{m} = 0.9$, giving the best-fit line $y = 0.9x + 1.3$.

5.5.3 QR Based Least Squares

QR Based Least Squares

While the normal equations provide a direct formula, they can lead to numerical instability, especially when $\mathbf{A}$ is ill-conditioned. This is because forming $\mathbf{A}^T\mathbf{A}$ effectively squares the condition number:

$$\kappa(\mathbf{A}^T\mathbf{A}) = \kappa(\mathbf{A})^2$$

To address this, we can use the QR decomposition. When $\mathbf{A} = \mathbf{QR}$ where $\mathbf{Q}$ has orthonormal columns and $\mathbf{R}$ is upper triangular, the least squares problem becomes:

$$\min_{\boldsymbol{x}} \|\mathbf{QR}\boldsymbol{x} - \boldsymbol{b}\|_2 = \min_{\boldsymbol{x}} \|\mathbf{R}\boldsymbol{x} - \mathbf{Q}^T\boldsymbol{b}\|_2$$

The solution is found by solving the triangular system:

$$\mathbf{R}\boldsymbol{x} = \mathbf{Q}^T\boldsymbol{b}$$

The QR and SVD approaches avoid forming $\mathbf{A}^T\mathbf{A}$ explicitly, thus preserving numerical precision. For large, sparse systems, iterative methods like Conjugate

Gradient applied to the normal equations (CGNE) can be more efficient than direct factorization methods.

Examples Continuing with our line-fitting example, we can use the QR decomposition of $\mathbf{A}$ to solve the least squares problem.
If $\mathbf{A} = \mathbf{QR}$ where:

$$\mathbf{Q} \approx \begin{pmatrix} 0.5 & -0.67 & ... \\ 0.5 & -0.22 & ... \\ 0.5 & 0.22 & ... \\ 0.5 & 0.67 & ... \end{pmatrix} \quad \text{and} \quad \mathbf{R} \approx \begin{pmatrix} 2 & 5 \\ 0 & 3.74 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}$$

Then the solution can be found by solving:

$$\begin{pmatrix} 2 & 5 \\ 0 & 3.74 \end{pmatrix} \begin{pmatrix} b \\ m \end{pmatrix} = \begin{pmatrix} 7 \\ 3.37 \end{pmatrix}$$

where the right-hand side is $\mathbf{Q}^T\boldsymbol{b}$. This triangular system is easily solved by back-substitution, yielding the same solution as before but with better numerical stability.

NOTE The least squares method can be extended in many ways (covered in introductory machine learning courses), including:

- Weighted least squares, where some measurements are given more importance than others
- Regularized least squares (e.g., ridge regression, LASSO), which address ill-conditioning and prevent overfitting
- Nonlinear least squares for fitting more complex models

These extensions maintain the core principle of minimizing the sum of squared residuals while incorporating additional constraints or structures.

6 Applications of Numerical Linear Algebra

6.1 Principal Component Analysis via SVD

Principal Component Analysis (PCA) is a fundamental technique for dimensionality reduction. It aims to transform a dataset with many variables into a smaller set of variables, called principal components, while retaining most of the original information or variance. The core idea is to find orthogonal directions in the feature space along which the data exhibits the maximum variance. PCA is widely used for data compression, noise reduction, feature extraction, and visualization. A crucial first step for PCA is data centering.

Data Centering

For PCA, the data matrix $\mathbf{X} \in \mathbb{R}^{n \times p}$ (where n is the number of samples and p is the number of features) must be centered. This means that the mean of each column (feature) is subtracted from all entries in that column, resulting in each feature having a zero mean. Throughout this section, $\mathbf{X}$ will denote a centered data matrix.

Covariance Matrix

For a centered data matrix $\mathbf{X} \in \mathbb{R}^{n \times p}$, the sample covariance matrix $\mathbf{S}$ is defined as:

$$\mathbf{S} = \frac{1}{n-1}\mathbf{X}^T\mathbf{X}$$

where $\mathbf{S} \in \mathbb{R}^{p \times p}$. The matrix $\mathbf{S}$ is symmetric and positive semi-definite. Its diagonal entries S_{jj} represent the variance of the j-th feature, and its off-diagonal entries S_{ij} represent the covariance between the i-th and j-th features.

Principal Component Analysis (PCA)

Given a centered data matrix $\mathbf{X} \in \mathbb{R}^{n \times p}$, Principal Component Analysis (PCA) seeks to find a new set of orthogonal basis vectors (principal directions) in the $\mathbb{R}^p$ feature space such that the data projected onto these directions has maximal variance. The principal directions are the eigenvectors of the sample covariance matrix $\mathbf{S}$, ordered according to their corresponding eigenvalues in decreasing order. The eigenvalues represent the variance of the data along these principal directions.

Optimization Problem for Principal Directions The first principal direction $\boldsymbol{w}_1$ is found by solving the following optimization problem:

$$\boldsymbol{w}_1 = \arg\max_{\|\boldsymbol{w}\|=1} \mathrm{Var}(\mathbf{X}\boldsymbol{w}) = \arg\max_{\|\boldsymbol{w}\|=1} \boldsymbol{w}^T\mathbf{S}\boldsymbol{w}$$

The solution $\boldsymbol{w}_1$ is the eigenvector corresponding to the largest eigenvalue of $\mathbf{S}$. Subsequent principal directions $\boldsymbol{w}_k$ are found by solving:

$$\boldsymbol{w}_k = \arg\max_{\|\boldsymbol{w}\|=1, \boldsymbol{w} \perp \{\boldsymbol{w}_1, \ldots, \boldsymbol{w}_{k-1}\}} \boldsymbol{w}^T\mathbf{S}\boldsymbol{w}$$

This iterative process yields the full eigendecomposition of the covariance matrix $\mathbf{S}$:

$$\mathbf{S} = \mathbf{V}\mathbf{V}^T$$

where $\mathbf{V} = [\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_p]$ is an orthogonal matrix whose columns $\boldsymbol{v}_i$ are the eigenvectors of $\mathbf{S}$ (the principal directions), and $= \mathrm{diag}(\lambda_1, \lambda_2, \ldots, \lambda_p)$ is a diagonal matrix whose entries are the eigenvalues of $\mathbf{S}$ (the variances along each principal direction), ordered such that $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_p \geq 0$.

