UCSB, Physics 129L, Computational Physics
Lecture notes, Week 4

Zihang Wang (UCSB), zihangwang@ucsb.edu

January 30, 2025

Contents

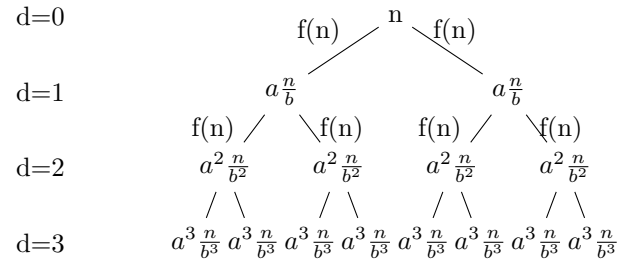
Master Theorem and Divide-and-conquer

The **Master Theorem** is a fundamental tool in the analysis of **divide-and-conquer** algorithms. It determines the **asymptotic** running time of algorithms that solve problems by recursively dividing them into smaller subproblems. The theorem applies to recurrence relations of a Random-access Turing machine, and the the time complexity T can be written in the following recursive form,

$$T(n) = aT\left(\frac{n}{b}\right) + f(n), \tag{1}$$

where,

- n: Size of the input,
- $a \geq 1$: Number of subproblems into which the problem is divided,
- $b > 1$: Factor by which the size of each subproblem is reduced,
- $f(n)$: Cost of dividing the problem and combining the results.



The above tree diagram shows that at each level of recursion,

- The input size decreases by a factor of b, so at level d, the size of each subproblem is n/b^d.
- The number of subproblems increases geometrically, with a^d subproblems at level d.

The total work by recursive calls at

$$\text{Work at level } d \to a^d T\left(\frac{n}{b^d}\right). \tag{2}$$

We can express the total work as a sum over all levels of recursion,

$$T(n) = \sum_{d=0}^{d_{\max}} T\left(\frac{n}{b_d}\right) + f(n), \tag{3}$$

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \tag{4}$$

$$T(n) = a\left[aT\left(\frac{n}{b^2}\right) + f\left(\frac{n}{b}\right)\right] + f(n) \tag{5}$$

$$= a^2 T\left(\frac{n}{b^2}\right) + af\left(\frac{n}{b}\right) + f(n) \tag{6}$$

$$T(n) = a^{d_{\max}} T\left(\frac{n}{b^{d_{\max}}}\right) + \sum_{d=0}^{d_{\max}-1} a^d f\left(\frac{n}{b^d}\right). \tag{7}$$

The recursion continues until the input size reduces to $\mathcal{O}(1)$. This happens after $d = \log_b n$ levels, because at each level, the input size is divided by b. Thus, the height of the recursion tree is,

$$n = b^{d_{\max}} \to d_{\max} = \log_b n. \tag{8}$$

Let's assume that when we reach to the bottom of the tree $d_{\max}$, the work done by the recursion will have a constant time of division, and it is given by,

$$a^{d_{\max}} T(1) = a^{\log_b n} = n^{\log_b a} \to \mathcal{O}(n^{\log_b a}). \tag{9}$$

The term $d_{\max} = \log_b n$ is called the **depth** of the tree, and $c = \log_b a$ is called the **critical exponent**.

$$T(n) = a^{d_{\max}} \mathcal{O}(1) + \sum_{d=0}^{d_{\max}-1} a^d f\left(\frac{n}{b^d}\right) = n^c + \sum_{d=0}^{d_{\max}-1} a^d f\left(\frac{n}{b^d}\right). \tag{10}$$

- **Case 1:** If $f(n) = \mathcal{O}(n^{c-\epsilon})$ and $\epsilon \geq 0$, we have,

$$T(n) = n^c + \sum_{d=0}^{d_{\max}-1} a^d f\left(\frac{n}{b^d}\right) \sim n^c + n^{c-\epsilon} \sum_{d=0}^{d_{\max}-1} \left(\frac{a}{b^{c-\epsilon}}\right)^d = \mathcal{O}(n^c). \tag{11}$$

  The logarithmic divergence (or logarithmic growth) is smaller than any polynomial with a non-negative exponent for sufficiently large values of n.

