October 2024

Y. Oksamytnyi

MathsCodeBook

A Comprehensive Guide

YURIODEV

Contents

Chapter 1

Calculus and Basic Algebra

1.1 Limits and Continuity

1.1.1 Definition of a Limit

Definition 1.1 : Limit Definition.
A function $f(x)$ approaches the limit L as x approaches c if, for every number $\epsilon > 0$, there exists a number $\delta > 0$ such that whenever $0 < |x - c| < \delta$, we have $|f(x) - L| < \epsilon$.

1.1.2 Theorem: Intermediate Value Theorem

Theorem 1.1 : Intermediate Value Theorem . If a function f is continuous on the interval $[a, b]$, and $f(a)$ and $f(b)$ have opposite signs, then there exists some $x \in [a, b]$ such that $f(x) = 0$.

Exercise 1.1. *Use the Intermediate Value Theorem to show that the equation* $x^3 - x - 1 = 0$ *has a solution in the interval* $[1, 2]$.

1.1.3 Example: Calculating Limits

Example 1.1 .

Calculate the limit:

$$\lim_{x \to 1} \frac{x^2 - 1}{x - 1}$$

Solution: Factor the numerator:

$$\lim_{x \to 1} \frac{(x-1)(x+1)}{x-1} = \lim_{x \to 1} (x+1) = 2$$

1.1.4 Proposition: Continuity Implies Limit

Proposition 1.2. *For any rational function $f(x)$, if $f(x)$ is continuous at $x = a$, then:*

$$\lim_{x \to a} f(x) = f(a)$$

● **Observation .** *Even though a function can be continuous at a point, it may not be differentiable at that point. For example, $f(x) = |x|$ is continuous at $x = 0$ but not differentiable there.*

1.2 Differentiation

1.2.1 Definition: Derivative

Definition 1.2 : Derivative Definition.
The derivative of a function $f(x)$ at a point $x = a$ is defined as:

$$f'(a) = \lim_{h \to 0} \frac{f(a+h) - f(a)}{h}$$

This measures the rate of change of the function at that point.

1.2.2 Theorem: Power Rule for Derivatives

Theorem 1.3 : Power Rule . For any real number n, the derivative of $f(x) = x^n$ is:

$$f'(x) = nx^{n-1}$$

1.2.3 Corollary: Special Case of Power Rule

Corollary 1.4. *As a special case of the Power Rule, the derivative of $f(x) = x^2$ is:*

$$f'(x) = 2x$$

1.2.4 Exercise: Differentiation Practice

Exercise 1.2. *Differentiate the polynomial:*

$$f(x) = x^3 - 3x^2 + 2x$$

1.2.5 Python Code Snippet: Differentiation

Code Snippet 1.1 : Differentiating a Polynomial in Python.

```
import sympy as sp

# Define the variable and the function
x = sp.symbols('x')
f = x**3 - 3*x**2 + 2*x

# Differentiate the function
derivative = sp.diff(f, x)
print(f"The derivative of f(x) is: {derivative}")
```

1.2.6 Example: Derivative Calculation

Example 1.2 .

Find the derivative of the function:

$$f(x) = x^3 - 2x^2 + 3x - 1$$

Solution: Using the Power Rule:

$$f'(x) = 3x^2 - 4x + 3$$

1.3 Applications of Derivatives

1.3.1 Theorem: Critical Points and Extrema

Theorem 1.5 : Critical Points Theorem . Let $f(x)$ be a differentiable function. A point c is called a critical point if either $f'(c) = 0$ or $f'(c)$ does not exist.

Exercise 1.3. *Find the critical points of the function:*

$$f(x) = x^3 - 6x^2 + 9x$$

Then, determine whether each critical point corresponds to a local maximum, local minimum, or neither.

1.4 Algebraic Functions and Equations

1.4.1 Axiom: Properties of Real Numbers

Axiom 1.1 .
For any real numbers a and b, the sum $a+b$ is also a real number. This is known as closure under addition in real numbers.

1.4.2 Proposition: Solving Quadratic Equations

Proposition 1.6. *The solutions to the quadratic equation* $ax^2+bx+c=0$ *are given by the quadratic formula:*

$$x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$$

1.4.3 Example: Solving Quadratic Equations

Example 1.3 .

