
Project - 1 MA517M-Basic Programming Laboratory	Last Date: 14 November 2026
Name	Roll No.: MA26M103

Implementation of Riemann Integrable Functions using C++ and Classes

1. Design a C++ class `RiemannFunction` to represent a real-valued function.

$$f : [a, b] \rightarrow \mathbb{R}$$

And to numerically compute its Riemann sum and integral.

The class should:

- Store the interval endpoints a, b ,
- Store a function pointer or `std::function<double(double)> f`,
- Include parameters such as the number of subintervals N and partition type (left, right, midpoint).

2. Implement the following constructors and functions:

- Default constructor (for $f(x) = 0$ on $[0, 1]$),
- Parameterized constructor accepting (a, b, N, f) ,
- A method `evaluate(double x)` returning $f(x)$,
- A method `riemannSum()` that computes:

$$S_N(f) = \sum_{i=1}^N f(x_i^*) \Delta x,$$

where x_i^* depends on the partition rule,

- A method `integrate()` that approximates $\int_a^b f(x) dx$ using the Riemann sum.

3. Implement operator overloading to combine functions symbolically:

$$(f + g)(x) = f(x) + g(x)$$

$$(f - g)(x) = f(x) - g(x)$$

$$(c * f)(x) = c f(x)$$

using:

- `operator+` for addition of two functions,
- `operator-` for subtraction,
- `operator*` for scalar multiplication.

Example:

$$f(x) = x^2, \quad g(x) = \sin(x), \quad h = f + g \implies h(x) = x^2 + \sin(x)$$

4. Add methods to compute specific Riemann approximations:

- Left Riemann Sum,
- Right Riemann Sum,
- Midpoint Riemann Sum,
- Trapezoidal Rule (for comparison),
- Simpson's Rule (as higher-order approximation).

5. Implement a method `isRiemannIntegrable(double tol)` that:

- Computes upper and lower sums U_N, L_N ,
- Checks whether $|U_N - L_N| < \text{tol}$ for sufficiently large N ,
- Returns `true` if f is approximately Riemann integrable.

$$L_N(f) = \sum_{i=1}^N m_i \Delta x, \quad U_N(f) = \sum_{i=1}^N M_i \Delta x, \quad m_i = \inf_{x \in I_i} f(x), \quad M_i = \sup_{x \in I_i} f(x)$$

6. Create a test program to:

- Define the following functions using `lambda` expressions:

$$f(x) = x^2, \quad g(x) = \sin(x), \quad h(x) = |x|$$

- Compute $\int_0^1 f(x) dx$, $\int_0^\pi g(x) dx$, $\int_{-1}^1 h(x) dx$ using different Riemann rules.
- Test whether $h(x) = |x|$ is Riemann integrable on $[-1, 1]$.

7. (Optional Extension:) Implement operator overloading for function composition:

$$(f \circ g)(x) = f(g(x))$$

using `operator()` or a custom operator.

8. (Optional Extension:) Overload comparison operators to compare integrals of two functions:

$$f == g \iff \int_a^b f(x) dx = \int_a^b g(x) dx,$$
$$f > g \iff \int_a^b f(x) dx > \int_a^b g(x) dx.$$

Project - 2: Match-3 Game (Candy Crush Variant) Using C++ Classes

Problem Statement

Design and implement a **Match-3 Game** using **C++ classes**. The game consists of a grid of colored tiles (or symbols), where the player swaps adjacent tiles to form a sequence of three or more identical

tiles in a row or column. When a match is formed, the tiles disappear, points are awarded, and new tiles fall from the top to fill empty spaces. The project should utilize object-oriented programming concepts such as classes, objects, encapsulation, and methods to handle grid updates, scoring, and user interaction.

Project Requirements

1. Create a **Tile** class to represent an individual tile in the grid.
 - (a) Data member: type or color of the tile.
 - (b) Methods: `getType()`, `setType()`.
2. Create a **Board** class to represent the game grid.
 - (a) Initialize the grid with random tiles.
 - (b) Display the current state of the grid in a readable format.
 - (c) Detect and remove matches of three or more tiles in a row or column.
 - (d) Drop new tiles from the top to fill empty spaces.
 - (e) Include a method to check if possible moves exist.
3. Create a **Game** class to manage gameplay.
 - (a) Display options for the user: **Play** or **Solution**.
 - (b) Allow the user to swap adjacent tiles using input commands.
 - (c) Keep track of the player's score.
 - (d) Allow the user to quit and confirm before exiting.
 - (e) Optionally, allow the user to reset the board or play multiple rounds.
4. Ensure proper encapsulation of tile and board operations within the respective classes.

Suggested Class Structure

1. **Tile Class:**
 - Data member: `int type` or `char color`
 - Methods: `getType()`, `setType()`
2. **Board Class:**
 - Data member: 2D array of **Tile** objects
 - Methods: `initializeBoard()`, `displayBoard()`, `detectMatches()`, `removeMatches()`, `dropTiles()`, `hasPossibleMoves()`
3. **Game Class:**
 - Data member: **Board** object, user choice, score
 - Methods: `playGame()`, `showSolution()`, `processSwap(int row1, int col1, int row2, int col2)`, `confirmQuit()`, `resetBoard()`

Reference

For more details about Match-3 games and mechanics, visit: https://en.wikipedia.org/wiki/Tile-matching_video_game