1. Continuous Mean Value Calculation
Approach – 1

Date: 17/01/2024

Explanation

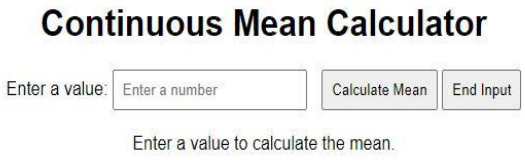
Index.html

The provided HTML code creates a web page for a "Continuous Mean Calculator." The page features an input field where users can enter numeric values, along with two buttons: "Calculate Mean" and "End Input." The layout is aesthetically styled using inline CSS, ensuring a clean and readable design. The JavaScript section of the code is responsible for handling user interactions and asynchronous communication with the server.

The `calculateMean()` function is triggered by the "Calculate Mean" button and asynchronously sends a POST request to the server (`/calculate`). It retrieves the user-entered value, updates the result display area with the calculated mean received from the server, and logs any errors to the console.

The `endInput()` function, activated by the "End Input" button, resets the display to the initial message prompting users to enter a value. Simultaneously, it sends a POST request to the server (`/reset`) to reset ongoing calculations and displays a response alert. Similar to `calculateMean()`, error handling is in place to log any issues.

The user interface encourages a straightforward experience: users input values, calculate the mean asynchronously with the click of a button, and have the option to reset calculations. The use of asynchronous communication with the server enhances responsiveness, providing an interactive and user-friendly environment for continuous mean calculation.



Steps

1. **HTML Structure:**
    - The HTML document starts with the usual structure, specifying the document type, language, and containing a `<head>` and `<body>` section.
    - The `<head>` section includes meta tags for character set and viewport settings. It also sets the title of the page to "Continuous Mean Calculator."
    - Inside the `<head>` section, there's an inline CSS style block defining styles for the body, input, button, and result display area.

2. **Body Content:**
    - The `<body>` section contains a heading (`<h1>`) indicating the title of the calculator: "Continuous Mean Calculator."
    - Following the heading, there is an input field (`<input>`) labeled "Enter a value" and two buttons (`<button>`):
    - "Calculate Mean" button: Triggers the `calculateMean()` JavaScript function.
    - "End Input" button: Triggers the `endInput()` JavaScript function.
    - A `<div>` element with the id "result" is present to display the calculated mean or instructions to enter a value.

3. **Inline CSS Styling:**
    - The inline CSS styling provides a clean and simple layout for the webpage. It adjusts margins, padding, and text alignment for a visually appealing user interface.

4. **JavaScript Code:**
    - The JavaScript code is embedded within a `<script>` block.
    - The `calculateMean()` function is an asynchronous function that retrieves the input value from the user, sends a POST request to the server using the Fetch API, and updates the result display area with the calculated mean received from the server. Any errors during this process are caught and logged to the console.
    - The `endInput()` function is also asynchronous and is triggered by the "End Input" button. It resets the result display to the initial message, indicating that the user should enter a value to calculate the mean. Additionally, it sends a POST request to the server to reset calculations, displaying an alert with the server's response. Like `calculateMean()`, errors are caught and logged.

5. **Server Communication:**
    - The Fetch API is used for asynchronous communication with the server. Two endpoints are utilized:
    - `/calculate`: For calculating the mean based on the user-provided value.
    - `/reset`: For resetting the calculation, triggered by the "End Input" button.

6. **User Interaction:**
    - Users can input values in the provided text field and trigger calculations using the "Calculate Mean" button. The "End Input" button resets the ongoing calculation, providing a user-friendly interface for continuous mean calculation.

App.py

The `app.py` code represents a Flask web application designed to serve as the backend for a Continuous Mean Calculator. The code utilizes the Flask framework to handle HTTP requests and responses. It includes global variables, `total_sum` and `count`, which maintain the running sum and count of entered values. The `is_valid_float` function checks whether an input value is a valid floating-point number by looking for the presence of a decimal point.

The application defines three routes: the main page ("/"), a route for calculating the mean ("/calculate"), and a route to reset calculations ("/reset"). The main page renders an HTML template, and the "/calculate" route handles POST requests with JSON payloads containing numerical values. It calculates the mean, updates global variables, and returns the current mean. The "/reset" route resets the global variables to zero, providing a way to restart the calculation process.

The script includes an execution block to run the Flask application when executed directly, and it runs in debug mode to facilitate development and debugging. It's important to note that this backend code is intended to work in conjunction with a corresponding HTML template (such as "index.html") that handles user interaction and asynchronous communication with the backend using the Fetch API. The overall structure allows for a dynamic and interactive continuous mean calculation experience on the frontend.