Principal Component Scores

Once the principal directions $\mathbf{V}$ are determined, the original centered data $\mathbf{X}$ can be projected onto these directions to obtain the principal component scores, denoted by $\mathbf{Z}$:

$$\mathbf{Z} = \mathbf{X}\mathbf{V}$$

The matrix $\mathbf{Z} \in \mathbb{R}^{n \times p}$ contains the new coordinates of the n data points in the basis defined by the principal directions. The k-th column of $\mathbf{Z}$, denoted $\boldsymbol{z}_k = \mathbf{X}\boldsymbol{v}_k$, is called the k-th principal component (PC). The sample variance of the k-th principal component is λ_k. The principal components are uncorrelated: $\text{Cov}(\boldsymbol{z}_j, \boldsymbol{z}_k) = 0$ for $j \neq k$.

The SVD-PCA Connection

For a centered data matrix $\mathbf{X} \in \mathbb{R}^{n \times p}$, let its Singular Value Decomposition (SVD) be $\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$. The connection to PCA is as follows:

- **Principal Directions:** The columns of $\mathbf{V}$ (the right singular vectors of $\mathbf{X}$) are precisely the principal directions $\boldsymbol{v}_i$ (eigenvectors of $\mathbf{S}$).
- **Variances:** The squared singular values of $\mathbf{X}$, denoted σ_i^2, are related to the eigenvalues λ_i of $\mathbf{S}$ by $\lambda_i = \frac{\sigma_i^2}{n-1}$. Thus, $\sigma_i^2/(n-1)$ is the variance explained by the i-th principal component.
- **Principal Component Scores:** The principal component scores $\mathbf{Z} = \mathbf{X}\mathbf{V}$ can be directly computed using the SVD as $\mathbf{Z} = \mathbf{U}\mathbf{\Sigma}$. (Assuming $\mathbf{U}$ is $n \times p$ and $\mathbf{\Sigma}$ is $p \times p$ for $n \geq p$, or more generally, $\mathbf{U}$ is $n \times r$, $\mathbf{\Sigma}$ is $r \times r$, $\mathbf{V}$ is $p \times r$ where $r = \text{rank}(\mathbf{X})$).

This connection highlights that PCA can be performed directly using the SVD of $\mathbf{X}$, often providing better numerical stability than forming and diagonalizing $\mathbf{S}$.

Derivation of the SVD-PCA Connection To establish this connection, substitute the SVD of $\mathbf{X}$ into the formula for the covariance matrix $\mathbf{S}$:

$$\begin{aligned}
\mathbf{S} &= \frac{1}{n-1}\mathbf{X}^T\mathbf{X} \\
&= \frac{1}{n-1}(\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T)^T(\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T) \\
&= \frac{1}{n-1}(\mathbf{V}\mathbf{\Sigma}^T\mathbf{U}^T)(\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T) \\
&= \frac{1}{n-1}\mathbf{V}\mathbf{\Sigma}^T(\mathbf{U}^T\mathbf{U})\mathbf{\Sigma}\mathbf{V}^T
\end{aligned} \tag{51}$$

Assuming the "economy" SVD where $\mathbf{U}$ has orthonormal columns (i.e., $\mathbf{U}^T\mathbf{U} = \mathbf{I}_r$, where $r = \text{rank}(\mathbf{X})$ is the number of non-zero singular values, typically $r = p$ if $n \geq p$ and $\mathbf{X}$ has full column rank), and $\mathbf{\Sigma}$ is an $r \times r$ diagonal matrix of singular values σ_i, then $\mathbf{\Sigma}^T\mathbf{\Sigma} = \mathbf{\Sigma}^2$. The equation becomes:

$$\mathbf{S} = \frac{1}{n-1}\mathbf{V}\mathbf{\Sigma}^2\mathbf{V}^T$$

Comparing this with the eigendecomposition of $\mathbf{S} = \mathbf{V}\mathbf{V}^T$, we see that the columns of $\mathbf{V}$ are the eigenvectors of $\mathbf{S}$, and the eigenvalues λ_i are given by $\lambda_i = \frac{\sigma_i^2}{n-1}$.

Dimensionality Reduction via PCA

PCA is often used to reduce the dimensionality from p features to $k < p$ principal components. The first k principal components $\mathbf{Z}_k = \mathbf{X}\mathbf{V}_k = \mathbf{U}_k\mathbf{\Sigma}_k$ (where $\mathbf{V}_k$ contains the first k columns of $\mathbf{V}$, $\mathbf{U}_k$ the first k columns of $\mathbf{U}$, and $\mathbf{\Sigma}_k$ the top-left $k \times k$ block of $\mathbf{\Sigma}$) capture the most variance. The rank-k approximation of the original data $\mathbf{X}$ using the first k principal components is:

$$\mathbf{X}_k = \mathbf{Z}_k\mathbf{V}_k^T = \mathbf{X}\mathbf{V}_k\mathbf{V}_k^T = \mathbf{U}_k\mathbf{\Sigma}_k\mathbf{V}_k^T$$

This matrix $\mathbf{X}_k$ represents the projection of the original data onto the subspace spanned by the first k principal directions. By the Eckart-Young-Mirsky theorem, $\mathbf{X}_k$ is the best rank-k approximation of $\mathbf{X}$ in terms of Frobenius norm, meaning it minimizes $\|\mathbf{X} - \mathbf{A}\|_F^2$ among all rank-k matrices $\mathbf{A}$.

Examples

Consider a centered data matrix $\mathbf{X} \in \mathbb{R}^{100\times 3}$ representing 100 3D points ($n = 100, p = 3$). Suppose the singular values of $\mathbf{X}$ are $\sigma_1 = 10\sqrt{n-1}$, $\sigma_2 = 5\sqrt{n-1}$, $\sigma_3 = 0.5\sqrt{n-1}$. The corresponding eigenvalues of $\mathbf{S}$ would be $\lambda_1 = \sigma_1^2/(n-1) = 100$, $\lambda_2 = 25$, $\lambda_3 = 0.25$.

