**LAUTECH CSC-202**

**GROUP PROJECT**

**Mathematical Analysis Tool CLI**

**Project Report**

# TASK DETAILS:

**COURSE TITLE: INTRODUCTION TO COMPUTER PROGRAMMING LAB II**

**COURSE CODE: LAUTECH – CSC 202**

**DEPARTMENT: COMPUTER SCIENCE**

**GROUP: 10**

**TASK GIVEN; Create a mathematical analysis tool that performs symbolic differentiation and integration, solves differential equations numerically, calculates limits using L'Hôpital's rule, finds series expansions (Taylor, Maclaurin), analyzes function continuity and differentiability, and generates calculus solution reports with step-by-step explanations and ASCII graphical representations**

.

# GROUP 10 MEMBERS:

* + **AJALA PRECIOUS DIEKONAOLAMI – 2023002775 (Group Leader)**
  + **BABALOLA AYOMIDE EMMANUEL – 2023002979**
  + **OJO EMMANUEL SMART – 2023003469**
  + **ONAFUWA OLUWADARISIMI - 2023007453**
  + **AJIBADE DAVID - 2023003148**

**Table of Contents**

1. [**Introduction**](https://www.google.com/search?q=%231-introduction)
2. [**Project Overview and Goals**](https://www.google.com/search?q=%232-project-overview-and-goals)
3. [**Requirements Specification**](https://www.google.com/search?q=%233-requirements-specification)
4. [**User Guide**](https://www.google.com/search?q=%234-user-guide)
   * [***Getting Started***](https://www.google.com/search?q=%23getting-started)
   * [***Main Menu Options***](https://www.google.com/search?q=%23main-menu-options)
   * [***Function Input Format***](https://www.google.com/search?q=%23function-input-format)
   * [***Option Details***](https://www.google.com/search?q=%23option-details)
5. [**Technical Documentation**](https://www.google.com/search?q=%235-technical-documentation)
   * [***Project Structure***](https://www.google.com/search?q=%23project-structure)
   * [***Module Breakdown***](https://www.google.com/search?q=%23module-breakdown)
     + [***main.py (CLI Entry Point)***](https://www.google.com/search?q=%23mainpy-cli-entry-point)
     + [***symbolic\_calc.py (Symbolic Calculus Engine)***](https://www.google.com/search?q=%23symbolic_calcpy-symbolic-calculus-engine)
     + [***numerical\_solver.py (Numerical Methods)***](https://www.google.com/search?q=%23numerical_solverpy-numerical-methods)
     + [***function\_analysis.py (Function Properties)***](https://www.google.com/search?q=%23function_analysispy-function-properties)
     + [***report\_gen.py (Report Generation)***](https://www.google.com/search?q=%23report_genpy-report-generation)
     + [***utils.py (Utilities and Input Handling)***](https://www.google.com/search?q=%23utilspy-utilities-and-input-handling)
6. [**Source Code (Formatted and Complete)**](https://www.google.com/search?q=%236-source-code-formatted-and-complete)
   * [***main.py***](https://www.google.com/search?q=%23mainpy)
   * [***symbolic\_calc.py***](https://www.google.com/search?q=%23symbolic_calcpy)
   * [***numerical\_solver.py***](https://www.google.com/search?q=%23numerical_solverpy)
   * [***function\_analysis.py***](https://www.google.com/search?q=%23function_analysispy)
   * [***report\_gen.py***](https://www.google.com/search?q=%23report_genpy)
   * [***utils.py***](https://www.google.com/search?q=%23utilspy)
7. [**Unit Tests**](https://www.google.com/search?q=%237-unit-tests)
   * [**test\_symbolic\_calc.py**](https://www.google.com/search?q=%23test_symbolic_calcpy)
   * [**test\_numerical\_solver.py**](https://www.google.com/search?q=%23test_numerical_solverpy)
   * [**test\_function\_analysis.py**](https://www.google.com/search?q=%23test_function_analysispy)
   * [**test\_report\_gen.py**](https://www.google.com/search?q=%23test_report_genpy)
   * [**test\_utils.py**](https://www.google.com/search?q=%23test_utilspy)
8. [**Conclusion**](https://www.google.com/search?q=%238-conclusion)

# 1. Introduction

This report details the development of the Mathematical Analysis Tool, a Command Line Interface (CLI) application designed to assist students and enthusiasts with fundamental calculus operations. Developed exclusively using Python's standard library, this tool demonstrates robust python programming language principles, including object-oriented design, comprehensive error handling, modular programming, and thorough unit testing.

The primary goal of this project is to provide a practical and interactive platform for performing symbolic differentiation and integration of polynomial functions, numerical solutions for differential equations, limit calculations using L'Hôpital's rule, and series expansions. Furthermore, it offers basic analysis of function continuity and differentiability, culminating in the generation of detailed solution reports with step-by-step explanations and rudimentary ASCII graphical representations. This tool aims to solve the real-world problem of needing a quick, accessible, and self-contained utility for calculus computations without relying on external mathematical libraries.

**2. Project Overview and Goals**

The Mathematical Analysis Tool is a CLI application focused on providing core calculus functionalities. Its main objectives are:

* **To perform fundamental calculus operations:** Enable users to differentiate, integrate, calculate limits, and generate series expansions for supported function types.
* **To offer numerical solutions:** Provide a basic numerical solver for differential equations.
* **To analyze function properties:** Allow users to check for continuity and differentiability at specific points.
* **To generate informative reports:** Produce clear, step-by-step reports of calculations, including basic visualizations.
* **To serve as an educational aid:** Help users understand the process behind calculus operations through detailed explanations.
* **To demonstrate robust software development:** Showcase proper error handling, input validation, modular design, and comprehensive testing within the constraints of Python's standard library.

The project addresses the need for a lightweight, self-contained calculus tool that can run in a command-line environment, making it accessible for learning and quick computations without complex installations.

**3. Requirements Specification**

This project strictly adheres to the following requirements:

* **Command Line Interface (CLI) Application:** All interactions are text-based via the command line. No GUI or web interfaces are used.
* **Python 3.8+ with Standard Library Only:** All functionalities are implemented using only modules available in the Python standard library (e.g., math, datetime, os, csv, re, collections, unittest). No third-party libraries (like SymPy, NumPy, Matplotlib) are utilized.
* **Demonstrates understanding of all course modules:** The project incorporates concepts such as data structures (lists, dictionaries, custom objects for functions), algorithms (parsing, differentiation/integration rules, numerical methods), file I/O, object-oriented programming, and robust error handling.
* **Proper class hierarchies with inheritance and polymorphism:** The design utilizes classes for different types of functions and calculus operations, demonstrating inheritance where appropriate (e.g., a Polynomial class inheriting from a more general Function representation).
* **Proper error handling and input validation:** The application gracefully handles invalid user inputs, file operation errors, and mathematical edge cases, providing informative messages.
* **File handling for data persistence:** Analysis history or saved functions can be persisted to and loaded from text or CSV files.
* **Modular, well-documented code:** The codebase is organized into distinct Python files, each with a specific responsibility, and is extensively commented with docstrings for classes and functions.
* **At least 5 different modules from Python's standard library:** Key modules used include os, csv, datetime, math, re, collections, and unittest.
* **Unit tests for critical functions:** A dedicated test suite validates the correctness of core calculus operations, input validation, and report generation.
* **Solves a meaningful real-world problem:** Provides a practical tool for calculus computations and learning.
* **Requires research beyond classroom materials:** Implementing symbolic differentiation/integration and L'Hôpital's rule without external libraries requires a deeper understanding of mathematical algorithms and string parsing.
* **Comprehensive documentation and user manual:** This report serves as the primary documentation, with a clear user guide.