Index.html code

```
<!-- index.html -->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Continuous Mean Calculator</title>
    <!-- Inline CSS for styling -->
    <style>
        body {
            font-family: Arial, sans-serif;
            text-align: center;
            margin: 50px;
        }

        footer {
            position: fixed;
            bottom: 0;
            left: 0;
            width: 100%;
            padding: 10px;
            background-color: rgba(255, 255, 255, 0.8);
            z-index: 1000;
            display: flex;
            justify-content: center; /* Center the content horizontally */
            align-items: center; /* Center the content vertically */
        }

        footer img {
            height: 100px; /* Adjust the height as needed */
            width: 900px;
            margin-right: 20px;
        }

        input {
            padding: 8px;
            margin-right: 10px;
        }

        button {
            padding: 8px;
        }

        #result {
            margin-top: 20px;
```

```
        }
    </style>
</head>
<body>

    <h1>Continuous Mean Calculator</h1>

    <!-- Input field and two buttons for user interaction -->
    <label for="value">Enter a value:</label>
    <input type="text" id="value" placeholder="Enter a number">
    <button onclick="calculateMean()">Calculate Mean</button>
    <button onclick="endInput()">End Input</button>

    <!-- Display area for the result -->
    <div id="result">Enter a value to calculate the mean.</div>

    <footer>
        <img src="https://www.itgroup-
europe.com/fileadmin/_processed_/a/a/csm_logo_cab83c2c3a.jpg" alt="Footer
Logo">
    </footer>

<!-- JavaScript code using Fetch API for asynchronous communication -->
    <script>
        // Asynchronous function to calculate mean
        async function calculateMean() {
            // Get the input value from the user
            var inputValue = document.getElementById('value').value;

            try {
                // Send a POST request to the server using Fetch API
                var response = await fetch('/calculate', {
                    method: 'POST',
                    headers: {
                        'Content-Type': 'application/json'
                    },
                    body: JSON.stringify({ value: inputValue })
                });

                // Get the result from the response and display it
                var result = await response.text();
                document.getElementById('result').innerHTML = result;
            } catch (error) {
                // Handle errors, if any
                console.error('Error:', error);
            }
        }
```

```
        // Function to end input (triggered by the "End Input" button)
        async function endInput() {
            // Clear and reset the current mean value to 0
            document.getElementById('result').innerHTML = "Enter a value to
calculate the mean.";

            // Send a request to the server to reset calculations
            try {
                var response = await fetch('/reset', {
                    method: 'POST',
                    headers: {
                        'Content-Type': 'application/json'
                    }
                });

                // Get the result from the response and display it
                var result = await response.text();
                alert(result);
            } catch (error) {
                // Handle errors, if any
                console.error('Error:', error);
            }
        }
    </script>



</body>
</html>
```

App.py code

```python
# app.py (Python backend)
from flask import Flask, request, render_template

app = Flask(__name__)

# Variables to store the sum and count
total_sum = 0
count = 0

# Function to check if the input value is a valid floating-point number
def is_valid_float(value):
    return '.' in value

# Route for the main page
@app.route("/")
def index():
    return render_template("index.html")

# Route for calculating the mean
@app.route("/calculate", methods=["POST"])
def calculate_mean():
    global total_sum, count

    try:
        input_value = request.json["value"]

        # Check if the entered value is a valid floating-point number
        if not is_valid_float(input_value):
            raise ValueError("Please enter a valid floating-point number.")

        new_value = float(input_value)

    except ValueError:
        return "Invalid input. Please enter a valid floating-point number."

    # Update variables
    total_sum += new_value
    count += 1

    # Calculate the current mean
    current_mean = total_sum / count

    return f"Current mean: {current_mean}"

# Route to reset calculations (triggered by the "End Input" button)
```

```python
@app.route("/reset", methods=["POST"])
def reset_calculations():
    global total_sum, count

    # Reset total sum and count to 0
    total_sum = 0
    count = 0

    return "Calculations reset. You can start again."

if __name__ == "__main__":
    app.run(debug=True)
```