- The proportion of total variance explained by the first principal component is:
$$\frac{\lambda_1}{\lambda_1 + \lambda_2 + \lambda_3} = \frac{100}{100 + 25 + 0.25} = \frac{100}{125.25} \approx 0.798$$
- The condition number of the covariance matrix $\mathbf{S}$ (assuming $\lambda_3 > 0$) is:
$$\kappa(\mathbf{S}) = \frac{\lambda_{\max}}{\lambda_{\min}} = \frac{\lambda_1}{\lambda_3} = \frac{100}{0.25} = 400$$

A rank-2 approximation $\mathbf{X}_2$ would use the first two principal components. The proportion of variance captured by these two components is:

$$\frac{\lambda_1 + \lambda_2}{\lambda_1 + \lambda_2 + \lambda_3} = \frac{100 + 25}{125.25} = \frac{125}{125.25} \approx 0.998$$

This indicates that the data is nearly coplanar, as approximately 99.8% of the variance is contained within a 2D subspace.

Error Analysis and Choosing k

The reconstruction error from truncating to k principal components, i.e., using $\mathbf{X}_k$ to approximate $\mathbf{X}$, is given by the sum of the neglected singular values squared:

$$\|\mathbf{X} - \mathbf{X}_k\|_F^2 = \sum_{i=k+1}^{r} \sigma_i^2$$

where $r = \text{rank}(\mathbf{X})$. In terms of eigenvalues of $\mathbf{S}$, this is $(n-1)\sum_{i=k+1}^{r} \lambda_i$.
The choice of k (the number of principal components to retain) depends on the application. Common methods include:

- **Proportion of Variance Explained (PVE):** Choose k such that the cumulative PVE reaches a desired threshold (e.g., 90%, 95%). The PVE for k components is:

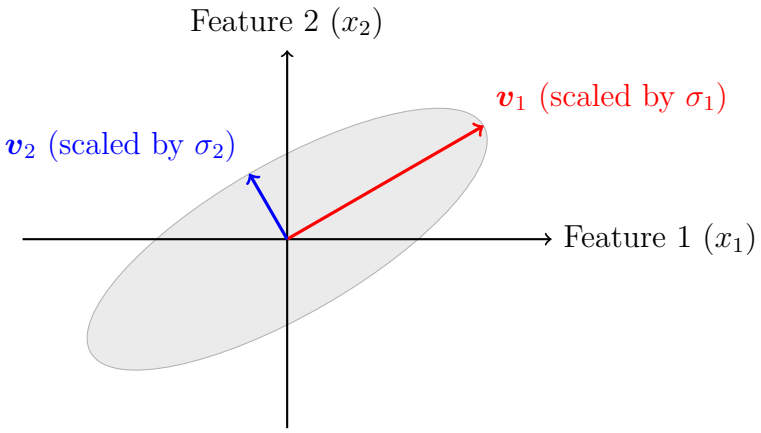
$$\text{PVE}_k = \frac{\sum_{i=1}^{k} \lambda_i}{\sum_{i=1}^{p} \lambda_i} = \frac{\sum_{i=1}^{k} \sigma_i^2}{\sum_{i=1}^{p} \sigma_i^2}$$

  (Assuming $\sum \sigma_i^2 \neq 0$).

- **Scree Plot:** Plot the eigenvalues λ_i (or singular values σ_i) in decreasing order. Look for an "elbow" or a point where the values level off. Components beyond this point might represent noise.

- **Application-specific criteria:** The choice of k might also be guided by the needs of a downstream task, such as visualization ($k = 2$ or $k = 3$) or computational constraints.

NOTE The SVD $\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$ offers a powerful geometric interpretation of PCA. The columns of $\mathbf{V}$ (principal directions) form an orthonormal basis for the feature space $\mathbb{R}^p$. The transformation $\mathbf{X} \to \mathbf{XV} = \mathbf{U\Sigma}$ projects the data points onto this new basis, yielding the principal component scores. The singular values σ_i (related to $\sqrt{\lambda_i}$) indicate the "stretch" or importance of each principal direction in terms of data variance. $\mathbf{U}$ contains an orthonormal basis for the column space of $\mathbf{X}$, representing the principal component scores in a rotated coordinate system in $\mathbb{R}^n$.
For a visualization of principal directions, the figure below (conceptual) illustrates how the right singular vectors from SVD (columns of $\mathbf{V}$) correspond to the principal directions of a 2D dataset. These vectors are typically scaled by their corresponding singular values (or square root of eigenvalues) to indicate the magnitude of variance along each direction.



Principal Directions from SVD

PCA effectively rotates the original coordinate system to a new one aligned with the directions of maximum variance.

Relationship to Eigendecomposition

The SVD of $\mathbf{X}$ is intrinsically linked to the eigendecompositions of $\mathbf{X}^T\mathbf{X}$ and $\mathbf{X}\mathbf{X}^T$:

$$\begin{aligned}(\mathbf{X}^T\mathbf{X})\mathbf{V} &= \mathbf{V}\boldsymbol{\Sigma}^T\boldsymbol{\Sigma} = \mathbf{V}\boldsymbol{\Sigma}_p^2 \\ (\mathbf{X}\mathbf{X}^T)\mathbf{U} &= \mathbf{U}\boldsymbol{\Sigma}\boldsymbol{\Sigma}^T = \mathbf{U}\boldsymbol{\Sigma}_n^2\end{aligned} \tag{52}$$

where $\boldsymbol{\Sigma}_p^2$ is a $p \times p$ diagonal matrix with diagonal entries σ_i^2 (and zeros if $p > r$), and $\boldsymbol{\Sigma}_n^2$ is an $n \times n$ diagonal matrix with diagonal entries σ_i^2 (and zeros if $n > r$). This shows that the right singular vectors $\boldsymbol{v}_i$ (columns of $\mathbf{V}$) are eigenvectors of $\mathbf{X}^T\mathbf{X}$, and the left singular vectors $\boldsymbol{u}_i$ (columns of $\mathbf{U}$) are eigenvectors of $\mathbf{X}\mathbf{X}^T$. The eigenvalues of both $\mathbf{X}^T\mathbf{X}$ and $\mathbf{X}\mathbf{X}^T$ are σ_i^2.