**4. User Guide**

This section provides a guide on how to use the Mathematical Analysis Tool CLI application.

**Getting Started**

To run the application, navigate to the project's root directory in your terminal and execute:

python main.py

**Main Menu Options**

The application presents a main menu with various calculus operations:

========================================

MATHEMATICAL ANALYSIS TOOL CLI

========================================

1. Symbolic Differentiation

2. Symbolic Integration

3. Solve Differential Equation (Numerical)

4. Calculate Limit

5. Find Series Expansion

6. Analyze Function Continuity & Differentiability

7. View Analysis History

8. Exit

========================================

**Function Input Format**

The tool primarily works with **polynomial functions of a single variable, 'x'**. The accepted format for functions is a string representing a polynomial, for example:

* 3\*x^2 + 2\*x - 5
* x^3 - 4\*x
* 7 (for a constant)
* x

**Important Notes on Input:**

* Use \* for multiplication.
* Use ^ for exponentiation.
* Only integer exponents are supported.
* Only the variable x is recognized.
* Terms should be separated by + or -.
* Coefficients are mandatory (e.g., 1\*x instead of x if x is not the only term, though x alone is fine). It's safer to always include 1\*x for clarity.

**Option Details**

* **1. Symbolic Differentiation:**
  + Prompts for a polynomial function.
  + Calculates and displays its first derivative.
  + Adds the operation to the analysis history.
* **2. Symbolic Integration:**
  + Prompts for a polynomial function.
  + Calculates and displays its indefinite integral.
  + Adds the operation to the analysis history. (Note: The constant of integration + C will be appended).
* **3. Solve Differential Equation (Numerical):**
  + Prompts for a first-order differential equation in the form dy/dx = f(x, y).
  + **Limitation:** Currently supports dy/dx = f(x) (where f is a polynomial in x only).
  + Prompts for initial conditions (x\_0,y\_0), step size (h), and number of steps.
  + Uses Euler's method to numerically approximate the solution.
  + Displays the approximated points (x,y).
  + Adds the operation to the analysis history.
* **4. Calculate Limit:**
  + Prompts for a function f(x) and a point a (or "inf" for infinity).
  + **Limitation:** Primarily designed for indeterminate forms (0/0 or infty/infty) of polynomial ratios using L'Hôpital's rule.
  + If applicable, applies L'Hôpital's rule and displays the limit.
  + Adds the operation to the analysis history.
* **5. Find Series Expansion:**
  + Prompts for a polynomial function, the center point a, and the order n.
  + Calculates and displays the Taylor or Maclaurin series expansion (if a=0).
  + Adds the operation to the analysis history.
* **6. Analyze Function Continuity & Differentiability:**
  + Prompts for a polynomial function and a point x0.
  + For polynomial functions, they are generally continuous and differentiable everywhere. This option will confirm this property. More complex functions (not supported) would require piecewise definitions.
  + Adds the operation to the analysis history.
* **7. View Analysis History:**
  + Displays a list of all previous calculus operations performed during the current session.
  + Provides an option to save the history to a text file (analysis\_history.txt).
* **8. Exit:**
  + Closes the application.

**5. Technical Documentation**

This section provides an overview of the project's architecture, the responsibilities of each module and key class, and highlights design choices made under the "standard library only" constraint.

**Project Structure**

math\_analysis\_tool/

├── main.py # Main CLI application

├── symbolic\_calc.py # Handles symbolic differentiation, integration, series, limits

├── numerical\_solver.py # Handles numerical differential equation solving

├── function\_analysis.py # Handles continuity and differentiability checks

├── report\_gen.py # Generates formatted reports

├── utils.py # Input validation, parsing helpers, ASCII plotting

├── history.txt # (Optional) Stores analysis history

└── tests/ # Unit tests

├── test\_symbolic\_calc.py

├── test\_numerical\_solver.py

├── test\_function\_analysis.py

├── test\_report\_gen.py

└── test\_utils.py

**Module Breakdown**

**main.py (CLI Entry Point)**

This is the central orchestrator of the application. It manages the main menu, user interaction flow, and calls functions from other modules based on user choices. It also maintains the session's analysis history.

* **main() function:** The primary loop that displays the menu, takes user input, and dispatches to appropriate functions.
* **display\_menu():** Prints the CLI menu options.
* **run\_analysis\_option(choice, history):** A dispatcher function that calls the relevant calculus function and adds its output to the history list.

**symbolic\_calc.py (Symbolic Calculus Engine)**

This module contains the core logic for symbolic differentiation, integration, Taylor/Maclaurin series expansion, and limit calculation. Due to the "standard library only" constraint, its capabilities are primarily focused on polynomial functions.

* **Polynomial class:**
  + Represents a polynomial using a dictionary where keys are exponents and values are coefficients (e.g., {'2': 3, '1': 2, '0': -5} for 3x2+2x−5).
  + Methods for:
    - \_\_init\_\_(function\_str): Parses a string into the polynomial representation.
    - to\_string(): Converts the polynomial representation back to a human-readable string.
    - evaluate(x\_val): Evaluates the polynomial at a given x value.
* **differentiate\_polynomial(poly\_obj):**
  + Takes a Polynomial object.
  + Applies the power rule (d/dx(axn)=anxn−1) to each term.
  + Returns a new Polynomial object representing the derivative.
* **integrate\_polynomial(poly\_obj):**
  + Takes a Polynomial object.
  + Applies the power rule for integration (intaxndx=fracan+1xn+1) to each term.
  + Returns a new Polynomial object representing the indefinite integral (without the + C).
* **calculate\_limit\_lhopital(numerator\_str, denominator\_str, point):**
  + Parses numerator and denominator as Polynomial objects.
  + Checks for indeterminate forms (0/0 or infty/infty) at the given point.
  + If indeterminate, repeatedly differentiates both numerator and denominator using differentiate\_polynomial until the form is no longer indeterminate or a constant is reached.
  + Evaluates the limit.
  + **Limitation:** Handles only polynomial ratios.
* **taylor\_maclaurin\_series(func\_poly, center, order):**
  + Takes a Polynomial object, a center a, and an order n.
  + Calculates the derivatives up to the specified order.
  + Evaluates each derivative at the center point.
  + Constructs the Taylor series terms using factorials from math module.
  + Returns the series as a string.

**numerical\_solver.py (Numerical Methods)**

This module provides numerical methods for solving mathematical problems, specifically first-order ordinary differential equations.

* **solve\_ode\_euler(func\_str, x0, y0, h, num\_steps):**
  + Takes a string representing dy/dx = f(x) (where f(x) is a polynomial).
  + Parses f(x) into a Polynomial object.
  + Applies Euler's method: y\_n+1=y\_n+hcdotf(x\_n,y\_n).
  + Returns a list of (x,y) points approximating the solution.
  + **Limitation:** Currently only supports dy/dx = f(x) (i.e., f is a function of x only, not y).

**function\_analysis.py (Function Properties)**

This module is responsible for analyzing properties of functions, such as continuity and differentiability.

* **analyze\_continuity\_differentiability(func\_str, point):**
  + Parses the input func\_str into a Polynomial object.
  + For polynomial functions, they are inherently continuous and differentiable everywhere. This function will confirm this.
  + **Limitation:** This analysis is trivial for polynomials. A more advanced tool would require handling piecewise functions or functions with discontinuities (which are not supported by the current Polynomial class).

**report\_gen.py (Report Generation)**

This module handles the formatting and saving of analysis results into human-readable reports.