Solve the quadratic equation:

$$2x^2-4x+1=0$$

Solution: Using the quadratic formula:

$$x = \frac{-(-4) \pm \sqrt{(-4)^2-4(2)(1)}}{2(2)} = \frac{4 \pm \sqrt{16-8}}{4} = \frac{4 \pm \sqrt{8}}{4} = \frac{4 \pm 2\sqrt{2}}{4}$$

Thus, the solutions are:

$$x = 1 \pm \frac{\sqrt{2}}{2}$$

1.4.4 Python Code Snippet: Quadratic Solver

Code Snippet 1.2 : Solving Quadratics in Python.

```python
import sympy as sp

# Define the variable and the quadratic equation
x = sp.symbols('x')
quadratic_eq = 2*x**2 - 4*x + 1

```

```python
# Solve the quadratic equation
solutions = sp.solve(quadratic_eq, x)
print(f"The solutions to the quadratic equation are: {solutions}")
```

1.5 Trigonometric Limits

1.5.1 Theorem: Sine Limit

Theorem 1.7 : Sine Limit Theorem .

$$\lim_{x \to 0} \frac{\sin x}{x} = 1$$

Corollary 1.8. *From the limit of sine, we derive:*

$$\lim_{x \to 0} \frac{1 - \cos x}{x^2} = \frac{1}{2}$$

Exercise 1.4. *Evaluate the following limit:*

$$\lim_{x \to 0} \frac{\tan x}{x}$$

1.5.2 C++ Code Snippet: Calculating Sine Limit

Code Snippet 1.3 : Sine Limit Calculation in C++.

```cpp
#include <iostream>
#include <cmath>
using namespace std;

int main() {
    double x = 0.0001; // Approaching zero
    double sine_limit = sin(x) / x;

    cout << "The value of sin(x)/x as x approaches 0 is approximately: " <<
     sine_limit << endl;
    return 0;
}
```

Chapter 2

Introduction to Algorithms

2.1 Basics of Algorithms

2.1.1 Definition of an Algorithm

Definition 2.1 : Algorithm Definition.
An algorithm is a finite sequence of well-defined instructions to solve a problem or perform a computation. Each step must be precise and executable in a finite amount of time.

Algorithms are the foundation of programming and problem-solving in computer science. Their efficiency is often measured in terms of time complexity, which we introduce next.

2.1.2 Theorem: Big-O Notation

Theorem 2.1 : Big-O Notation . Let $f(n)$ and $g(n)$ be two non-negative functions. We say that $f(n)$ is $O(g(n))$ if there exist constants $c > 0$ and n_0 such that for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$.

Big-O notation is used to classify algorithms according to their worst-case performance as the input size grows. For example, linear search has a time complexity of $O(n)$, meaning that the time it takes to complete grows linearly with the input size.

2.1.3 Example: Time Complexity of Linear Search

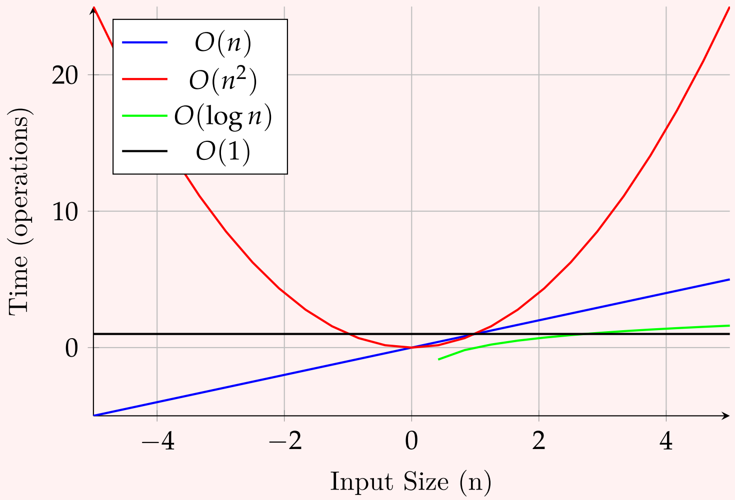
Example 2.1 : Linear Search Time Complexity.

The time complexity of the linear search algorithm is $O(n)$. In the worst case, the algorithm checks each element in the array, resulting in a linear number of operations.