- **Case 2:** If $f(n) = \mathcal{O}(n^{c-\epsilon} \log^k n)$ with $k \geq 0$, we have,

$$T(n) = n^c + \sum_{d=0}^{d_{\max}-1} a^d f\left(\frac{n}{b^d}\right) \sim n^c + n^{c-\epsilon} (\log n)^k \sum_{d=0}^{d_{\max}-1} \left(\frac{a}{b^{c-\epsilon}}\right)^d = \mathcal{O}(n^{c-\epsilon} \log^{k+1} n). \tag{12}$$

- **Case 3:** If $f(n) = \mathcal{O}(n^{c-\epsilon})$ and $\epsilon < 0$, and the **regularity condition** holds ($af(n/b) \leq \lambda f(n)$ for some constant $0 < \lambda < 1$), then

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \tag{13}$$

$$T(n) = a\left[aT\left(\frac{n}{b^2}\right) + f\left(\frac{n}{b}\right)\right] + f(n) \tag{14}$$

$$= a^2 T\left(\frac{n}{b^2}\right) + af\left(\frac{n}{b}\right) + f(n) \tag{15}$$

$$= a^2 T\left(\frac{n}{b^2}\right) + (1+\lambda) f(n) \tag{16}$$

$$= a^2\left[aT\left(\frac{n}{b^3}\right) + f\left(\frac{n}{b^2}\right)\right] + (1+\lambda) f(n) \tag{17}$$

$$= a^3 T\left(\frac{n}{b^3}\right) + a\lambda[f\left(\frac{n}{b}\right)] + (1+\lambda) f(n) \tag{18}$$

$$= a^3 T\left(\frac{n}{b^3}\right) + (1+\lambda+\lambda^2) f(n) \tag{19}$$

$$T(n) = a^{d_{\max}} T\left(\frac{n}{b^{d_{\max}}}\right) + f(n) \sum_{i=0}^{d_{\max}-1} \lambda^i \sim a^{d_{\max}} T\left(\frac{n}{b^{d_{\max}}}\right) + \frac{1}{1-\lambda} f(n). \tag{20}$$

Since we know $f(n) = \mathcal{O}(n^{c-\epsilon})$, the above expression becomes,

$$T(n) \sim \mathcal{O}(f(n)). \tag{21}$$

Example 2:Binary Search

$$T(n) = T\left(\frac{n}{2}\right) + \mathcal{O}(1) \tag{22}$$

Given:

$$a = 1, \quad b = 2, c = 0, \quad k = 0, \quad \epsilon = 0 \tag{23}$$

Using case 1 (or 2), we have,

$$T(n) \sim \mathcal{O}(1) + \sum_{d=0}^{d_{\max}-1} \mathcal{O}(1) = \mathcal{O}(d_{\max}) = \mathcal{O}(\log n). \tag{24}$$

Example 2: Merge Sort

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Given:

$$a = 2, \quad b = 2, \quad c = 1, \quad k = 0, \quad \epsilon = 0, \tag{25}$$

Using case 2, we have,

$$T(n) \sim n \sum_{d=0}^{d_{\max}-1} \left(\frac{2}{2}\right)^d = \mathcal{O}(nd_{\max}) = \mathcal{O}(n \log n). \tag{26}$$

Example 3: Complex Recurrence

Let's consider the following,

$$T(n) = 3T\left(\frac{n}{2}\right) + n^2 \tag{27}$$

Given:

$$a = 3, \quad b = 2, c = \log_b a = \log_2 3, \quad k = 0, \quad \epsilon = c - 2, \tag{28}$$

Following the result from case 3, we have,

$$T(n) = O(n^2). \tag{29}$$

Example 4

$$T(n) = 3T(n/2) + (\log n)^2, \tag{30}$$

Given,

$$a = 3, \quad b = 2, \quad c = \log_2 3, \quad k = 2, \quad \epsilon = c, \tag{31}$$

$$\begin{aligned} T(n) \sim & n^{\log_2 3} + (\log n)^2 \sum_{d=0}^{d_{\max}-1} (3)^d \\ \sim & n^{\log_2 3} + (\log n)^2 3^{d_{\max}} = n^{\log_2 3} + (\log_2 n)^2 3^{log_2(n)} = n^{\log_2 3} + (\log_2 n)^2 n^{\log_2 3} \\ \sim & (\log_2 n)^2 n^{\log_2 3} \sim n^{\log_2 3}. \end{aligned} \tag{32}$$

Example 5

$$T(n) = 2T(n/2) + n(\log n)^2, \tag{33}$$

Given,

$$a = 2, \quad b = 2, \quad c = 1, \quad k = 2, \quad \epsilon = 0, \tag{34}$$

$$T(n) \sim n(\log n)^2 \sum_{d=0}^{d_{\max}-1} \left(\frac{2}{2}\right)^d = \mathcal{O}(n(\log n)^2 d_{\max}) = \mathcal{O}(n(\log n)^3). \tag{35}$$

Dynamic programming: Top-Down vs Bottom-Up

Dynamic Programming (DP) is an optimization technique used to solve problems with overlapping subproblems and optimal substructure properties. In Divided and Conquer, subproblems are independent of each other while in dynamic programming the solution to one subproblem is reused in other subproblems. It usually involves solving smaller subproblems, storing their results, and reusing these results to construct the solution to the main problem. There are two primary approaches to implementing DP: **Top-Down Approach (Memoization), and Bottom-Up Approach (Tabulation)**.