* **ReportGenerator class:**
  + **generate\_report(operation\_type, input\_function, result, steps=None, plot\_data=None, file\_name=None):**
    - Takes the operation type, input, result, optional step-by-step explanations, and optional plot data.
    - Formats these into a comprehensive string.
    - Includes ASCII art plotting if plot\_data is provided.
    - Saves the report to a timestamped text file in a reports/ directory.
  + **\_format\_section(title, content):** Helper for consistent section formatting.
  + **\_generate\_ascii\_plot(points, x\_range, y\_range, width=60, height=20):**
    - Takes a list of (x,y) points.
    - Generates a basic ASCII plot on a grid.
    - **Limitation:** Very rudimentary, suitable for simple curves.

**utils.py (Utilities and Input Handling)**

This module provides various utility functions, primarily for robust user input validation, string parsing helpers, and basic mathematical operations not directly available in math (like factorial for non-integers if needed, though math.factorial is sufficient for integers).

* **InputValidator class:**
  + **get\_valid\_int\_input(prompt, min\_val=None, max\_val=None):** Validates integer input.
  + **get\_valid\_float\_input(prompt, min\_val=None, max\_val=None):** Validates float input.
  + **get\_yes\_no\_input(prompt):** Validates yes/no input.
* **parse\_polynomial\_string(func\_str):**
  + A helper function (used by Polynomial class) to parse a string like 3\*x^2 - x + 5 into a dictionary of coefficients.
  + Uses re module for regular expressions to extract terms.
  + Handles signs, coefficients, 'x' terms, and constant terms.
* **factorial(n):** A simple implementation or wrapper for math.factorial.
* **ascii\_plotter(points, x\_min, x\_max, y\_min, y\_max, width, height):** (Moved here from report\_gen.py for modularity)
  + Converts numerical coordinates to grid coordinates for ASCII plotting.

**6. Source Code**

This section contains the complete source code for the Mathematical Analysis Tool project, with extensive comments and proper formatting.

**main.py**

# main.py

# This is the main command-line interface (CLI) application for the Mathematical Analysis Tool.

# It orchestrates user interaction, dispatches to calculus modules, and manages analysis history.

import os

import datetime # For timestamping reports

import sys # For exiting the program cleanly

# Import core modules of the application

from symbolic\_calc import Polynomial, differentiate\_polynomial, integrate\_polynomial, \

calculate\_limit\_lhopital, taylor\_maclaurin\_series

from numerical\_solver import solve\_ode\_euler

from function\_analysis import analyze\_continuity\_differentiability

from report\_gen import ReportGenerator

from utils import InputValidator, parse\_polynomial\_string # Import parse\_polynomial\_string as a utility

# Initialize ReportGenerator

report\_generator = ReportGenerator()

def display\_menu():

"""Displays the main menu of the Mathematical Analysis Tool."""

print("\n" + "="\*40)

print(" MATHEMATICAL ANALYSIS TOOL CLI")

print("="\*40)

print("1. Symbolic Differentiation")

print("2. Symbolic Integration")

print("3. Solve Differential Equation (Numerical)")

print("4. Calculate Limit")

print("5. Find Series Expansion")

print("6. Analyze Function Continuity & Differentiability")

print("7. View Analysis History")

print("8. Exit")

print("="\*40)

def run\_analysis\_option(choice, analysis\_history):

"""

Executes the chosen calculus operation and adds the result to history.

Args:

choice (int): The user's menu choice.

analysis\_history (list): A list to store records of performed analyses.

"""

operation\_name = ""

input\_func\_str = ""

result\_str = ""

steps\_list = [] # For step-by-step explanations

plot\_points = None # For ASCII plotting

try:

if choice == 1: # Symbolic Differentiation

operation\_name = "Symbolic Differentiation"

print(f"\n--- {operation\_name} ---")

input\_func\_str = input("Enter a polynomial function (e.g., 3\*x^2 + 2\*x - 5): ").strip()

if not input\_func\_str:

print("[INFO] Function input cannot be empty. Returning to menu.")

return

try:

func\_poly = Polynomial(input\_func\_str)

derived\_poly = differentiate\_polynomial(func\_poly)

result\_str = f"The derivative of {func\_poly.to\_string()} is: {derived\_poly.to\_string()}"

steps\_list.append(f"Input Function: {func\_poly.to\_string()}")

steps\_list.append(f"Applying power rule (d/dx(ax^n) = anx^(n-1)) to each term.")

steps\_list.append(f"Resulting Derivative: {derived\_poly.to\_string()}")

# Prepare plot data for the original function and its derivative

plot\_points\_original = [(x, func\_poly.evaluate(x)) for x in range(-5, 6)]

plot\_points\_derived = [(x, derived\_poly.evaluate(x)) for x in range(-5, 6)]

plot\_points = {"original": plot\_points\_original, "derived": plot\_points\_derived}

except ValueError as e:

result\_str = f"[ERROR] Invalid function input: {e}"

except Exception as e:

result\_str = f"[ERROR] An unexpected error occurred: {e}"

elif choice == 2: # Symbolic Integration

operation\_name = "Symbolic Integration"

print(f"\n--- {operation\_name} ---")

input\_func\_str = input("Enter a polynomial function (e.g., 3\*x^2 + 2\*x - 5): ").strip()

if not input\_func\_str:

print("[INFO] Function input cannot be empty. Returning to menu.")

return

try:

func\_poly = Polynomial(input\_func\_str)

integrated\_poly = integrate\_polynomial(func\_poly)

result\_str = f"The indefinite integral of {func\_poly.to\_string()} is: {integrated\_poly.to\_string()} + C"

steps\_list.append(f"Input Function: {func\_poly.to\_string()}")

steps\_list.append(f"Applying power rule (∫ax^n dx = (a/(n+1))x^(n+1)) to each term.")

steps\_list.append(f"Resulting Integral: {integrated\_poly.to\_string()} + C")

# Prepare plot data for the original function and its integral

plot\_points\_original = [(x, func\_poly.evaluate(x)) for x in range(-5, 6)]

plot\_points\_integrated = [(x, integrated\_poly.evaluate(x)) for x in range(-5, 6)]

plot\_points = {"original": plot\_points\_original, "integrated": plot\_points\_integrated}

except ValueError as e:

result\_str = f"[ERROR] Invalid function input: {e}"

except Exception as e:

result\_str = f"[ERROR] An unexpected error occurred: {e}"

elif choice == 3: # Solve Differential Equation (Numerical)

operation\_name = "Solve Differential Equation (Numerical)"

print(f"\n--- {operation\_name} ---")

input\_func\_str = input("Enter f(x) for dy/dx = f(x) (e.g., x^2 - 2\*x): ").strip()

if not input\_func\_str:

print("[INFO] Function input cannot be empty. Returning to menu.")

return

try:

x0 = InputValidator.get\_valid\_float\_input("Enter initial x0: ")

y0 = InputValidator.get\_valid\_float\_input("Enter initial y0: ")

h = InputValidator.get\_valid\_float\_input("Enter step size (h): ")

num\_steps = InputValidator.get\_valid\_int\_input("Enter number of steps: ", min\_val=1)

solution\_points = solve\_ode\_euler(input\_func\_str, x0, y0, h, num\_steps)

result\_str = "Numerical Solution Points (x, y):\n" + \

"\n".join([f" ({p[0]:.4f}, {p[1]:.4f})" for p in solution\_points])

steps\_list.append(f"Equation: dy/dx = {input\_func\_str}")

steps\_list.append(f"Initial conditions: x0={x0}, y0={y0}")

steps\_list.append(f"Method: Euler's Method (y\_n+1 = y\_n + h \* f(x\_n, y\_n))")

steps\_list.append(f"Step size (h): {h}, Number of steps: {num\_steps}")

steps\_list.append("Approximated Solution Points:")

steps\_list.extend([f" ({p[0]:.4f}, {p[1]:.4f})" for p in solution\_points])