1. Continuous Mean Value Calculation
Approach - 2

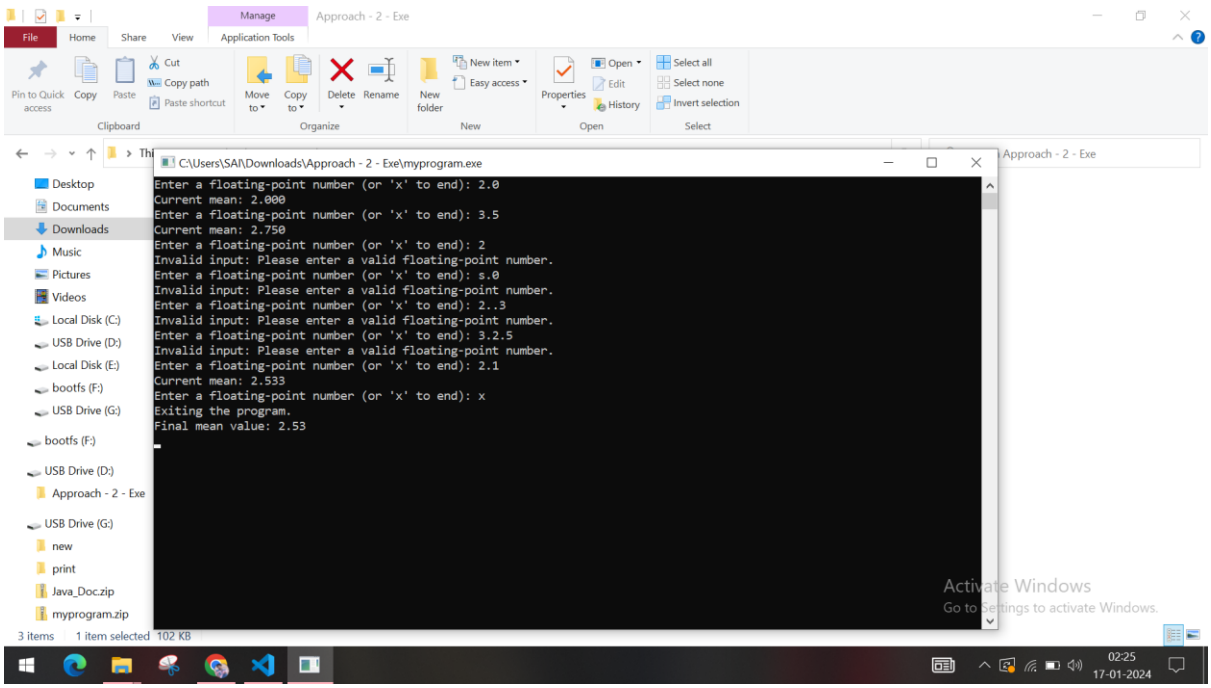
Explanation

This C++ program is designed to interactively calculate the mean (average) of a series of floating-point numbers entered by the user. The program consists of a main function and a supporting function, `isValidFloatingPoint`, for input validation.

In the `isValidFloatingPoint` function, a string representing user input is processed using an `std::isstringstream`. The function checks whether the input can be successfully converted to a double and ensures that there are no remaining characters after the conversion. Additionally, it verifies if the input contains a decimal point, returning true if these conditions are met, indicating a valid floating-point number, and false otherwise.

The main function initializes variables for the total sum (`total_sum`) and the count of entered numbers (`count`). It enters a continuous input loop, prompting the user to enter floating-point numbers until the user inputs 'x' to terminate the process. Within the loop, user input is validated using the `isValidFloatingPoint` function. If the input is deemed invalid, an error message is displayed, and the loop continues to the next iteration. Valid input is converted to a double using `std::stod`, and the program updates the sum and count variables. The current mean is then calculated and displayed, with precision control using `std::fixed` and `std::setprecision`.

Exception handling is implemented to catch and handle cases where the conversion of input to a double fails due to an invalid argument. The final mean value is displayed after the user terminates input, and the program introduces a deliberate delay of 5 seconds using `std::this_thread::sleep_for` before exiting. This allows the user a brief moment to review the final result before the program concludes. The program encapsulates input validation, exception handling, and interactive calculation of the mean, providing a straightforward utility for users to analyze a series of floating-point numbers.



Steps

1. **Input Validation Function (isValidFloatingPoint)**
   The program begins with a function that validates whether a given string represents a valid floating-point number. This is achieved by using an std::istringstream to attempt conversion to a double and checking for any remaining characters. The presence of a decimal point is also verified. This function is crucial for ensuring that user input is valid before attempting further processing.

2. **Main Function Initialization**
   The main function initializes variables total_sum and count to maintain the cumulative sum of entered numbers and the count of entered values, respectively. These variables are used to calculate the mean.