The Top-Down approach

The Top-Down approach begins with the main problem and recursively breaks it into smaller subproblems. The results of these subproblems are stored in memory to avoid redundant computations. It is **recursive**: The solution is implemented using recursive function calls from its **memoization:** A lookup table is maintained to store the results of subproblems. Before solving a subproblem, the table is checked for precomputed values.

Top-Down approach **sub-divides the main problem at the top of the tree and recursively delays evaluations of subproblems** until reaching the base case. Those delayed function called are stored in the cache (memory), namely the **stack**. When it reaches the base case, each delayed function calls will be evaluated backwards (that is way we use stack instead of queue) until the top of the tree.

For example,

$$F(n) = F(n-1) + F(n-2)$$

Memorization stores intermediate results. The call stack for computing Fibonacci numbers looks like this:

$$\text{Call Stack: } F(n) \to F(n-1) \to F(n-2) \to \dots$$

Once a value is computed, it is stored for reuse, avoiding redundant calculations.

The Bottom-Up approach

The Bottom-Up approach starts by solving the smallest subproblems first and iteratively builds up to the solution of the main problem. This method avoids recursion and uses iteration instead. **Tabulation:** A table or array is used to store the results of subproblems in increasing order of complexity.

Bottom-Up approach **has all subproblems solved systematically and builds from smallest subproblems one by one**, and the results will be stored in a table. It has no delayed function calls, therefore, no need additional memory.

Bottom-Up Approach (Tabulation)

Iteratively calculate from the bottom:

$$F(2) = F(1) + F(0), \quad F(3) = F(2) + F(1), \quad \ldots, \quad F(n)$$

and store all values up to $F(n)$ in an array.

Traversal Problem

Traversal algorithms are essential for simulating dynamics on grid-based systems, such as solving Laplace's equation on a discretized domain. Traversal problems involve systematically visiting all nodes or edges in a data structure, such as a graph or tree, to compute values, extract information, or solve optimization tasks. These problems often require exploring the structure in a specific order, such as depth-first or breadth-first, depending on the nature of the problem. For example, traversal is used to find connected components in a graph, compute shortest paths, or process hierarchical data in trees. The choice of traversal algorithm—Depth-First Search (DFS) or Breadth-First Search (BFS)—depends on the problem's requirements, such as whether depth-first exploration or level-by-level processing is more appropriate.

Depth-First Search (DFS)

DFS explores as deep as possible along one branch of the graph before backtracking. It uses a stack (either explicitly or via recursion) to keep track of the nodes to visit next.

Breadth-First Search (BFS)

BFS explores all neighbors of a node at the current depth level before moving deeper into the graph. It uses a queue to manage the order of node exploration.

Common Data Structures

Arrays

An **array** is a linear data structure where elements are stored in memory locations. An array can be denoted as:

$$A = \{a_0, a_1, \ldots, a_{n-1}\},$$

where each element a_i can be accessed in constant time ($O(1)$) via its index i:

$$\text{Access}(A, i) = a_i.$$

They are **ordered**.

Linked Lists

A **linked list** is a linear structure in which each element (node) stores data and a pointer (or reference) to the next node. Formally, a node can be represented by:

$$\text{Node} = (\text{data}, \text{next}),$$

where data holds the node's value, and next points to the subsequent node in the list. Key operations in a linked list (e.g., insertion and deletion of nodes) typically take $O(1)$ time if the node position is known, but searching an element takes $O(n)$ time in the worst case.

Stacks and Queues

Stacks and **queues** are linear data structures with restricted access patterns:

- A **stack** follows **Last In, First Out** (LIFO).
- A **queue** follows **First In, First Out** (FIFO).