2.1.4 Exercise: Analyze Linear Search

Exercise 2.1. *Implement the linear search algorithm in Python. Analyze its time complexity in both the best and worst cases.*

Example 2.2 : Time Complexity Chart.



2.2 Sorting Algorithms

2.2.1 Theorem: Time Complexity of Sorting Algorithms

Theorem 2.2 : Time Complexity of Bubble Sort . Bubble Sort is an $O(n^2)$ algorithm. In the worst case, Bubble Sort performs $n(n-1)/2$ comparisons.

2.2.2 Proposition: Bubble Sort Efficiency

Proposition 2.3. *In Bubble Sort, each pass through the array pushes the largest unsorted element to its correct position. This results in $O(n^2)$ time complexity, as every element may need to be compared multiple times.*

2.2.3 Exercise: Implement Bubble Sort

Exercise 2.2. *Implement Bubble Sort in Python. Analyze its performance in both the best and worst cases.*

2.2.4 Python Code Snippet: Bubble Sort

Code Snippet 2.1 : Implementing Bubble Sort in Python.

```python
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        # Last i elements are already sorted
        for j in range(0, n-i-1):
            # Swap if the element found is greater than the next element
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr

# Example usage
arr = [64, 34, 25, 12, 22, 11, 90]
sorted_arr = bubble_sort(arr)
print("Sorted array:", sorted_arr)
```

Example 2.3 : Bubble Sort Steps.

$$64, 34, 25, 12, 22, 11, 90$$
$$34, 25, 12, 22, 11, 64, 90$$
$$25, 12, 22, 11, 34, 64, 90$$
$$12, 22, 11, 25, 34, 64, 90$$
$$12, 11, 22, 25, 34, 64, 90$$
$$11, 12, 22, 25, 34, 64, 90$$

2.3 Recursive Algorithms

2.3.1 Theorem: Recursion

Theorem 2.4 : Recursion Theorem . A recursive algorithm solves a problem by breaking it down into smaller instances of the same problem, solving each instance recursively until reaching the base case.

2.3.2 Example: Factorial Function

Example 2.4 : Factorial Function.

The factorial of a number n is defined as:

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ n \cdot (n-1)! & \text{if } n > 0. \end{cases}$$

2.3.3 Exercise: Implement Recursive Factorial

Exercise 2.3. *Implement the factorial function in Python using recursion.*

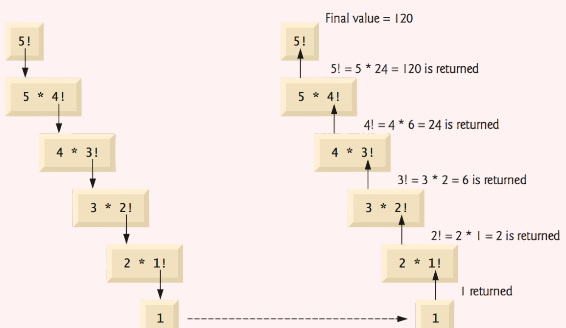
2.3.4 Python Code Snippet: Recursive Factorial

Code Snippet 2.2 : Recursive Factorial in Python.

```python
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)

# Example usage
n = 5
print(f"Factorial of {n} is {factorial(n)}")
```

Example 2.5 : Recursion Call Stack.



(a) Sequence of recursive calls (b) Values returned from each recursive call

2.4 Dynamic Programming

2.4.1 Theorem: Principle of Optimality

Theorem 2.5 : Bellman's Principle of Optimality . An optimal solution to a problem can be constructed from optimal solutions to its subproblems. This principle forms the basis of dynamic programming.

2.4.2 Example: Fibonacci Sequence

Example 2.6 : Fibonacci Sequence Example.

The Fibonacci sequence is defined as:

$$F(n) = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ F(n-1) + F(n-2) & \text{if } n \geq 2. \end{cases}$$

Using dynamic programming, we can compute $F(n)$ efficiently by storing previously computed values.