Computational Aspect for High Dimensions ($p \gg n$): When the number of features p is much larger than the number of samples n, computing $\mathbf{S}$ (a $p \times p$ matrix) and its eigenvectors can be prohibitively expensive. Instead, one can compute the eigenvectors of the smaller $n \times n$ matrix $_G = \mathbf{X}\mathbf{X}^T$ (sometimes called the Gram matrix). If $\boldsymbol{u}_i$ is an eigenvector of $\mathbf{X}\mathbf{X}^T$ with eigenvalue σ_i^2:

$$\mathbf{X}\mathbf{X}^T\boldsymbol{u}_i = \sigma_i^2\boldsymbol{u}_i$$

Pre-multiplying by $\mathbf{X}^T$:

$$\mathbf{X}^T(\mathbf{X}\mathbf{X}^T\boldsymbol{u}_i) = \mathbf{X}^T(\sigma_i^2\boldsymbol{u}_i) \implies (\mathbf{X}^T\mathbf{X})(\mathbf{X}^T\boldsymbol{u}_i) = \sigma_i^2(\mathbf{X}^T\boldsymbol{u}_i)$$

This shows that if $\boldsymbol{u}_i$ is an eigenvector of $\mathbf{X}\mathbf{X}^T$, then $\mathbf{X}^T\boldsymbol{u}_i$ is an eigenvector of $\mathbf{X}^T\mathbf{X}$ (i.e., a principal direction). Thus, for $\sigma_i \neq 0$, we can obtain the principal directions $\boldsymbol{v}_i$ by:

$$\boldsymbol{v}_i = \frac{1}{\sigma_i}\mathbf{X}^T\boldsymbol{u}_i$$

The principal component scores can then be found as $\mathbf{Z} = \mathbf{U}\boldsymbol{\Sigma}$. The columns of $\mathbf{U}$ are the eigenvectors of $\mathbf{X}\mathbf{X}^T$, and $\boldsymbol{\Sigma}$ contains the square roots of its eigenvalues.

Statistical Properties of PCA

- **Loadings:** The entries of the principal directions $\boldsymbol{v}_j$ (eigenvectors of $\mathbf{S}$) are called loadings. The loading v_{ji} (the j-th element of $\boldsymbol{v}_i$) quantifies the contribution of the j-th original feature to the i-th principal component. A large absolute value of v_{ji} indicates that the j-th feature is important for the i-th principal component.
- **Correlation with Principal Components:** If the original data $\mathbf{X}$ is standardized (each feature has zero mean and unit variance), then the loading v_{ji} is the correlation coefficient between the j-th original feature and the i-th principal component score $\boldsymbol{z}_i$.

- **Proportion of Variance Explained (PVE):** As mentioned earlier, PVE by the first k principal components is $\frac{\sum_{i=1}^{k} \lambda_i}{\sum_{i=1}^{p} \lambda_i} = \frac{\sum_{i=1}^{k} \sigma_i^2}{\sum_{i=1}^{p} \sigma_i^2}$. This is a key metric for assessing how much information is retained after dimensionality reduction.

NOTE PCA is sensitive to the relative scaling of the original features. Features with larger variances will tend to dominate the first few principal components, even if their variance is due to units of measurement rather than inherent importance. Therefore, it is standard practice to standardize the data before applying PCA, i.e., scale each feature to have unit variance (in addition to zero mean). However, if features are measured in the same units and their relative variances are meaningful, one might choose not to scale.

NOTE

- **Linearity:** PCA assumes that the principal components are linear combinations of the original features and that the underlying subspace is linear. It may not capture non-linear structures effectively (Kernel PCA can address this).
- **Variance as Importance:** PCA assumes that directions with high variance are the most important. This may not always be true, especially in supervised learning contexts where directions that best separate classes might have small variance.
- **Orthogonality:** The principal components are orthogonal by construction. While this ensures they are uncorrelated and provides a unique basis, real-world underlying factors may not be orthogonal.
- **Interpretation:** While loadings help, interpreting principal components can sometimes be challenging as they are combinations of all original features.
- **Sensitivity to Outliers:** PCA is sensitive to outliers in the data, as they can disproportionately influence the mean and variance calculations.

The effectiveness of PCA often hinges on whether the data's intrinsic dimensionality is lower than p and whether directions of high variance are indeed the most informative for the problem at hand. The rapid decay of singular values (or eigenvalues) often indicates that PCA will be effective for dimensionality reduction.

6.2 Spectral Graph Theory

Spectral graph theory studies graph properties through the eigenvalues and eigenvectors of their associated matrices. This approach bridges discrete graph

structures with continuous spectral analysis, enabling powerful insights into connectivity, partitioning, diffusion, and community detection.

Graph Representation

A graph $G = (V, E)$ consists of a vertex set V with $n = |V|$ vertices and an edge set E with $m = |E|$ edges. For an undirected graph, each edge is an unordered pair (i, j) with $i, j \in V$. A weighted graph assigns a positive weight $w_{ij} > 0$ to each edge $(i, j) \in E$. The degree d_i of vertex i is defined as $d_i = \sum_{j:(i,j)\in E} w_{ij}$, which reduces to the number of adjacent vertices in the unweighted case.

Matrix Representations

A graph can be represented by several matrices:

- **Adjacency Matrix (A)**: For unweighted graphs, $A_{ij} = 1$ if $(i, j) \in E$ and 0 otherwise. For weighted graphs, $A_{ij} = w_{ij}$ if $(i, j) \in E$ and 0 otherwise.
- **Degree Matrix (D)**: A diagonal matrix with $D_{ii} = d_i$.
- **Incidence Matrix (M)**: For an undirected graph with arbitrary orientation, $M_{ie} = 1$ if vertex i is the head of edge e, $M_{ie} = -1$ if it is the tail, and $M_{ie} = 0$ otherwise.
- **Laplacian Matrix (L)**: $\mathbf{L} = \mathbf{D} - \mathbf{A}$, or equivalently, $\mathbf{L} = \mathbf{M}\mathbf{M}^T$.

Laplacian as a Differential Operator

The graph Laplacian can be interpreted as a discrete analog of the continuous Laplace operator ∇^2. For a function $f : V \to \mathbb{R}$ defined on vertices, the Laplacian applied to f at vertex i is:

$$(\mathbf{L}f)_i = \sum_{j:(i,j)\in E} w_{ij}(f_i - f_j)$$

This measures how f at vertex i differs from its neighbors, similar to how the continuous Laplacian measures the deviation of a function from its average value in an infinitesimal neighborhood.