Trees

A **tree** is a hierarchical structure consisting of nodes connected by edges, and a node can have zero or more children. We have discussed few examples previously,

- **Binary Tree**: Each node has at most two children.
- **Binary Search Tree (BST)**: all elements in the left subtree are smaller, and all elements in the right subtree are larger.
- **Heaps**: Satisfy the heap property: each node has a smaller (larger) key than its children key(parent) $\leq$ ($\geq$)key(child) for a **min heap**/**max heap**.

Hash Tables

A **hash table** is a data structure that stores **key-value** pairs for efficient lookups, insertions, and deletions. Unlike the array structure, hash tables are **not ordered**. To access the value of a given key, it relies on a hash function that maps a universe of possible keys U to an index in an underlying array of size m,

$$h : U \to \{0, 1, 2, \ldots, m-1\},$$

In other words, it preforms a dimension reduction on the universe. The load factor α measures how full the hash table is:

$$\alpha = \frac{n}{m},$$

where n is the number of elements stored and m is the table size. Maintaining α below a certain threshold (e.g., $\alpha < 1$) helps sustain average-case $O(1)$ performance. Using hash tables can significantly reduce the runtime of simulations involving large datasets, such as those encountered in astrophysics or high-energy physics.

Collision Handling. When multiple keys map to the same index, collisions occur, where collided keys can be stored in a linked list.

Properly implemented, hash tables offer average $O(1)$ time complexity for inserts, lookups, and deletions.

Cryptographic hash functions, such as SHA-256, are designed to provide security guarantees like, collision resistance, and the avalanche effect. These properties make them suitable for applications like digital signatures, password hashing, and data integrity verification. However, cryptographic hash functions are computationally expensive due to their stringent security requirements. In contrast, **hash functions used in hash tables** prioritize speed and efficiency over security. These non-cryptographic hash functions aim to **uniformly distribute keys across a fixed-size table to minimize collisions and ensure fast lookups**.

6. Graphs

A **graph** is a collection of vertices (or nodes) and edges, expressed as $G = (V, E)$, where V is the set of vertices and E is the set of edges. The triangulation of a point we discussed previously can be understood as an undirected (or directed, depending on the context) graph. Each node has a list of adjacent vertices (you can generalize it to a matrix).

Rectangular Matrices

A **rectangular matrix** is a matrix with dimensions $m \times n$, where $m \neq n$. This means the number of rows is not equal to the number of columns. Rectangular matrices are fundamental in representing systems of linear equations, particularly when the system is either overdetermined ($m > n$) or underdetermined ($m < n$).

The **rank** of a matrix A is the number of linearly independent rows or columns such that,

$$r = \text{rank}(A) \leq \min(m, n). \tag{36}$$

A matrix is said to be **rank deficient** if its rank is less than the minimum of its number of rows and columns.

Representation of Linear Systems

A system of linear equations can be written in matrix form as:

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

where:

- A is an $m \times n$ rectangular matrix representing the coefficients,
- $\mathbf{x}$ is an $n \times 1$ column vector of unknowns,
- $\mathbf{b}$ is an $m \times 1$ column vector of constants.

Solving Overdetermined Systems ($m > n$)

In **overdetermined** systems, there are **more equations than unknowns**, which often makes the system inconsistent. To find an approximate solution, we use the **least squares method**, which minimizes the **convex error functional**,

$$E = \min_{\mathbf{x}} \|A\mathbf{x} - \mathbf{b}\|^2, \rightarrow \frac{\partial E}{\partial \mathbf{x}} = 0, \tag{37}$$

which gives,

$$\nabla_{\mathbf{x}} \|A\mathbf{x} - \mathbf{b}\|^2 = 2A^T(A\mathbf{x} - \mathbf{b}) = 0. \tag{38}$$

In other words, minimizing the error functional is equivalent to solve equation of motion: namely the **normal equations**:

$$A^T A\mathbf{x} = A^T\mathbf{b}. \tag{39}$$

You should realize that in this case, the $\mathbf{x}$ we obtained from the **normal equations are not the solution to** $A\mathbf{x} = \mathbf{b}$, but it is a solution that minimize the error functional.