3. **Continuous Input Loop**
   The program enters a continuous loop where the user is prompted to input floating-point numbers. The loop continues until the user enters 'x' to signify the end of input.

4. **User Input and Termination Check**
   Within the loop, the program uses std::getline to obtain a string input from the user. If the user enters 'x', the program displays an exit message and breaks out of the loop, concluding the input phase.

5. **Input Validation and Error Handling**
   User input is then validated using the isValidFloatingPoint function. If the input is invalid, an error message is displayed, and the program skips to the next iteration of the loop.

6. **Conversion and Update**
   Valid input is converted to a double using std::stod, and the program updates the cumulative sum and count variables. The current mean is calculated and displayed with a precision of three decimal places.

7. **Exception Handling**
   The program includes exception handling to catch cases where the conversion of input to a double fails due to an invalid argument. In such cases, an error message is displayed, allowing the program to gracefully handle exceptional input scenarios.

8. **Final Mean Calculation, Display and Exit**
   After the user terminates input, the program calculates the final mean value based on the accumulated sum and count. This value is displayed with a precision of two decimal places. And the program introduces a 5-second delay using std::this_thread::sleep_for before exiting.

Code

```
#include <iostream>
#include <string>
#include <sstream>
#include <iomanip>
#include <limits>
#include <chrono>
#include <thread>

bool isValidFloatingPoint(const std::string& input) {
    std::istringstream iss(input);
    double test;
    char leftover;

    // Attempt to read a double followed by any remaining characters
    if (iss >> test && !(iss >> leftover)) {
        // Check if the input contains a decimal point
        return (input.find('.') != std::string::npos);
    }

    return false;
}

int main() {
    // Variable declaration
    double total_sum = 0.0;
    int count = 0;

    // Continuous input loop
    while (true) {
        // Get a new value from the user
        std::string input_value;
        std::cout << "Enter a floating-point number (or 'x' to end): ";
        std::getline(std::cin, input_value);

        // Check if the user wants to end the input
        if (input_value == "x") {
            std::cout << "Exiting the program." << std::endl;
            break;
        }

        // Validate the input
        if (!isValidFloatingPoint(input_value)) {
            std::cerr << "Invalid input: Please enter a valid floating-point
number." << std::endl;
            continue;
        }
```

```
        try {
            // Convert input to a double
            double new_value = std::stod(input_value);

            // Update variables
            total_sum += new_value;
            count++;

            // Calculate and display the current mean
            double current_mean = total_sum / count;
            std::cout << "Current mean: " << std::fixed <<
std::setprecision(3) << current_mean << std::endl;

        } catch (const std::invalid_argument& e) {
            std::cerr << "Invalid input: " << e.what() << std::endl;
        }
    }

    // Display the final mean value
    double final_mean = (count > 0) ? total_sum / count : 0.0;
    std::cout << "Final mean value: " << std::fixed << std::setprecision(2) <<
final_mean << std::endl;

    // Introduce a delay before exiting (e.g., 5 seconds)
    std::this_thread::sleep_for(std::chrono::seconds(5));

    return 0;
}
```