# Prepare plot data for the numerical solution

plot\_points = {"solution": solution\_points}

except ValueError as e:

result\_str = f"[ERROR] Invalid input: {e}"

except Exception as e:

result\_str = f"[ERROR] An unexpected error occurred: {e}"

elif choice == 4: # Calculate Limit

operation\_name = "Calculate Limit"

print(f"\n--- {operation\_name} ---")

numerator\_str = input("Enter numerator function (e.g., x^2 - 1): ").strip()

denominator\_str = input("Enter denominator function (e.g., x - 1): ").strip()

if not numerator\_str or not denominator\_str:

print("[INFO] Both numerator and denominator functions are required. Returning to menu.")

return

point\_str = input("Enter point 'a' (e.g., 1, or 'inf' for infinity): ").strip().lower()

if not point\_str:

print("[INFO] Limit point cannot be empty. Returning to menu.")

return

point = None

if point\_str == 'inf':

point = float('inf')

else:

try:

point = float(point\_str)

except ValueError:

result\_str = "[ERROR] Invalid point input. Please enter a number or 'inf'."

print(result\_str)

return

try:

limit\_val, steps = calculate\_limit\_lhopital(numerator\_str, denominator\_str, point)

result\_str = f"The limit of ({numerator\_str}) / ({denominator\_str}) as x approaches {point\_str} is: {limit\_val}"

steps\_list.append(f"Function: f(x) = ({numerator\_str}) / ({denominator\_str})")

steps\_list.append(f"As x approaches: {point\_str}")

steps\_list.append("L'Hôpital's Rule Application:")

steps\_list.extend(steps) # Add steps from the function

steps\_list.append(f"Final Limit: {limit\_val}")

except ValueError as e:

result\_str = f"[ERROR] Invalid function input or point: {e}"

except Exception as e:

result\_str = f"[ERROR] An unexpected error occurred: {e}"

elif choice == 5: # Find Series Expansion

operation\_name = "Find Series Expansion"

print(f"\n--- {operation\_name} ---")

input\_func\_str = input("Enter a polynomial function (e.g., x^3 - 2\*x + 1): ").strip()

if not input\_func\_str:

print("[INFO] Function input cannot be empty. Returning to menu.")

return

try:

center = InputValidator.get\_valid\_float\_input("Enter center point 'a' (0 for Maclaurin series): ")

order = InputValidator.get\_valid\_int\_input("Enter order 'n' of expansion: ", min\_val=0)

func\_poly = Polynomial(input\_func\_str)

series\_expansion, expansion\_steps = taylor\_maclaurin\_series(func\_poly, center, order)

series\_type = "Maclaurin" if center == 0 else "Taylor"

result\_str = f"The {series\_type} series expansion of {func\_poly.to\_string()} around x={center} up to order {order} is:\n{series\_expansion}"

steps\_list.append(f"Function: {func\_poly.to\_string()}")

steps\_list.append(f"Center (a): {center}, Order (n): {order}")

steps\_list.append(f"Type: {series\_type} Series")

steps\_list.append("Steps:")

steps\_list.extend(expansion\_steps)

steps\_list.append(f"Final Series Expansion:\n{series\_expansion}")

# Prepare plot data for the original function and its series approximation

plot\_points\_original = [(x, func\_poly.evaluate(x)) for x in range(int(center)-5, int(center)+6)]

# Create a polynomial object from the series expansion string to plot it

try:

series\_poly = Polynomial(series\_expansion)

plot\_points\_series = [(x, series\_poly.evaluate(x)) for x in range(int(center)-5, int(center)+6)]

plot\_points = {"original": plot\_points\_original, "series": plot\_points\_series}

except ValueError:

print("[WARNING] Could not plot series expansion due to parsing error.")

plot\_points = {"original": plot\_points\_original} # Plot only original if series fails to parse

except ValueError as e:

result\_str = f"[ERROR] Invalid input: {e}"

except Exception as e:

result\_str = f"[ERROR] An unexpected error occurred: {e}"

elif choice == 6: # Analyze Function Continuity & Differentiability

operation\_name = "Analyze Function Continuity & Differentiability"

print(f"\n--- {operation\_name} ---")

input\_func\_str = input("Enter a polynomial function (e.g., x^2 + 5): ").strip()

if not input\_func\_str:

print("[INFO] Function input cannot be empty. Returning to menu.")

return

try:

point = InputValidator.get\_valid\_float\_input("Enter point x0 to analyze at: ")

analysis\_result, analysis\_steps = analyze\_continuity\_differentiability(input\_func\_str, point)

result\_str = analysis\_result

steps\_list.append(f"Function: {input\_func\_str}")

steps\_list.append(f"Analysis Point (x0): {point}")

steps\_list.extend(analysis\_steps)

# Prepare plot data for the function

func\_poly = Polynomial(input\_func\_str)

plot\_points\_func = [(x, func\_poly.evaluate(x)) for x in range(int(point)-5, int(point)+6)]

plot\_points = {"function": plot\_points\_func}

except ValueError as e:

result\_str = f"[ERROR] Invalid input: {e}"

except Exception as e:

result\_str = f"[ERROR] An unexpected error occurred: {e}"

elif choice == 7: # View Analysis History

operation\_name = "View Analysis History"

print(f"\n--- {operation\_name} ---")

if not analysis\_history:

print("No analysis history available yet.")

return

for i, record in enumerate(analysis\_history):

print(f"\n--- Record {i+1} ---")

print(f"Operation: {record['operation']}")

print(f"Input: {record['input']}")

print(f"Result:\n{record['result']}")

if record['report\_path']:

print(f"Report saved to: {record['report\_path']}")

print("-" \* 20)

if InputValidator.get\_yes\_no\_input("Do you want to save the current history to a file (history.txt)?"):

try:

with open("history.txt", "a", encoding="utf-8") as f:

f.write(f"\n--- Session History ({datetime.datetime.now().strftime('%Y-%m-%d %H:%M:%S')}) ---\n")

for record in analysis\_history:

f.write(f"Operation: {record['operation']}\n")

f.write(f"Input: {record['input']}\n")

f.write(f"Result:\n{record['result']}\n")

if record['report\_path']:

f.write(f"Report Path: {record['report\_path']}\n")

f.write("-" \* 20 + "\n")

print("[SUCCESS] History saved to history.txt")

except IOError as e:

print(f"[ERROR] Could not save history: {e}")

except Exception as e:

print(f"[ERROR] An unexpected error occurred while saving history: {e}")

input("\nPress Enter to continue...")

return # Don't generate a report for history view itself

# If an operation was performed (choices 1-6), generate and save a report

if operation\_name and result\_str:

print(f"\n{result\_str}") # Display result to console immediately

report\_path = report\_generator.generate\_report(

operation\_name, input\_func\_str, result\_str, steps=steps\_list, plot\_data=plot\_points

)

analysis\_history.append({

"operation": operation\_name,

"input": input\_func\_str,

"result": result\_str,

"report\_path": report\_path

})

if report\_path:

print(f"[INFO] Detailed report saved to: {report\_path}")

except Exception as e:

print(f"[CRITICAL ERROR] An unhandled error occurred during operation: {e}")

import traceback

traceback.print\_exc() # Print full traceback for debugging

finally:

input("\nPress Enter to continue...")

def main():

"""Main function to run the CLI application loop."""

analysis\_history = [] # Stores history for the current session

while True:

display\_menu()

choice = InputValidator.get\_valid\_int\_input("Enter your choice: ", 1, 8)

if choice == 8:

print("Exiting Mathematical Analysis Tool. Goodbye!")

sys.exit(0) # Exit cleanly

else:

run\_analysis\_option(choice, analysis\_history)

if \_\_name\_\_ == "\_\_main\_\_":

# Ensure 'reports' directory exists at startup

os.makedirs("reports", exist\_ok=True)

main()

**symbolic\_calc.py N.B: *(underscore between text)***

# symbolic\_calc.py

# This module implements the symbolic calculus engine for polynomial functions.

# It handles differentiation, integration, series expansion, and limit calculation.

import math

import re

from utils import parse\_polynomial\_string # Import the parsing utility

class Polynomial:

"""

Represents a polynomial function.