Solving Underdetermined Systems ($m < n$)

In underdetermined systems, there are more unknowns than equations, leading to infinitely many solutions. The general solution can be expressed as:

$$\mathbf{x} = \mathbf{x}_p + \mathbf{x}_h, \tag{40}$$

where $\mathbf{x}_p$ is a particular solution, and$\mathbf{x}_h$ is a homogeneous solution in the **null space** $\mathbf{x}_h \in \mathcal{N}$ that satisfied the following homogeneous equation,

$$A\mathbf{x}_h = 0. \tag{41}$$

These infinitely many solutions that satisfied $A\mathbf{x} = \mathbf{b}$ reflect the underlying **redundancy** of an underdetermined problem. Those redundancy can be further reduced by enforcing additional **convex functionals** $L_1(\mathbf{x}), L_2(\mathbf{x}) \cdots$. We solve the following **optimization problem**,

$$L_1(\mathbf{x}), L_2(\mathbf{x}) \cdots : \quad \text{subject to the constraint} A\mathbf{x} = \mathbf{b}. \tag{42}$$

The systematic elimination of redundancy in underdetermined physical systems can dramatically simplify problems without altering their underlying physics.

As an example, let's consider a functional that describes the smallest Euclidean norm $L(\mathbf{x}) = \|\mathbf{x}\|_2^2$. To find the smallest Euclidean norm $L(\mathbf{x}) = \|\mathbf{x}\|_2^2$, we are looking for the vector $\mathbf{x}$ such that,

- Satisfies the constraint $A\mathbf{x} = \mathbf{b}$,
- Has the smallest possible magnitude (Euclidean norm) $L(\mathbf{x}) = \|\mathbf{x}\|_2^2$.

The problem can be solved using **Lagrange multipliers**. Define the Lagrangian:

$$\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}) = \|\mathbf{x}\|_2^2 + 2\boldsymbol{\lambda}^T(A\mathbf{x} - \mathbf{b}), \tag{43}$$

where $\boldsymbol{\lambda}$ is a vector of Lagrange multipliers.

Take derivatives with respect to $\mathbf{x}$ and set them to zero,

$$\nabla_{\mathbf{x}}\mathcal{L} = 2\mathbf{x} + 2A^T\boldsymbol{\lambda} = 0. \tag{44}$$

This gives,

$$\mathbf{x} = -A^T\boldsymbol{\lambda}, \tag{45}$$

and substitute this into the constraint $A\mathbf{x} = \mathbf{b}$,

$$A(-A^T\boldsymbol{\lambda}) = \mathbf{b}. \tag{46}$$

$$AA^T\boldsymbol{\lambda} = \mathbf{b}, \rightarrow \boldsymbol{\lambda} = (AA^T)^{-1}\mathbf{b}. \tag{47}$$

Substitute back into $\mathbf{x} = -A^T\boldsymbol{\lambda}$, we have,

$$\mathbf{x}_{ln} = A^T(AA^T)^{-1}\mathbf{b}. \tag{48}$$

This is the least-norm solution. The derivation above shows that solving the normal equations ensures the solution satisfies the constraint $A\mathbf{x}_{ln} = \mathbf{b}$, and among all possible solutions, it minimizes the norm. Let's assume the matrix A has m rows and n columns ($m < n$). The time complexity can be understood as the following,

- Computing AA^T: This requires $O(m^2n)$ operations.
- Inverting AA^T: This involves inverting an $m \times m$ matrix, which requires $O(m^3)$ operations.
- Computing $(AA^T)^{-1}\mathbf{b}$: This requires $O(m^2)$ operations.
- Computing $A^T(AA^T)^{-1}\mathbf{b}$: This involves a matrix-vector multiplication with A^T (with dimension $n \times m$) and an m-dimensional vector, requiring $O(mn)$ operations.

Thus, the total time complexity is:

$$O(m^2n + m^3 + mn).$$

It is important to note that $\boldsymbol{\lambda}$ is an **auxiliary field** that contains no dynamic terms ($\dot{\boldsymbol{\lambda}} = 0$) while have $\partial_{\mathbf{x}}\boldsymbol{\lambda}$ in general. The auxiliary field $\boldsymbol{\lambda}$ sets the constraint and is not physical. This is different from a gauge field, which is physical and can be dynamical.

Time Complexity of matrix decomposition and operations

Gaussian Elimination

Gaussian Elimination is a fundamental algorithm in linear algebra for solving systems of linear equations. It transforms a given system of equations into an equivalent upper triangular form (**row echelon form**) using a series of elementary row operations. Once the system is in upper triangular form, the solution can be easily obtained through back substitution.

Given a system of linear equations represented in matrix form $Ax = b$, where $A \in \mathbb{R}^{n \times n}$ is the coefficient matrix and $b \in \mathbb{R}^n$ is the constant vector, the steps are as follows:

1. **Forward Elimination**: Perform row operations to eliminate the entries below the **pivot (diagonal) elements**, transforming A into an upper triangular matrix U.
2. **Back Substitution**: Solve the upper triangular system $Ux = b'$ (where b' is the modified constant vector) by **starting from the last equation and substituting backward**.