Quadratic Form

The quadratic form of the Laplacian provides a measure of function smoothness across the graph:

$$f^T \mathbf{L} f = \sum_{i=1}^{n} \sum_{j=1}^{n} L_{ij} f_i f_j \tag{53}$$

$$= \sum_{i=1}^{n} d_i f_i^2 - \sum_{i=1}^{n} \sum_{j=1}^{n} A_{ij} f_i f_j \tag{54}$$

$$= \sum_{i=1}^{n} \sum_{j:(i,j)\in E} w_{ij} f_i^2 - \sum_{i=1}^{n} \sum_{j:(i,j)\in E} w_{ij} f_i f_j \tag{55}$$

$$= \frac{1}{2} \sum_{i=1}^{n} \sum_{j:(i,j)\in E} w_{ij} (f_i^2 + f_j^2 - 2 f_i f_j) \tag{56}$$

$$= \frac{1}{2} \sum_{i=1}^{n} \sum_{j:(i,j)\in E} w_{ij} (f_i - f_j)^2 \tag{57}$$

The last expression is a weighted sum of squared differences across edges, clearly non-negative, proving $\mathbf{L}$ is positive semidefinite.

Matrix Form of Laplacian

If we identify the elements w_{ij} as defining an adjacency matrix $\mathbf{W}$, then the discrete Laplacian in quadratic form for a vector input (x) can be written as

$$\mathbf{x}^T \mathbf{L} \mathbf{x} = \sum_{i \sim j} w_{ij} (\mathbf{x}_i - \mathbf{x}_j)^2 \tag{58}$$

$$= \sum_{i \sim j} w_{ij} \mathbf{x}_i^2 + \sum_{i \sim j} w_{ij} \mathbf{x}_j^2 - \sum_{i \sim j} 2 w_{ij} \mathbf{x}_i \mathbf{x}_j \tag{59}$$

$$= \sum_{i \sim j} \mathbf{d}_j \mathbf{x}_i^2 + \sum_{i \sim j} \mathbf{d}_i \mathbf{x}_j^2 - 2 \sum_{i \sim j} w_{ij} \mathbf{x}_i \mathbf{x}_j \tag{60}$$

$$= \frac{1}{2} (\mathbf{x}^T \mathbf{D} \mathbf{x} + \mathbf{x}^T \mathbf{D} \mathbf{x} - 2 \mathbf{x}^T \mathbf{W} \mathbf{x}) \tag{61}$$

$$= \mathbf{x}^T (\mathbf{D} - \mathbf{W}) \mathbf{x}. \tag{62}$$

Where $\mathbf{D}$ is the degree diagonal matrix and $\mathbf{W}$ is a matrix encoding w_{ij} values. The expression is divided by half in the third step as $\sum_{i \sim j} = \frac{1}{2} \sum_i \sum_j$, so we do not count repeated maps in the permutation sum ($\therefore \mathbf{D}_{ij} = \mathbf{D}_{ji}$). The previous derivation implies the following well-known relationship for the *Laplacian matrix*

Spectral Properties of the Laplacian

The spectrum of the Laplacian matrix $\mathbf{L}$ consists of eigenvalues $0 = \lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n$ with corresponding eigenvectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$.
Key spectral properties include:

- **Zero Eigenvalue**: $\lambda_1 = 0$ always, with eigenvector $\mathbf{v}_1 = \mathbf{1}/\sqrt{n}$ (the normalized constant vector).

- **Component Count**: The multiplicity of $\lambda_1 = 0$ equals the number of connected components in G.
- **Algebraic Connectivity**: $\lambda_2 > 0$ if and only if G is connected. This value, known as the Fiedler value, quantifies how well-connected the graph is.
- **Diameter Bound**: For a connected graph, $\text{diam}(G) \geq \frac{2}{\lambda_2}$, where $\text{diam}(G)$ is the maximum shortest path length between any two vertices.
- **Cheeger Inequality**: $\frac{\lambda_2}{2} \leq h(G) \leq \sqrt{2\lambda_2}$, where $h(G)$ is the Cheeger constant (isoperimetric number) measuring the graph's "bottleneckedness."

Examples

Complete and Path Graphs Consider two extremes: the complete graph K_n and the path graph P_n.
For K_n, every vertex connects to all others, so $d_i = n-1$ for all i. The Laplacian has eigenvalues:

- $\lambda_1 = 0$ (multiplicity 1)
- $\lambda_2 = \lambda_3 = \ldots = \lambda_n = n$ (multiplicity $n-1$)

For P_3 (path with 3 vertices), the Laplacian is:

$$\mathbf{L} = \begin{pmatrix} 1 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{pmatrix}$$

Its eigenvalues are $\lambda_1 = 0$, $\lambda_2 = 2 - \sqrt{2} \approx 0.59$, $\lambda_3 = 2 + \sqrt{2} \approx 3.41$, with corresponding eigenvectors:

$$\mathbf{v}_1 = \frac{1}{\sqrt{3}}(1, 1, 1)^T \tag{63}$$

$$\mathbf{v}_2 = \frac{1}{\sqrt{2 + (2 - \sqrt{2})^2}}(1, \sqrt{2} - 1, -1)^T \tag{64}$$

$$\mathbf{v}_3 = \frac{1}{\sqrt{2 + (2 + \sqrt{2})^2}}(1, -(\sqrt{2} + 1), 1)^T \tag{65}$$

The low λ_2 for P_3 compared to K_3 reflects its weaker connectivity.

Variational Characterization

The Courant-Fischer theorem provides a variational characterization of Laplacian eigenvalues:

$$\lambda_k = \min_{\substack{S \subset \mathbb{R}^n, \dim(S) = k \\ S \perp \text{span}\{\mathbf{v}_1, \ldots, \mathbf{v}_{k-1}\}}} \max_{\mathbf{x} \in S, \mathbf{x} \neq \mathbf{0}} \frac{\mathbf{x}^T \mathbf{L} \mathbf{x}}{\mathbf{x}^T \mathbf{x}}$$

For the second eigenvalue (Fiedler value):

$$\lambda_2 = \min_{\mathbf{x}\perp\mathbf{1},\mathbf{x}\neq\mathbf{0}} \frac{\mathbf{x}^T\mathbf{L}\mathbf{x}}{\mathbf{x}^T\mathbf{x}} = \min_{\mathbf{x}\perp\mathbf{1},\mathbf{x}\neq\mathbf{0}} \frac{\sum_{(i,j)\in E} w_{ij}(x_i - x_j)^2}{\sum_{i=1}^n x_i^2}$$

This connects the Fiedler value to the optimal solution of a continuous relaxation of the graph partitioning problem.