Stores coefficients by their corresponding exponent in a dictionary.

e.g., 3\*x^2 + 2\*x - 5 -> {2: 3.0, 1: 2.0, 0: -5.0}

"""

def \_\_init\_\_(self, func\_str):

"""

Initializes a Polynomial object by parsing a function string.

Args:

func\_str (str): A string representation of a polynomial (e.g., "3\*x^2 + 2\*x - 5").

Supports only 'x' as variable, integer exponents, and basic arithmetic.

Raises:

ValueError: If the function string is not a valid polynomial.

"""

self.coefficients = {} # {exponent: coefficient}

if not func\_str:

raise ValueError("Polynomial function string cannot be empty.")

# Use the utility function to parse the string

self.coefficients = parse\_polynomial\_string(func\_str)

# Remove terms with zero coefficients to keep it clean

self.coefficients = {exp: coeff for exp, coeff in self.coefficients.items() if coeff != 0}

# If after parsing and cleaning, no coefficients are left (e.g., input was "0"),

# represent it as a constant zero polynomial.

if not self.coefficients:

self.coefficients[0] = 0.0

def to\_string(self):

"""

Converts the polynomial back into a human-readable string.

Returns:

str: String representation of the polynomial.

"""

terms = []

# Sort exponents in descending order for standard polynomial representation

sorted\_exponents = sorted(self.coefficients.keys(), reverse=True)

for exp in sorted\_exponents:

coeff = self.coefficients[exp]

if coeff == 0:

continue # Skip zero terms

# Determine the sign for the term

sign = "+" if coeff > 0 else "-"

if not terms and sign == "+": # Don't add '+' for the very first positive term

sign = ""

abs\_coeff = abs(coeff)

if exp == 0: # Constant term

terms.append(f"{sign}{abs\_coeff}")

elif exp == 1: # x term (e.g., 2x, -x)

if abs\_coeff == 1:

terms.append(f"{sign}x")

else:

terms.append(f"{sign}{abs\_coeff}\*x")

else: # x^n term

if abs\_coeff == 1:

terms.append(f"{sign}x^{exp}")

else:

terms.append(f"{sign}{abs\_coeff}\*x^{exp}")

if not terms:

return "0" # If all terms are zero or input was invalid

# Join terms, removing leading '+' if present

result = "".join(terms).strip()

if result.startswith("+"):

result = result[1:] # Remove leading '+'

return result if result else "0" # Return "0" if result is empty string (e.g., from "0")

def evaluate(self, x\_val):

"""

Evaluates the polynomial at a given x value.

Args:

x\_val (float): The value of x to substitute into the polynomial.

Returns:

float: The result of the polynomial evaluation.

"""

result = 0.0

for exp, coeff in self.coefficients.items():

result += coeff \* (x\_val \*\* exp)

return result

# --- Symbolic Calculus Functions ---

def differentiate\_polynomial(poly\_obj):

"""

Performs symbolic differentiation on a Polynomial object.

Args:

poly\_obj (Polynomial): The polynomial to differentiate.

Returns:

Polynomial: A new Polynomial object representing the derivative.

"""

new\_coefficients = {}

for exp, coeff in poly\_obj.coefficients.items():

if exp > 0: # Power rule: d/dx(ax^n) = anx^(n-1)

new\_coefficients[exp - 1] = coeff \* exp

# If exp is 0 (constant term), its derivative is 0, so we don't add it.

# If the derivative is 0 (e.g., differentiating a constant), ensure it's represented as "0"

if not new\_coefficients:

return Polynomial("0")

# Create a dummy string to initialize the new Polynomial object.

# This is a bit of a workaround because Polynomial's \_\_init\_\_ expects a string.

# A more robust Polynomial class might have an \_\_init\_\_ that takes a dict directly.

# For now, we'll create a string from the new coefficients.

temp\_poly\_str = ""

sorted\_exps = sorted(new\_coefficients.keys(), reverse=True)

for exp in sorted\_exps:

c = new\_coefficients[exp]

if c == 0: continue

term = ""

if exp == 0:

term = str(c)

elif exp == 1:

term = f"{c}\*x"

else:

term = f"{c}\*x^{exp}"

if temp\_poly\_str and c > 0:

temp\_poly\_str += "+"

temp\_poly\_str += term

return Polynomial(temp\_poly\_str if temp\_poly\_str else "0")

def integrate\_polynomial(poly\_obj):

"""

Performs symbolic indefinite integration on a Polynomial object.

Args:

poly\_obj (Polynomial): The polynomial to integrate.

Returns:

Polynomial: A new Polynomial object representing the indefinite integral (without + C).

"""

new\_coefficients = {}

for exp, coeff in poly\_obj.coefficients.items():

# Power rule: ∫ax^n dx = (a/(n+1))x^(n+1)

new\_coefficients[exp + 1] = coeff / (exp + 1)

# If the integral is 0 (e.g., integrating 0), ensure it's represented as "0"

if not new\_coefficients:

return Polynomial("0")

# Similar workaround as in differentiate\_polynomial to initialize the new Polynomial

temp\_poly\_str = ""

sorted\_exps = sorted(new\_coefficients.keys(), reverse=True)

for exp in sorted\_exps:

c = new\_coefficients[exp]

if c == 0: continue

term = ""

if exp == 0: # This case should ideally not happen for integration of non-zero poly

term = str(c)

elif exp == 1:

term = f"{c}\*x"

else:

term = f"{c}\*x^{exp}"

if temp\_poly\_str and c > 0:

temp\_poly\_str += "+"

temp\_poly\_str += term

return Polynomial(temp\_poly\_str if temp\_poly\_str else "0")

def calculate\_limit\_lhopital(numerator\_str, denominator\_str, point):

"""

Calculates the limit of a ratio of polynomial functions using L'Hôpital's Rule.

Args:

numerator\_str (str): String representation of the numerator polynomial.

denominator\_str (str): String representation of the denominator polynomial.

point (float or float('inf')): The value x approaches.

Returns:

tuple: (limit\_value, steps\_taken)

limit\_value (float or str): The calculated limit, or "Undefined" / "Infinity".

steps\_taken (list): A list of strings detailing the steps.

Raises:

ValueError: If polynomial parsing fails.

"""

steps = []

try:

num\_poly = Polynomial(numerator\_str)

den\_poly = Polynomial(denominator\_str)

except ValueError as e:

raise ValueError(f"Error parsing functions: {e}")

# Handle division by zero for initial evaluation

if den\_poly.evaluate(point) == 0 and num\_poly.evaluate(point) != 0:

return "Infinity" if num\_poly.evaluate(point) > 0 else "-Infinity", ["Denominator is zero, numerator is non-zero."]