The time complexity of Gaussian Elimination is dominated by the forward elimination step, which involves $O(n^3)$ operations for an $n \times n$ matrix.

- **Forward Elimination**:
  - For each column, perform row operations to eliminate the entries below the pivot $O(n)$.
  - Each row operation involves $O(n)$ arithmetic operations (multiplications, additions, etc.).
  - We have to do this for all n columns. Therefore, the total number of operations is approximately $O(n^3)$.
- **Back Substitution**:
  - Solving the upper triangular system requires $O(n^2)$ operations, as each of the n variables is computed using a linear combination of previously computed variables.

Let's look at the following example,

$$x + y + z = 6, \quad 2x + 3y + 7z = 18, \quad 4x + 9y + 15z = 40.$$

Represent the system as an **augmented matrix**:

$$A = \left[\begin{array}{ccc|c} 1 & 1 & 1 & 6 \\ 2 & 3 & 7 & 18 \\ 4 & 9 & 15 & 40 \end{array}\right].$$

Perform row operations to reduce A to upper triangular form:

$$A = \left[\begin{array}{ccc|c} 1 & 1 & 1 & 6 \\ 0 & 1 & 5 & 6 \\ 0 & 0 & -2 & -2 \end{array}\right].$$

Solve using back substitution:

$$z = 1, \quad y = 1, \quad x = 4.$$

Rectangular matrix multiplication

Let's consider the multiplication between two **matrices** $A \in \mathbb{R}^{n\times k}$ and $B \in \mathbb{R}^{k\times m}$, with following components,

$$AB_{nm} = \sum_{i=1}^{k} A_{ni}B_{im}. \tag{49}$$

The computational cost of matrix multiplication depends on the dimensions of the matrices: For $A \in \mathbb{R}^{n\times k}$ and $B \in \mathbb{R}^{k\times m}$, the total number of steps (scalar multiplications) required is,

$$T \sim \mathcal{O}(nkm). \tag{50}$$

For square matrix multiplication $A, B \in \mathbb{R}^{n\times n}$, the time complexity becomes $T \sim \mathcal{O}(n^3)$. You will look at a particular example called Strassen's Algorithm in the section worksheet.

Gauss–Jacobi Iteration

Gauss–Jacobi is an iterative method for solving large systems of linear equations. It is particularly useful for sparse matrices. The matrix A is split into two parts:

$$A = D + L + U$$

where:

- D is the diagonal matrix consisting of the diagonal elements of A,
- L is the lower triangular part of A (excluding the diagonal),
- U is the upper triangular part of A (excluding the diagonal).

Rearrange the system of equations to express $\mathbf{x}$ in terms of the previous iteration's values:

$$\mathbf{x}^{(k+1)} = D^{-1}\left(\mathbf{b} - (L+U)\mathbf{x}^{(k)}\right)$$

where $\mathbf{x}^{(k)}$ represents the values of the unknowns at the k-th iteration. Start with an initial guess $\mathbf{x}^{(0)}$ (often set to zero). Update the values of $\mathbf{x}^{(k+1)}$ using the above equation, iterating until the solution converges. The iteration continues until the change in the solution becomes sufficiently small, typically when,

$$\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\| < \epsilon$$

where ϵ is a small tolerance value (like 10^{-6}).

QR Decomposition

The QR decomposition factorizes a matrix $A \in \mathbb{R}^{m \times n}$ into the following form,

$$A = QR, \tag{51}$$

where,

- Q: Orthogonal matrix ($Q^T Q = I$).
- R: Upper triangular matrix.

Given matrix $A = (\mathbf{a}_1\, \mathbf{a}_2\, \ldots\, \mathbf{a}_n)$, where $\mathbf{a}_i$ represents the i-th column of A, the **Gram-Schmidt process** constructs orthonormal vectors $\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_n$. Compute the First Column of Q Take the first column of A, denoted $\mathbf{a}_1$:

$$\mathbf{a}_1 = \begin{pmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{pmatrix}$$

Normalize $\mathbf{a}_1$ to get $\mathbf{q}_1$:

$$\|\mathbf{a}_1\| = \sqrt{a_{11}^2 + a_{21}^2 + \cdots + a_{m1}^2}$$

$$\mathbf{q}_1 = \frac{\mathbf{a}_1}{\|\mathbf{a}_1\|}$$

Compute the Second Column of Q Take the second column $\mathbf{a}_2$:

$$\mathbf{a}_2 = \begin{pmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{m2} \end{pmatrix}$$