NOTE The Fiedler vector $\mathbf{v}_2$ carries significant structural information about the graph. Its components can be interpreted as an embedding of vertices on a line that minimizes the sum of squared distances between connected vertices, subject to orthogonality to the constant vector and unit norm constraint. This provides a natural ordering of vertices that tends to place connected vertices close together.

Figure 1: Graph with vertices embedded according to Fiedler vector values. The signs of the components suggest a natural bipartition.

Normalized Laplacians Two important normalized versions of the Laplacian address the bias towards high-degree vertices in the standard Laplacian:

- **Symmetric Normalized Laplacian**: $\mathbf{L}_{sym} = \mathbf{D}^{-1/2}\mathbf{L}\mathbf{D}^{-1/2} = \mathbf{I}–\mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$
- **Random Walk Normalized Laplacian**: $\mathbf{L}_{rw} = \mathbf{D}^{-1}\mathbf{L} = \mathbf{I} - \mathbf{D}^{-1}\mathbf{A}$

These matrices have eigenvalues in $[0, 2]$. While $\mathbf{L}_{sym}$ is symmetric, $\mathbf{L}_{rw}$ is not generally symmetric but relates directly to random walks on the graph. The matrix $\mathbf{P} = \mathbf{D}^{-1}\mathbf{A}$ represents transition probabilities of a random walk, where P_{ij} is the probability of moving from vertex i to vertex j.
The two normalized Laplacians share the same eigenvalues. If $\mathbf{v}$ is an eigenvector of $\mathbf{L}_{sym}$ with eigenvalue λ, then $\mathbf{D}^{-1/2}\mathbf{v}$ is an eigenvector of $\mathbf{L}_{rw}$ with the same eigenvalue.

Spectral Clustering Algorithm Spectral clustering leverages the Laplacian's eigenvectors to partition graphs:

Algorithm 1 Spectral Clustering

Construct the graph Laplacian $\mathbf{L}$ (or normalized version) Compute eigenvectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k$ for the k smallest eigenvalues Form matrix $\mathbf{X} = [\mathbf{v}_2, \ldots, \mathbf{v}_k] \in \mathbb{R}^{n\times(k-1)}$ (skip $\mathbf{v}_1$) Let row i of $\mathbf{X}$ represent vertex i in $\mathbb{R}^{k-1}$ Apply k-means clustering to these rows Assign vertex i to cluster j if row i is assigned to cluster j

This algorithm maps the graph to a low-dimensional space where standard Euclidean clustering methods can be applied. The eigenvectors provide coordinates that capture the graph's community structure.

Graph Cuts and Spectral Partitioning

For a partition of vertices V into disjoint sets A and $B = V \setminus A$, the cut between them is defined as:

$$\text{cut}(A, B) = \sum_{i \in A, j \in B} w_{ij}$$

Several normalized cut objectives have been proposed:

- **RatioCut**: $\text{RatioCut}(A, B) = \frac{\text{cut}(A,B)}{|A|} + \frac{\text{cut}(A,B)}{|B|}$
- **NCut (Normalized Cut)**: $\text{NCut}(A, B) = \frac{\text{cut}(A,B)}{\text{vol}(A)} + \frac{\text{cut}(A,B)}{\text{vol}(B)}$

where $\text{vol}(A) = \sum_{i \in A} d_i$ is the volume of set A.
Minimizing RatioCut is NP-hard, but a relaxation leads to:

$$\min_{\mathbf{f} \perp \mathbf{1}, \|\mathbf{f}\| = \sqrt{n}} \mathbf{f}^T \mathbf{L} \mathbf{f}$$

The solution is the Fiedler vector $\mathbf{v}_2$. Similarly, minimizing NCut relates to the second eigenvector of $\mathbf{L}_{rw}$.

NOTE

The connection between spectral graph theory and random walks provides further insight. The transition matrix $\mathbf{P} = \mathbf{D}^{-1}\mathbf{A}$ describes a Markov process on the graph. Its eigenvalues relate to mixing times and convergence rates. For a connected, non-bipartite graph, a random walk converges to a stationary distribution π with $\pi_i = \frac{d_i}{\sum_j d_j}$.
The convergence rate is governed by the spectral gap $1 - |\eta_2|$, where η_2 is the second largest eigenvalue of $\mathbf{P}$ in absolute value. This relates to the Laplacian through $\eta_i = 1 - \lambda_i$ for the eigenvalues of $\mathbf{L}_{rw}$.
This spectral perspective on random walks has applications in PageRank algorithms, diffusion maps, and semi-supervised learning on graphs.

Heat Kernel and Diffusion

The graph Laplacian governs diffusion processes on graphs through the heat equation:

$$\frac{\partial f}{\partial t} = -\mathbf{L} f$$

The solution is $f(t) = e^{-t\mathbf{L}} f(0)$, where $e^{-t\mathbf{L}}$ is the heat kernel. Using the spectral decomposition $\mathbf{L} = \sum_{i=1}^{n} \lambda_i \mathbf{v}_i \mathbf{v}_i^T$, the heat kernel is:

$$e^{-t\mathbf{L}} = \sum_{i=1}^{n} e^{-t\lambda_i} \mathbf{v}_i \mathbf{v}_i^T$$

This exponentially dampens high-frequency components associated with large eigenvalues while preserving low-frequency components associated with small eigenvalues, functioning as a low-pass filter.

The heat kernel trace $\text{Tr}(e^{-t\mathbf{L}}) = \sum_{i=1}^{n} e^{-t\lambda_i}$ captures the graph's global properties and relates to the concept of "shape" in spectral geometry.

Examples Visualizing the Fiedler Vector Consider the barbell graph consisting of two complete graphs K_5 connected by a single edge.

The Fiedler vector for this graph has approximately constant values within each clique, with opposite signs, reflecting the natural partition. The algebraic connectivity λ_2 is very small, indicating the graph can be easily disconnected.

Matrix Tree Theorem The Matrix Tree Theorem provides a remarkable connection between the Laplacian spectrum and spanning trees. For a connected graph G, the number of spanning trees $\tau(G)$ is:

$$\tau(G) = \frac{1}{n} \prod_{i=2}^{n} \lambda_i$$

where λ_i are the non-zero eigenvalues of the Laplacian. Equivalently, for any i:

$$\tau(G) = \det(\mathbf{L}_{i,i})$$

where $\mathbf{L}_{i,i}$ is the (i,i)-minor of $\mathbf{L}$, obtained by removing the i-th row and column. This theorem connects the purely combinatorial concept of spanning tree enumeration with the analytical properties of the Laplacian spectrum.