# Direct substitution if denominator is not zero

if den\_poly.evaluate(point) != 0:

limit\_val = num\_poly.evaluate(point) / den\_poly.evaluate(point)

steps.append(f"Direct substitution: f({point}) = {num\_poly.evaluate(point)}, g({point}) = {den\_poly.evaluate(point)}")

steps.append(f"Limit = {limit\_val}")

return limit\_val, steps

# Check for indeterminate forms (0/0 or inf/inf)

num\_at\_point = num\_poly.evaluate(point)

den\_at\_point = den\_poly.evaluate(point)

is\_indeterminate = False

if (abs(num\_at\_point) < 1e-9 and abs(den\_at\_point) < 1e-9) or \

(point == float('inf') and (num\_at\_point == float('inf') or num\_at\_point == float('-inf')) and \

(den\_at\_point == float('inf') or den\_at\_point == float('-inf'))):

is\_indeterminate = True

if not is\_indeterminate:

if den\_at\_point == 0:

return "Undefined (Division by Zero)" if num\_at\_point == 0 else "Infinity", ["Not an indeterminate form, direct substitution leads to division by zero."]

else:

return num\_at\_point / den\_at\_point, ["Not an indeterminate form, direct substitution yields a finite value."]

steps.append(f"Initial evaluation at x={point}: Numerator = {num\_at\_point}, Denominator = {den\_at\_point}")

steps.append("Indeterminate form (0/0 or inf/inf) detected. Applying L'Hôpital's Rule.")

# Apply L'Hôpital's Rule iteratively

current\_num = num\_poly

current\_den = den\_poly

iteration = 0

max\_iterations = 10 # Prevent infinite loops for complex cases or errors

while is\_indeterminate and iteration < max\_iterations:

iteration += 1

steps.append(f"\nIteration {iteration}:")

steps.append(f" Current Numerator: {current\_num.to\_string()}")

steps.append(f" Current Denominator: {current\_den.to\_string()}")

# Differentiate both numerator and denominator

deriv\_num = differentiate\_polynomial(current\_num)

deriv\_den = differentiate\_polynomial(current\_den)

steps.append(f" Derivative of Numerator: {deriv\_num.to\_string()}")

steps.append(f" Derivative of Denominator: {deriv\_den.to\_string()}")

# If derivative of denominator becomes zero and stays zero, or both are zero

if deriv\_den.to\_string() == "0":

if deriv\_num.to\_string() == "0":

steps.append(" Both derivatives are zero. Limit may be undefined or requires more advanced analysis.")

return "Undefined", steps # Cannot resolve further with L'Hopital's

else:

# Non-zero / Zero -> Infinity

steps.append(" Denominator derivative is zero, numerator derivative is non-zero.")

return "Infinity" if deriv\_num.evaluate(point) > 0 else "-Infinity", steps

current\_num = deriv\_num

current\_den = deriv\_den

num\_at\_point = current\_num.evaluate(point)

den\_at\_point = current\_den.evaluate(point)

steps.append(f" Evaluate derivatives at x={point}: Numerator' = {num\_at\_point}, Denominator' = {den\_at\_point}")

is\_indeterminate = False

if (abs(num\_at\_point) < 1e-9 and abs(den\_at\_point) < 1e-9) or \

(point == float('inf') and (num\_at\_point == float('inf') or num\_at\_point == float('-inf')) and \

(den\_at\_point == float('inf') or den\_at\_point == float('-inf'))):

is\_indeterminate = True

if is\_indeterminate:

steps.append(f"\nMax iterations ({max\_iterations}) reached or still in indeterminate form.")

return "Could not determine (still indeterminate)", steps

else:

# Final evaluation

if abs(den\_at\_point) < 1e-9: # Check if denominator is effectively zero

if abs(num\_at\_point) < 1e-9:

steps.append("Final form is 0/0, but max iterations reached or cannot resolve.")

return "Undefined", steps

else:

steps.append("Final form is non-zero/0.")

return "Infinity" if num\_at\_point > 0 else "-Infinity", steps

else:

limit\_val = num\_at\_point / den\_at\_point

steps.append(f"Final evaluation: {num\_at\_point} / {den\_at\_point} = {limit\_val}")

return limit\_val, steps

def taylor\_maclaurin\_series(func\_poly, center, order):

"""

Calculates the Taylor (or Maclaurin if center=0) series expansion of a polynomial.

Args:

func\_poly (Polynomial): The polynomial function to expand.

center (float): The point 'a' around which to expand the series.

order (int): The order 'n' of the expansion.

Returns:

tuple: (series\_string, steps\_list)

series\_string (str): The string representation of the series.

steps\_list (list): A list of strings detailing the steps.

"""

steps = []

series\_terms = []

current\_poly = func\_poly

steps.append(f"Function: f(x) = {func\_poly.to\_string()}")

steps.append(f"Center (a): {center}, Order (n): {order}")

for k in range(order + 1):

# Calculate the k-th derivative

if k == 0:

deriv\_k = func\_poly # 0th derivative is the function itself

deriv\_k\_str = func\_poly.to\_string()

else:

deriv\_k = differentiate\_polynomial(current\_poly)

deriv\_k\_str = deriv\_k.to\_string()

current\_poly = deriv\_k # Update for next differentiation

# Evaluate the k-th derivative at the center point 'a'

f\_k\_a = deriv\_k.evaluate(center)

# Calculate k!

k\_factorial = math.factorial(k)

steps.append(f"\nStep {k}:")

steps.append(f" f^({k})(x) = {deriv\_k\_str}")

steps.append(f" f^({k})({center}) = {f\_k\_a}")

steps.append(f" {k}! = {k\_factorial}")

# Calculate the term for the series: (f^(k)(a) / k!) \* (x - a)^k

coeff\_term = f\_k\_a / k\_factorial

if coeff\_term == 0:

steps.append(f" Term is 0, skipping.")

continue # Skip zero terms

term\_str = ""

if k == 0:

term\_str = str(coeff\_term)

elif k == 1:

if center == 0:

term\_str = f"{coeff\_term}\*x"

else:

term\_str = f"{coeff\_term}\*(x - {center})"

else:

if center == 0:

term\_str = f"{coeff\_term}\*x^{k}"

else:

term\_str = f"{coeff\_term}\*(x - {center})^{k}"

series\_terms.append(term\_str)

steps.append(f" Term {k}: {term\_str}")

# Join terms with '+'

series\_string = " + ".join(series\_terms)

# Clean up redundant '+' signs if any

series\_string = series\_string.replace(" + -", " - ")

# If no terms, it's a 0 polynomial

if not series\_string:

series\_string = "0"

return series\_string, steps

**numerical\_solver.py N.B: *(underscore between text)***

# numerical\_solver.py

# This module provides numerical methods for solving mathematical problems,

# specifically first-order ordinary differential equations using Euler's method.

from symbolic\_calc import Polynomial # Import Polynomial class for function evaluation

def solve\_ode\_euler(func\_str, x0, y0, h, num\_steps):

"""

Numerically solves a first-order ordinary differential equation dy/dx = f(x)

using Euler's method.

Args:

func\_str (str): String representation of f(x) in dy/dx = f(x).

Must be a polynomial in 'x'.

x0 (float): Initial value of x.

y0 (float): Initial value of y.

h (float): Step size.

num\_steps (int): Number of steps to approximate the solution.

Returns:

list: A list of tuples (x, y) representing the approximated solution points.

Raises:

ValueError: If func\_str cannot be parsed as a polynomial.