Now, project $\mathbf{a}_2$ onto $\mathbf{q}_1$:

$$\text{proj}_{\mathbf{q}_1} \mathbf{a}_2 = (\mathbf{a}_2 \cdot \mathbf{q}_1)\, \mathbf{q}_1$$

The dot product is:

$$\mathbf{a}_2 \cdot \mathbf{q}_1 = \sum_{i=1}^{m} a_{i2} q_{i1}$$

Subtract the projection from $\mathbf{a}_2$:

$$\mathbf{a}_2' = \mathbf{a}_2 - \text{proj}_{\mathbf{q}_1} \mathbf{a}_2$$

This is called **Gram-Schmidt process**. We can then normalize $\mathbf{a}_2'$ to get $\mathbf{q}_2$:

$$\|\mathbf{a}_2'\| = \sqrt{a_2'^2}$$

$$\mathbf{q}_2 = \frac{\mathbf{a}_2'}{\|\mathbf{a}_2'\|}$$

Compute the Remaining Columns of Q For each subsequent column $\mathbf{a}_k$ (for $k = 3, 4, \ldots, n$), repeat the following process:

- Project $\mathbf{a}_k$ onto all previous vectors $\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_{k-1}$:

$$\text{proj}_{\mathbf{q}_i} \mathbf{a}_k = (\mathbf{a}_k \cdot \mathbf{q}_i)\, \mathbf{q}_i \quad \text{for } i = 1, 2, \ldots, k-1$$

- Subtract the projections from $\mathbf{a}_k$ to obtain $\mathbf{a}_k'$:

$$\mathbf{a}_k' = \mathbf{a}_k - \sum_{i=1}^{k-1} (\mathbf{a}_k \cdot \mathbf{q}_i)\, \mathbf{q}_i$$

- Normalize $\mathbf{a}_k'$ to get $\mathbf{q}_k$:

$$\mathbf{q}_k = \frac{\mathbf{a}_k'}{\|\mathbf{a}_k'\|}$$

Construct the Matrix Q After applying the Gram-Schmidt process to all columns of A, construct the matrix Q by placing the orthonormal vectors $\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_n$ as the columns:

$$Q = \begin{pmatrix} \mathbf{q}_1 & \mathbf{q}_2 & \cdots & \mathbf{q}_n \end{pmatrix}$$

The matrix R is an upper triangular matrix, and it can be computed as:

$$R = Q^T A$$

Thus, we have the QR decomposition $A = QR$, where Q is an orthogonal matrix, and R is an upper triangular matrix.

The QR algorithm is an iterative method for diagonalizing a matrix, which works by repeatedly performing QR decompositions:

1. Start with the matrix $H_0 = H$.
2. Perform a QR decomposition of H_0, i.e., write
$$H_0 = Q_0 R_0,$$
where Q_0 is orthogonal and R_0 is upper triangular.
3. Construct the new matrix
$$H_1 = R_0 Q_0.$$
4. Repeat the process until converging to a diagonal matrix,
$$H_{n+1} = R_n Q_n.$$

LU Decomposition

The **LU decomposition** factorizes a square matrix $A \in \mathbb{R}^{n \times n}$ into a lower triangular matrix L and an upper triangular matrix U,

$$A = LU, \tag{52}$$

where,

- L: Lower triangular matrix with ones on the diagonal.
- U: Upper triangular matrix.

It solves linear systems $Ax = b$ efficiently since the LU decomposition $A = LU$ is computed once. If the same matrix A is used to solve multiple systems with different right-hand sides b, the LU factorization can be reused for each new b, reducing the computational cost. Additionally, solving two triangular systems (forward and backward substitution) is computationally less expensive than performing Gaussian elimination multiple times. The steps can be expressed as the following,

1. Decompose A into LU.
2. Solve $Ly = b$ using forward substitution.
3. Solve $Ux = y$ using backward substitution.

It can be used to compute the **inverse of a matrix** A (A^{-1}),

$$AX = I,$$

where:

- A is the matrix we want to invert.
- I is the identity matrix.
- X is the resulting inverse matrix, A^{-1}, such that $AX = I$.

Using LU decomposition:

- First, decompose A into:
$$A = LU,$$
where L is a lower triangular matrix, and U is an upper triangular matrix.
- Substitute $A = LU$ into $AX = I$:
$$(LU)X = I.$$
- Break this into two simpler systems:
$$LY = I,$$
$$UX = Y.$$

- Solve:
  - $LY = I$ using **forward substitution** (since L is lower triangular).
  - $UX = Y$ using **backward substitution** (since U is upper triangular).