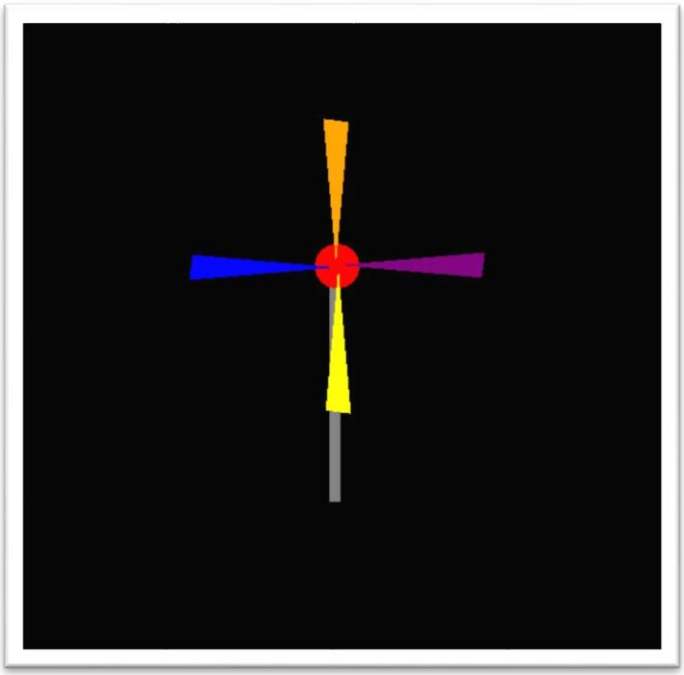
2. Windmill project in pyleap

Date: 17/01/2024

Explanation

This Python code utilizes the `pyleap` library to create a graphical animation representing a windmill. The graphical elements, including a background rectangle, text, and various shapes to depict the windmill components, are defined. These components include a rectangular body, a circular midpoint, and four triangular handles of different colors. The handles are positioned at different angles to simulate rotation. The windmill's rotation speed is read from a file specified by `speed_file_path`. The `read_speed` function attempts to retrieve the speed from the file, ensuring it falls within a valid range of 1 (Low) to 10 (High).

The main animation function, `windmill`, repeatedly draws the graphical elements and updates the rotation angles of the handles based on the read rotation speed. The animation loop is established with the `repeat` function, and the entire application is executed with `run()`. In essence, this code creates a visually appealing windmill animation where the rotation speed can be dynamically adjusted through an external file.



Steps

1. Importing Modules

   from pyleap import *
   import time
   The code imports necessary modules from the pyleap library for graphical elements and animation. The time module is also imported, though it's not explicitly used in the provided code.

2. File Path and Speed Limit

   speed_file_path =’ ‘
   This variable holds the path to a file (speed.txt) that presumably contains the rotation speed of the windmill. The windmill's rotation speed is expected to be an integer between 1 and 10.

3. Reading Windmill Rotation Speed from File

   def read_speed():
   # ...
   This function attempts to read the windmill rotation speed from the specified file (speed.txt). It checks if the value is within the allowed range (1 to 10) and handles various error cases.

4. Drawing Windmill and Updating Rotation

   def windmill(dt):
   # ...
   This function is responsible for drawing the graphical elements of the windmill and updating the rotation of the handles based on the windmill's rotation speed.

5. Animation Loop

   repeat(windmill)

   This line repeats the windmill function, creating an animation loop.

6. Running the Application

   run()

   Finally, this line executes the pyleap application, running the graphical animation.

Code:

```
# Importing necessary modules from the pyleap library
from pyleap import *

# Setting the size of the window
window.set_size(600, 600)

# Creating graphical elements for the windmill project
bg = Rectangle(0, 0, 600, 600, 'black')  # Background rectangle
#txt = Text("Windmill project in pyleap", 130, 540, 20, "black")  # Text
element
body = Rectangle(285, 160, 10, 200, 'gray')  # Body of the windmill
mid = Circle(292, 372, 20, 'red')  # Circle representing the midpoint of the
windmill

# Handles of the windmill
handle1 = Triangle(293, 374, 392, 460, 403, 440, 'purple')
handle1.rotation = 0
handle1.set_anchor(292, 372)

handle2 = Triangle(293, 374, 392, 460, 403, 440, 'orange')
handle2.rotation = 90
handle2.set_anchor(292, 372)

handle3 = Triangle(293, 374, 392, 460, 403, 440, 'blue')
handle3.rotation = 180
handle3.set_anchor(292, 372)

handle4 = Triangle(293, 374, 392, 460, 403, 440, 'yellow')
handle4.rotation = 270
handle4.set_anchor(292, 372)

# Path to the file containing windmill rotation speed
speed_file_path = 'C:/Users/z004tu0x/Desktop/IT_Group/2.
Graphical_Wind_Turbine/speed.txt'

# Function to read windmill rotation speed from the file
def read_speed():
    try:
        with open(speed_file_path, 'r') as file:
            speed_str = file.readline().strip()
            if speed_str:
                speed = int(speed_str)
                if 1 <= speed <= 10:
                    return speed
                else:
                    print('Please make sure to enter the Speed limit  between
1 - Low and 10 - High')
```

```
                    exit()

            else:
                print("Error: Empty line or non-integer value in the file.")
                return None
    except Exception as e:
        print(f"Error reading speed from file: {e}")
        return None

# Function to draw the windmill and update rotation based on speed
def windmill(dt):
    bg.draw()
    #txt.draw()
    body.draw()
    mid.draw()
    handle1.draw()
    handle2.draw()
    handle3.draw()
    handle4.draw()

    speed = read_speed()

    if speed is not None:
        # Update handle rotation based on speed
        handle1.rotation += -speed
        handle2.rotation += -speed
        handle3.rotation += -speed
        handle4.rotation += -speed

# Repeating the windmill function to create animation
repeat(windmill)

# Running the pyleap application
run()
```