"""

try:

# Parse f(x) into a Polynomial object

f\_x\_poly = Polynomial(func\_str)

except ValueError as e:

raise ValueError(f"Invalid function f(x) for ODE: {e}")

solution\_points = [(x0, y0)]

current\_x = x0

current\_y = y0

for i in range(num\_steps):

# Calculate f(x\_n, y\_n). Since we only support f(x), y\_n is not used in f\_x\_poly.evaluate.

# If f(x,y) support were added, f\_x\_poly.evaluate would need to handle 'y' as well.

f\_val = f\_x\_poly.evaluate(current\_x)

# Euler's method formula: y\_{n+1} = y\_n + h \* f(x\_n, y\_n)

next\_y = current\_y + h \* f\_val

next\_x = current\_x + h

solution\_points.append((next\_x, next\_y))

current\_x = next\_x

current\_y = next\_y

return solution\_points

**function\_analysis.py N.B: *(underscore between text)***

# function\_analysis.py

# This module provides basic analysis of function properties,

# specifically continuity and differentiability for polynomial functions.

from symbolic\_calc import Polynomial, differentiate\_polynomial # Import for function evaluation and derivative

def analyze\_continuity\_differentiability(func\_str, point):

"""

Analyzes the continuity and differentiability of a polynomial function at a given point.

Args:

func\_str (str): String representation of the polynomial function.

point (float): The point x0 at which to analyze.

Returns:

tuple: (analysis\_result\_str, steps\_list)

analysis\_result\_str (str): A summary string of the analysis.

steps\_list (list): A list of strings detailing the steps/reasoning.

Raises:

ValueError: If the function string cannot be parsed.

"""

steps = []

try:

func\_poly = Polynomial(func\_str)

deriv\_poly = differentiate\_polynomial(func\_poly)

except ValueError as e:

raise ValueError(f"Invalid function input: {e}")

steps.append(f"Function f(x) = {func\_poly.to\_string()}")

steps.append(f"Analysis point x0 = {point}")

# For polynomial functions, continuity and differentiability are straightforward.

# Polynomials are continuous everywhere.

# Polynomials are differentiable everywhere.

steps.append("\n--- Continuity Analysis ---")

steps.append("1. Polynomial functions are defined for all real numbers.")

steps.append("2. The limit of a polynomial function as x approaches any point x0 is equal to f(x0).")

steps.append("3. Therefore, polynomial functions are continuous at all points in their domain.")

steps.append(f"Conclusion: f(x) = {func\_poly.to\_string()} is continuous at x = {point}.")

steps.append("\n--- Differentiability Analysis ---")

steps.append("1. The derivative of a polynomial function exists for all real numbers.")

steps.append(f"2. The derivative f'(x) = {deriv\_poly.to\_string()}.")

steps.append(f"3. Since f'(x) is also a polynomial, it is defined for all real numbers.")

steps.append("4. A function is differentiable at a point if its derivative exists at that point.")

steps.append(f"Conclusion: f(x) = {func\_poly.to\_string()} is differentiable at x = {point}.")

result\_str = (f"The function f(x) = {func\_poly.to\_string()} is both continuous and differentiable "

f"at x = {point}, as all polynomial functions are continuous and differentiable everywhere.")

return result\_str, steps

**report\_gen.py N.B: *(underscore between text)***

# report\_gen.py

# This module is responsible for formatting and generating comprehensive

# calculus analysis reports as text files.

import os

import datetime

from utils import ascii\_plotter # Import the ASCII plotter utility

class ReportGenerator:

"""

Generates comprehensive calculus reports based on analysis results.

Formats various mathematical operations and saves them to a text file.

"""

def \_\_init\_\_(self, output\_dir="reports"):

self.output\_dir = output\_dir

# Ensure the output directory exists, create if not

os.makedirs(self.output\_dir, exist\_ok=True)

def generate\_report(self, operation\_type, input\_function, result, steps=None, plot\_data=None, file\_name=None):

"""

Generates a text report summarizing a calculus analysis.

Args:

operation\_type (str): The type of calculus operation (e.g., "Symbolic Differentiation").

input\_function (str): The string representation of the input function(s).

result (str): The final result of the operation.

steps (list, optional): A list of strings detailing step-by-step explanations. Defaults to None.

plot\_data (dict, optional): Dictionary containing data for ASCII plotting.

e.g., {"original": [(x,y),...], "derived": [(x,y),...]}

or {"solution": [(x,y),...]}

Defaults to None.

file\_name (str, optional): The desired name for the report file.

If None, a timestamped name will be generated.

Returns:

str or None: The full path to the generated report file if successful,

None otherwise.

"""

# Format all analysis results into a single string

report\_content = self.\_format\_report(

operation\_type, input\_function, result, steps, plot\_data

)

# Determine the output file name

if file\_name is None:

timestamp\_str = datetime.datetime.now().strftime("%Y%m%d\_%H%M%S")

file\_name = f"{operation\_type.lower().replace(' ', '\_')}\_report\_{timestamp\_str}.txt"

file\_path = os.path.join(self.output\_dir, file\_name)

try:

# Write the formatted report content to the file

with open(file\_path, 'w', encoding='utf-8') as f:

f.write(report\_content)

# print(f"[INFO] Report successfully generated: {file\_path}") # Suppress this in report\_gen, main.py prints it

return file\_path

except IOError as e:

print(f"[ERROR] Could not write report to {file\_path}: {e}")

return None

except Exception as e:

print(f"[ERROR] An unexpected error occurred during report generation: {e}")

return None

def \_format\_report(self, operation\_type, input\_function, result, steps, plot\_data):

"""

Internal method to format the analysis results into a structured string.

Args:

operation\_type (str): The type of calculus operation.

input\_function (str): The input function(s) string.

result (str): The final result.

steps (list): Step-by-step explanations.

plot\_data (dict): Data for ASCII plotting.

Returns:

str: The formatted report content as a single string.

"""

report\_lines = []

report\_lines.append("=" \* 60)

report\_lines.append(f" MATHEMATICAL ANALYSIS REPORT - {operation\_type.upper()}")

report\_lines.append("=" \* 60)

report\_lines.append(f"Report Generated: {datetime.datetime.now().strftime('%Y-%m-%d %H:%M:%S')}\n")

# --- Operation Details ---

report\_lines.append(self.\_format\_section("Operation Details", [

f"Operation Type: {operation\_type}",

f"Input Function(s): {input\_function}"

]))

# --- Result ---

report\_lines.append(self.\_format\_section("Result", [result]))

# --- Step-by-Step Explanation ---

if steps:

report\_lines.append(self.\_format\_section("Step-by-Step Explanation", steps))

else:

report\_lines.append(self.\_format\_section("Step-by-Step Explanation", ["No detailed steps available for this operation."]))

# --- ASCII Plot ---

if plot\_data:

report\_lines.append("\n--- ASCII Plot ---")

report\_lines.append("Note: This is a basic ASCII representation and may not be perfectly accurate for all functions.")

all\_points = []

for key in plot\_data:

all\_points.extend(plot\_data[key])

if all\_points:

# Determine plot ranges dynamically

x\_values = [p[0] for p in all\_points]

y\_values = [p[1] for p in all\_points]

# Add some padding to ranges

x\_min = min(x\_values) - 1 if x\_values else -10

x\_max = max(x\_values) + 1 if x\_values else 10

y\_min = min(y\_values) - 1 if y\_values else -10

y\_max = max(y\_values) + 1 if y\_values else 10

# Ensure a minimum range to avoid division by zero if all points are identical

if x\_max - x\_min < 2: x\_max = x\_min + 2

if y\_max - y\_min < 2: y\_max = y\_min + 2

# Generate plot for each set of points

plot\_output\_lines = []

for label, points\_list in plot\_data.items():

plot\_output\_lines.append(f"Plot for {label.replace('\_', ' ').title()}:")

plot\_output\_lines.extend(ascii\_plotter(points\_list, x\_min, x\_max, y\_min, y\_max, width=60, height=20))

plot\_output\_lines.append("\n") # Separator between plots if multiple

report\_lines.extend(plot\_output\_lines)

else:

report\_lines.append("No plot data available.")

report\_lines.append("\n")

else:

report\_lines.append(self.\_format\_section("ASCII Plot", ["No plot generated for this operation."]))

report\_lines.append("=" \* 60)

report\_lines.append("END OF REPORT")

report\_lines.append("=" \* 60)

return "\n".join(report\_lines)

def \_format\_section(self, title, content\_lines):

"""Helper to format a section with a title and list of content lines."""

section = [f"\n--- {title} ---"]

section.extend([f" {line}" for line in content\_lines])

section.append("") # Add a blank line for spacing

return "\n".join(section)

**utils.py**

# utils.py

# This module contains various utility functions, including robust input validation,

# polynomial string parsing helpers, and a basic ASCII plotting utility.

import re

import math # For math.factorial

class InputValidator:

"""

Provides static methods for validating various types of user input.