Since the identity matrix I has known values (1 on the diagonal and 0 elsewhere), solving for each column of X is straightforward. The inverse A^{-1} can be found by solving $AX = I$, which is equivalent to solving n linear systems:

$$Ax_1 = e_1,\ Ax_2 = e_2,\ \ldots,\ Ax_n = e_n,$$

where:

- $x_1, x_2, \ldots, x_n$ are the columns of A^{-1},
- $e_1, e_2, \ldots, e_n$ are the columns of the identity matrix I.

Each system $Ax_i = e_i$ is solved using the LU decomposition of A. You process one column of I (e.g., e_1) at a time, solve for the corresponding column of X, and repeat.

Let's consider the following example of a 3×3 matrix A:

1. Decompose $A = LU$.
2. Solve $AX = I$ by solving:

$$Ax_1 = e_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad Ax_2 = e_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad Ax_3 = e_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}.$$

3. Assemble the results as:

$$A^{-1} = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix}.$$

You can see process avoids directly computing A^{-1} via costly determinant-based methods.

Cholesky Decomposition

For a **symmetric positive definite matrix** $A \in \mathbb{R}^{n \times n}$, the Cholesky decomposition factorizes A into:

$$A = LL^T,$$

- L: Lower triangular matrix.

Sparse Matrix

Consider the sparse matrix: Instead of storing all elements, we store only non-zero entries and their positions:

Values: $[a, b, c, d, d, e]$, Row Indices: $[1, 2, 3, 3, 4, 4]$, Column Indices: $[1, 2, 3, 4, 3, 4]$.

$$A = \begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & d \\ 0 & 0 & d & e \end{bmatrix}.$$

Singular Value Decomposition (SVD)

SVD is one of the most important decomposition algorithms. It allows us to solve ill-conditioned linear systems and preform data compressions.

Given a matrix $A \in \mathbb{R}^{m \times n}$, the SVD factorization is expressed as,

$$A = U\Sigma V^T, \tag{53}$$

where we define the rank of A as $r \leq \min(m, n)$, and,

- $U \in \mathbb{R}^{m \times m}$ is an orthogonal (or unitary, in the complex case) matrix, representing the left singular vectors of A,
- $\Sigma \in \mathbb{R}^{m \times n}$ is a diagonal matrix with non-negative real numbers on the diagonal, known as the **singular values** of A,
- $V \in \mathbb{R}^{n \times n}$ is an orthogonal (or unitary) matrix, representing the right singular vectors of A.

The columns of U and V are orthonormal bases, and the matrix A maps the basis vector $\mathbf{v}_i \in V$ to the basis vector $\mathbf{u}_i \in U$, scaled by the corresponding singular value σ_i,

$$AV = U\Sigma(V^T V) \rightarrow A\mathbf{v}_i = \sigma_i \mathbf{u}_i. \tag{54}$$

By definition of a unitary matrix, the same holds for the transpose. The first r columns of U form a basis for the column space of A. The last $m - r$ columns of U form a basis for the null space of A^T. Similarly, the first r columns of V form a basis for the column space of A^T (the row space of A). The last $n - r$ columns of V form a basis for the null space of A.

One of the SVD applications is the **principal component analysis**. PCA is a dimensionality reduction technique that identifies the directions (principal components) of maximum variance in high-dimensional data. It projects the data onto these components to reduce dimensionality.

$$\mu_j = \frac{1}{n} \sum_{i=1}^{n} X_{ij}, \quad \text{for } j = 1, 2, \ldots, m.$$

and

$$X_{\mathrm{c},ij} = X_{ij} - \mu_j, \quad \text{for } i = 1, 2, \ldots, n; j = 1, 2, \ldots, m.$$

The **covariance matrix** C is computed as:

$$C = \frac{1}{n-1} X_{\mathrm{c}}^T X_{\mathrm{c}}.$$

Each entry C_{jk} of the covariance matrix C is given by:

$$C_{jk} = \frac{1}{n-1} \sum_{i=1}^{n} (X_{ij} - \mu_j)(X_{ik} - \mu_k),$$

where C_{jj} represents the **variance** of variable j, - $C_{jk} (j \neq k)$ represents the **covariance** between variables j and k.

Using SVD, we can see that the singular values and the the first r columns of V are directly related to the eigenvalues and eigenvectors of C. The eigenvectors represent the principal components, and the eigenvalues indicate the variance along these components.