6.3 The PageRank Algorithm

PageRank is a celebrated algorithm, famously developed and used by Google, to measure the importance or "rank" of web pages. The core idea is that the importance of a page is determined not just by its content, but also by the number and quality of other pages that link to it. A link from an important page is considered more valuable than a link from a less important one. PageRank models this concept using a random surfer navigating the web graph, with a page's rank being proportional to the likelihood of the surfer landing on that page.

The Web Graph and Random Surfer The World Wide Web can be modeled as a directed graph $G = (V, E)$, where V is the set of web pages (vertices) and a directed edge $(p_j, p_i) \in E$ exists if page p_j contains a hyperlink to page p_i. Let $N = |V|$ be the total number of pages. The PageRank algorithm is often explained via the "random surfer" model:

- A user starts on a random web page.
- At each step, the user randomly clicks on one of the outbound links on the current page to navigate to a new page.

- If the current page has no outbound links (a "dangling node"), the user jumps to another page chosen randomly from all N pages.

The PageRank of a page p_i, denoted $PR(p_i)$, is the long-term probability that the random surfer will be on page p_i. Pages visited more frequently by the random surfer are considered more important.

Basic PageRank Formulation

Let $PR(p_i)$ be the PageRank score of page p_i. Let $B(p_i)$ be the set of pages that link to page p_i. For any page $p_j \in B(p_i)$, let $L(p_j)$ be the number of outbound links from page p_j. The basic idea is that page p_j "distributes" its PageRank score equally among all the pages it links to. Thus, the PageRank of p_i is the sum of the PageRank contributions from all pages linking to it:

$$PR(p_i) = \sum_{p_j \in B(p_i)} \frac{PR(p_j)}{L(p_j)}$$

This is a recursive definition: the rank of a page depends on the rank of other pages.

Matrix Formulation and Power Iteration Let $\mathbf{r}$ be a column vector of PageRank scores, where $r_i = PR(p_i)$, and $\sum_i r_i = 1$. We can define a hyperlink transition matrix $\mathbf{H} \in \mathbb{R}^{N \times N}$ (sometimes denoted $\mathbf{M}^T$ or $\mathbf{P}^T$ if $\mathbf{P}$ is row-stochastic) where:

$$H_{ij} = \begin{cases} 1/L(p_j) & \text{if page } p_j \text{ links to page } p_i \text{ and } L(p_j) > 0 \\ 0 & \text{otherwise} \end{cases}$$

The PageRank equation can then be written as an eigenvector problem: $\mathbf{r} = \mathbf{Hr}$. The PageRank vector $\mathbf{r}$ is the principal eigenvector of $\mathbf{H}$ corresponding to the eigenvalue 1. This can be found using the power iteration method: 1. Initialize $\mathbf{r}^{(0)}$ (e.g., $r_i^{(0)} = 1/N$ for all i). 2. Iterate: $\mathbf{r}^{(k+1)} = \mathbf{Hr}^{(k)}$. 3. Repeat until convergence.
However, this basic formulation has issues:

- **Dangling Nodes:** Pages with no outbound links ($L(p_j) = 0$). Columns in $\mathbf{H}$ corresponding to dangling nodes would be all zero, causing rank to "leak out" of the system during power iteration.
- **Sink Regions (Rank Traps):** If the graph has groups of pages that link to each other but not to outside pages (or if the graph is not strongly connected), the random surfer can get trapped, and power iteration may not converge to a unique, meaningful PageRank distribution.

The Damped PageRank Algorithm (Google Matrix)

To address the issues of dangling nodes and sink regions, and to ensure convergence to a unique positive PageRank vector, the damping factor d (typically $d \approx 0.85$) is introduced. The random surfer model is modified:

- With probability d, the surfer clicks a random link on the current page.
- With probability $1-d$, the surfer jumps to a random page chosen uniformly from all N pages in the web graph.

The PageRank formula becomes:

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in B(p_i)} \frac{PR(p_j)}{L(p_j)}$$

This leads to the Google Matrix . Let $\mathbf{S} \in \mathbb{R}^{N \times N}$ be the column-stochastic hyperlink matrix:

$$S_{ij} = \begin{cases} 1/L(p_j) & \text{if page } p_j \text{ links to page } p_i \text{ and } L(p_j) > 0 \\ 1/N & \text{if page } p_j \text{ is a dangling node (i.e., } L(p_j) = 0) \\ 0 & \text{otherwise (no link and not dangling contribution)} \end{cases}$$

Note: The $1/N$ for dangling nodes ensures $\mathbf{S}$ is column-stochastic (columns sum to 1). The Google Matrix is then defined as:

$$= d\mathbf{S} + \frac{1-d}{N}\mathbf{J}$$

where $\mathbf{J}$ is an $N \times N$ matrix of all ones. The matrix is column-stochastic, irreducible, and primitive. By the Perron-Frobenius theorem, has a unique largest eigenvalue $\lambda_{\max} = 1$, and its corresponding eigenvector $\mathbf{r}$ (the PageRank vector) has all positive entries and is unique up to scaling. We scale $\mathbf{r}$ such that $\|\mathbf{r}\|_1 = \sum_i r_i = 1$. The PageRank vector $\mathbf{r}$ is the solution to $\mathbf{r} = \mathbf{r}$.

Solving for PageRank with the Google Matrix The PageRank vector $\mathbf{r}$ is typically computed using the power iteration method: 1. Initialize $\mathbf{r}^{(0)}$ with $r_i^{(0)} = 1/N$ for all $i = 1, \ldots, N$. Ensure $\|\mathbf{r}^{(0)}\|_1 = 1$. 2. Iterate for $k = 0, 1, 2, \ldots$:

$$\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} = d\mathbf{S}\mathbf{r}^{(k)} + \frac{1-d}{N}\mathbf{J}\mathbf{r}^{(k)}$$

Since $\mathbf{J}\mathbf{r}^{(k)} = \mathbf{1}(\mathbf{1}^T\mathbf{r}^{(k)}) = \mathbf{1}(1) = \mathbf{1}$ (as $\|\mathbf{r}^{(k)}\|_1 = 1$), the iteration simplifies to:

$$\mathbf{r}^{(k+1)} = d\mathbf{S}\mathbf{r}^{(k)} + \frac{1-d}{N}\mathbf{1}$$

3. Continue until convergence, i.e., $\|\mathbf{r}^{(k+1)} - \mathbf{r}^{(k)}\|_1 < \epsilon$ for some small tolerance ϵ.
This iterative process is efficient because the matrix $\mathbf{S}$ (or the original hyperlink graph) is typically very sparse, even though itself is dense. The term $d\mathbf{S}\mathbf{r}^{(k)}$ can be computed by exploiting the sparse structure of web links. The $(1-d)/N \cdot \mathbf{1}$ term represents the teleportation probability uniformly distributed.