2.4.3 Exercise: Implement Fibonacci Sequence

Exercise 2.4. *Implement the Fibonacci sequence in Python using both recursion and dynamic programming. Compare the performance of both implementations.*

2.4.4 Python Code Snippet: Fibonacci Sequence (Dynamic Programming)

Code Snippet 2.3 : Fibonacci Sequence in Python (DP).

```python
def fibonacci_dp(n):
    # Base cases
    if n == 0:
        return 0
    elif n == 1:
        return 1

    # Initialize table for dynamic programming
    dp = [0] * (n + 1)
```

```python
    dp[1] = 1

    # Compute Fibonacci numbers iteratively
    for i in range(2, n + 1):
        dp[i] = dp[i-1] + dp[i-2]

    return dp[n]

# Example usage
n = 10
print(f"Fibonacci number at position {n} is {fibonacci_dp(n)}")
```

2.5 C++ Code Snippet: Sorting Algorithms

2.5.1 C++ Code Snippet: Quick Sort

Code Snippet 2.4 : Quick Sort in C++.

```cpp
#include <iostream>
using namespace std;

// Function to swap two elements
void swap(int* a, int* b) {
    int t = *a;
    *a = *b;
    *b = t;
}

// Partition function
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);
    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

// QuickSort function
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
```

```
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

// Driver code
int main() {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    quickSort(arr, 0, n - 1);
    cout << "Sorted array: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
    return 0;
}
```

Chapter 3

Linear Algebra and Matrices

3.1 Matrix Operations

3.1.1 Definition: Matrices

Definition 3.1 : Matrix Definition.
A matrix is a rectangular array of numbers arranged in rows and columns. The size or dimension of a matrix is given by the number of rows and columns. A matrix with m rows and n columns is called an $m \times n$ matrix.

3.1.2 Theorem: Properties of Matrix Operations

Theorem 3.1 : Matrix Addition and Multiplication . Let A and B be $m \times n$ matrices, and C an $n \times p$ matrix. The following properties hold:

- Matrix Addition: $(A+B)_{ij} = A_{ij} + B_{ij}$
- Scalar Multiplication: $(cA)_{ij} = c \cdot A_{ij}$
- Matrix Multiplication: $(AB)_{ij} = \sum_{k=1}^{n} A_{ik}B_{kj}$

3.1.3 Example: Matrix Multiplication

Example 3.1 : Matrix Multiplication.

Let $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ and $B = \begin{bmatrix} 2 & 0 \\ 1 & 2 \end{bmatrix}$. The product AB is calculated as:

$$AB = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 2 & 0 \\ 1 & 2 \end{bmatrix} = \begin{bmatrix} 1(2)+2(1) & 1(0)+2(2) \\ 3(2)+4(1) & 3(0)+4(2) \end{bmatrix} = \begin{bmatrix} 4 & 4 \\ 10 & 8 \end{bmatrix}$$

3.2 Determinants

3.2.1 Theorem: Properties of Determinants

Theorem 3.2 : Determinant Properties . Let A and B be square matrices. The determinant satisfies the following properties:

- $\det(A) = \det(A^T)$ (Determinant of transpose is the same as the determinant)
- $\det(AB) = \det(A)\det(B)$ (Multiplicative property)
- If A is triangular, then $\det(A)$ is the product of its diagonal entries

3.2.2 Example: Determinant of a 3x3 Matrix

Example 3.2 : Determinant Calculation.

Let $A = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 4 \\ 5 & 6 & 0 \end{bmatrix}$. The determinant of A is:

$$\det(A) = 1 \cdot \det \begin{bmatrix} 1 & 4 \\ 6 & 0 \end{bmatrix} - 2 \cdot \det \begin{bmatrix} 0 & 4 \\ 5 & 0 \end{bmatrix} + 3 \cdot \det \begin{bmatrix} 0 & 1 \\ 5 & 6 \end{bmatrix}$$

After simplifying:

$$\det(A) = 1(1 \cdot 0 - 4 \cdot 6) - 2(0 \cdot 0 - 5 \cdot 4) + 3(0 \cdot 6 - 5 \cdot 1) = -24 + 40 - 15 = 1$$

3.3 Inverse of a Matrix

3.3.1 Theorem: Inverse of a Matrix

Theorem 3.3 : Invertibility Condition . A square matrix A is invertible if and only if $\det(A) \neq 0$. The inverse of A is denoted by A^{-1} and satisfies the property $AA^{-1} = A^{-1}A = I$, where I is the identity matrix.

3.3.2 Example: Inverse of a 2x2 Matrix

Example 3.3 : Matrix Inverse.

Let $A = \begin{bmatrix} 4 & 7 \\ 2 & 6 \end{bmatrix}$. The inverse of A is calculated using the formula:

$$A^{-1} = \frac{1}{\det(A)} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

where $\det(A) = ad - bc$. For this matrix:

$$\det(A) = 4(6) - 7(2) = 10, \quad A^{-1} = \frac{1}{10} \begin{bmatrix} 6 & -7 \\ -2 & 4 \end{bmatrix} = \begin{bmatrix} 0.6 & -0.7 \\ -0.2 & 0.4 \end{bmatrix}$$

3.3.3 Python Code Snippet: Matrix Inversion

Code Snippet 3.1 : Matrix Inversion in Python.

```python
import numpy as np

A = np.array([[4, 7], [2, 6]])
A_inv = np.linalg.inv(A)
print("Inverse of A:\n", A_inv)
```

3.4 Eigenvalues and Eigenvectors

3.4.1 Definition: Eigenvalues and Eigenvectors

Definition 3.2 : Eigenvalue Definition.

Let A be an $n \times n$ matrix. A non-zero vector v is called an eigenvector of A if there exists a scalar λ, called the eigenvalue, such that $Av = \lambda v$.

3.4.2 Example: Eigenvalues of a 2x2 Matrix

Example 3.4 : Eigenvalue Calculation.

Let $A = \begin{bmatrix} 4 & 1 \\ 2 & 3 \end{bmatrix}$. The eigenvalues of A are found by solving the characteristic equation:

$$\det(A - \lambda I) = 0$$

For this matrix, the characteristic equation is:

$$\det \begin{bmatrix} 4-\lambda & 1 \\ 2 & 3-\lambda \end{bmatrix} = (4-\lambda)(3-\lambda) - 2 = \lambda^2 - 7\lambda + 10 = 0$$

The eigenvalues are $\lambda_1 = 5$ and $\lambda_2 = 2$.

3.4.3 Python Code Snippet: Eigenvalue Calculation

Code Snippet 3.2 : Eigenvalue and Eigenvector Calculation in Python.

```python
import numpy as np

A = np.array([[4, 1], [2, 3]])
eigenvalues, eigenvectors = np.linalg.eig(A)
print("Eigenvalues:\n", eigenvalues)
print("Eigenvectors:\n", eigenvectors)
```

3.5 Gaussian Elimination

3.5.1 Theorem: Gaussian Elimination

Theorem 3.4 : Gaussian Elimination . Gaussian elimination is a method for solving a system of linear equations. It works by transforming the system's augmented matrix into row echelon form, and then solving the system using back-substitution.

3.5.2 Example: Solving a System of Equations

Example 3.5 : Gaussian Elimination.

Solve the following system of equations using Gaussian elimination:

$$\begin{aligned} 2x + 3y - z &= 1 \\ 4x + y + 2z &= 2 \\ -2x + 7y + 3z &= 3 \end{aligned}$$

The augmented matrix is:

$$\left[\begin{array}{ccc|c} 2 & 3 & -1 & 1 \\ 4 & 1 & 2 & 2 \\ -2 & 7 & 3 & 3 \end{array}\right]$$

Apply row operations to reduce the matrix to row echelon form, and then solve the system using back-substitution.

3.5.3 Python Code Snippet: Gaussian Elimination

Code Snippet 3.3 : Gaussian Elimination in Python.

```python
import numpy as np

def gaussian_elimination(A, b):
    n = len(b)
    for i in range(n):
        A[i] = A[i] / A[i, i]  # Normalize pivot row
        for j in range(i + 1, n):
            A[j] = A[j] - A[j, i] * A[i]
    x = np.zeros(n)
    for i in range(n-1, -1, -1):
        x[i] = (b[i] - np.dot(A[i, i+1:], x[i+1:])) / A[i, i]
    return x

A = np.array([[2, 3, -1], [4, 1, 2], [-2, 7, 3]], dtype=float)
b = np.array([1, 2, 3], dtype=float)
solution = gaussian_elimination(A, b)
print("Solution:", solution)
```