Examples Illustrative PageRank Calculation Consider a simple web graph with three pages: P_1, P_2, P_3. Links: $P_1 \to P_2$, $P_2 \to P_1$, $P_2 \to P_3$, $P_3 \to P_1$. Out-degrees: $L(P_1) = 1$, $L(P_2) = 2$, $L(P_3) = 1$. No dangling nodes. Let $d = 0.85$. $N = 3$. $(1-d)/N = 0.15/3 = 0.05$.
The matrix $\mathbf{S}$:

- Column 1 (P_1 links to P_2): $S_{21} = 1/1 = 1$. Others 0.
- Column 2 (P_2 links to P_1, P_3): $S_{12} = 1/2 = 0.5$, $S_{32} = 1/2 = 0.5$. Others 0.
- Column 3 (P_3 links to P_1): $S_{13} = 1/1 = 1$. Others 0.

$$\mathbf{S} = \begin{pmatrix} 0 & 0.5 & 1 \\ 1 & 0 & 0 \\ 0 & 0.5 & 0 \end{pmatrix}$$

Initialize $\mathbf{r}^{(0)} = (1/3, 1/3, 1/3)^T \approx (0.333, 0.333, 0.333)^T$.
Iteration 1: $\mathbf{r}^{(1)} = d\mathbf{S}\mathbf{r}^{(0)} + \frac{1-d}{N}\mathbf{1}\ d\mathbf{S}\mathbf{r}^{(0)} = 0.85 \begin{pmatrix} 0 & 0.5 & 1 \\ 1 & 0 & 0 \\ 0 & 0.5 & 0 \end{pmatrix} \begin{pmatrix} 1/3 \\ 1/3 \\ 1/3 \end{pmatrix} = 0.85 \begin{pmatrix} 0.5 \cdot 1/3 + 1 \cdot 1/3 \\ 1 \cdot 1/3 \\ 0.5 \cdot 1/3 \end{pmatrix}$
$0.85 \begin{pmatrix} 1.5/3 \\ 1/3 \\ 0.5/3 \end{pmatrix} = \begin{pmatrix} 0.425 \\ 0.283 \\ 0.142 \end{pmatrix}$ (approx) $\frac{1-d}{N}\mathbf{1} = \begin{pmatrix} 0.05 \\ 0.05 \\ 0.05 \end{pmatrix}$ $\mathbf{r}^{(1)} = \begin{pmatrix} 0.425 \\ 0.283 \\ 0.142 \end{pmatrix} + \begin{pmatrix} 0.05 \\ 0.05 \\ 0.05 \end{pmatrix} =$
$\begin{pmatrix} 0.475 \\ 0.333 \\ 0.192 \end{pmatrix}$ (Sum = 1.0)
Iteration 2: $\mathbf{r}^{(2)} = d\mathbf{S}\mathbf{r}^{(1)} + \frac{1-d}{N}\mathbf{1}\ d\mathbf{S}\mathbf{r}^{(1)} = 0.85 \begin{pmatrix} 0 & 0.5 & 1 \\ 1 & 0 & 0 \\ 0 & 0.5 & 0 \end{pmatrix} \begin{pmatrix} 0.475 \\ 0.333 \\ 0.192 \end{pmatrix} = 0.85 \begin{pmatrix} 0.5(0.333) + 1(0.19 \\ 1(0.475) \\ 0.5(0.333) \end{pmatrix}$
$0.85 \begin{pmatrix} 0.1665 + 0.192 \\ 0.475 \\ 0.1665 \end{pmatrix} = 0.85 \begin{pmatrix} 0.3585 \\ 0.475 \\ 0.1665 \end{pmatrix} \approx \begin{pmatrix} 0.3047 \\ 0.4038 \\ 0.1415 \end{pmatrix}$ $\mathbf{r}^{(2)} \approx \begin{pmatrix} 0.3047 \\ 0.4038 \\ 0.1415 \end{pmatrix} +$
$\begin{pmatrix} 0.05 \\ 0.05 \\ 0.05 \end{pmatrix} = \begin{pmatrix} 0.3547 \\ 0.4538 \\ 0.1915 \end{pmatrix}$ (Sum $\approx$ 1.0)
Iterations would continue until convergence. P_1 appears to receive the most rank due to two direct incoming sources (one of which is P_2 that splits its rank).

E

- **Probability Distribution:** The PageRank vector $\mathbf{r}$ represents a stationary probability distribution of the random surfer model with teleportation. r_i is the long-term probability of finding the surfer on page p_i.
- **Damping Factor (d):** The choice of d balances between the link structure of the web and the uniform "teleportation" probability. A higher d

gives more weight to the link structure, while a lower d makes ranks more uniform. It also ensures convergence and helps mitigate rank sinks.

- **Scalability:** PageRank is designed to scale to billions of pages due to the sparse nature of the web graph and the efficiency of the power iteration method (when implemented carefully).
- **Personalized/Topic-Specific PageRank:** The teleportation term ($\frac{1-d}{N}\mathbf{1}$) can be modified to jump to a specific subset of pages or pages related to a topic, leading to personalized or topic-sensitive rankings. Instead of jumping to any page with uniform probability $1/N$, the jump can be biased towards a personalized vector $\mathbf{v}$, leading to $\mathbf{r}^{(k+1)} = d\mathbf{S}\mathbf{r}^{(k)} + (1-d)\mathbf{v}$.
- **Robustness:** While PageRank provides a robust measure of importance, it is not entirely immune to manipulation (e.g., link farms), leading to ongoing refinements in search engine algorithms.

PageRank revolutionized web search by providing a query-independent measure of page quality, which, when combined with content relevance, significantly improved search results.