"""

@staticmethod

def get\_valid\_int\_input(prompt, min\_val=None, max\_val=None):

"""

Prompts the user for an integer input and validates its format and optional range.

Args:

prompt (str): The message to display to the user.

min\_val (int, optional): The minimum allowed integer value.

max\_val (int, optional): The maximum allowed integer value.

Returns:

int: The validated integer input.

"""

while True:

try:

value = int(input(prompt).strip())

if min\_val is not None and value < min\_val:

print(f"[ERROR] Value must be at least {min\_val}.")

elif max\_val is not None and value > max\_val:

print(f"[ERROR] Value must be at most {max\_val}.")

else:

return value

except ValueError:

print("[ERROR] Invalid input. Please enter a whole number.")

except Exception as e:

print(f"[ERROR] An unexpected error occurred: {e}. Please try again.")

print("Please try again.")

@staticmethod

def get\_valid\_float\_input(prompt, min\_val=None, max\_val=None):

"""

Prompts the user for a floating-point number input and validates its format and optional range.

Args:

prompt (str): The message to display to the user.

min\_val (float, optional): The minimum allowed float value.

max\_val (float, optional): The maximum allowed float value.

Returns:

float: The validated float input.

"""

while True:

try:

value = float(input(prompt).strip())

if min\_val is not None and value < min\_val:

print(f"[ERROR] Value must be at least {min\_val}.")

elif max\_val is not None and value > max\_val:

print(f"[ERROR] Value must be at most {max\_val}.")

else:

return value

except ValueError:

print("[ERROR] Invalid input. Please enter a number.")

except Exception as e:

print(f"[ERROR] An unexpected error occurred: {e}. Please try again.")

print("Please try again.")

@staticmethod

def get\_yes\_no\_input(prompt):

"""

Prompts the user for a yes/no answer and returns a boolean.

Args:

prompt (str): The message to display to the user.

Returns:

bool: True for 'yes' or 'y', False for 'no' or 'n'.

"""

while True:

response = input(f"{prompt} (yes/no): ").strip().lower()

if response in ['yes', 'y']:

return True

elif response in ['no', 'n']:

return False

else:

print("[ERROR] Invalid input. Please enter 'yes' or 'no'.")

def parse\_polynomial\_string(func\_str):

"""

Parses a string representation of a polynomial into a dictionary of coefficients.

Supports terms like '3\*x^2', 'x^3', '2\*x', 'x', '5', '-x', '-4\*x^2'.

Args:

func\_str (str): The polynomial string.

Returns:

dict: A dictionary where keys are exponents (int) and values are coefficients (float).

Raises:

ValueError: If the string format is invalid.

"""

coefficients = {}

# Normalize the string: replace ' - ' with ' + -' to simplify splitting by '+'

# Also ensure 'x' without coefficient is treated as '1\*x'

# And 'x^n' without coefficient is treated as '1\*x^n'

normalized\_str = func\_str.replace("-", "+-").replace(" ", "") # Replace spaces for easier regex

# Handle cases like 'x' and '-x' by making coefficient explicit

normalized\_str = re.sub(r'(?<!\d|\\*)[x]', r'1\*x', normalized\_str)

normalized\_str = re.sub(r'-\\*x', r'-1\*x', normalized\_str) # Fix for -\*x becoming -1\*x

normalized\_str = re.sub(r'(?<!\d|\\*)[x]\^', r'1\*x^', normalized\_str) # Fix for x^n becoming 1\*x^n

normalized\_str = re.sub(r'-\\*x\^', r'-1\*x^', normalized\_str) # Fix for -\*x^n becoming -1\*x^n

# Split by '+' to get individual terms

terms = [term for term in normalized\_str.split("+") if term]

if not terms:

raise ValueError("Invalid polynomial format: No terms found.")

for term in terms:

if not term: continue # Skip empty strings from splitting

# Regex to capture coefficient, 'x' (optional), and exponent (optional)

# Group 1: coefficient (e.g., "3", "-2", "1")

# Group 2: 'x' (optional)

# Group 3: exponent (e.g., "2", "3")

match = re.match(r"([-+]?\d\*\.?\d\*)(?:(\\*x)(?:\^(\d+))?)?", term)

if not match:

# Handle constant terms like "5" or "-7"

if re.match(r"[-+]?\d\*\.?\d+$", term):

coeff = float(term)

coefficients[0] = coefficients.get(0, 0.0) + coeff

continue

else:

raise ValueError(f"Invalid term in polynomial: '{term}'")

coeff\_str = match.group(1)

has\_x = match.group(2)

exp\_str = match.group(3)

coeff = 1.0 if not coeff\_str else float(coeff\_str) # Default coeff to 1 if not specified (e.g., 'x^2')

if has\_x:

exponent = int(exp\_str) if exp\_str else 1 # Default exponent to 1 if 'x' but no '^n'

else:

exponent = 0 # Constant term (e.g., "5")

coefficients[exponent] = coefficients.get(exponent, 0.0) + coeff

return coefficients

def ascii\_plotter(points, x\_min, x\_max, y\_min, y\_max, width=60, height=20):

"""

Generates a basic ASCII plot for a set of (x, y) points.

Args:

points (list): A list of (x, y) tuples to plot.

x\_min (float): Minimum x-value for the plot range.

x\_max (float): Maximum x-value for the plot range.

y\_min (float): Minimum y-value for the plot range.

y\_max (float): Maximum y-value for the plot range.

width (int): Width of the plot grid (characters).

height (int): Height of the plot grid (characters).

Returns:

list: A list of strings, each representing a line of the ASCII plot.

"""

plot\_grid = [[' ' for \_ in range(width)] for \_ in range(height)]

# Draw Y-axis

for y in range(height):

plot\_grid[y][0] = '|'

# Draw X-axis

for x in range(width):

plot\_grid[height - 1][x] = '-'

# Origin

plot\_grid[height - 1][0] = '+'

# Scale factors

x\_scale = (width - 1) / (x\_max - x\_min) if (x\_max - x\_min) > 0 else 1

y\_scale = (height - 1) / (y\_max - y\_min) if (y\_max - y\_min) > 0 else 1

# Plot points

for x\_val, y\_val in points:

# Map x, y to grid coordinates

grid\_x = int((x\_val - x\_min) \* x\_scale)

grid\_y = int((y\_val - y\_min) \* y\_scale)

# Invert y-axis for console display (top-left is 0,0)

grid\_y = height - 1 - grid\_y

# Ensure coordinates are within bounds

if 0 <= grid\_x < width and 0 <= grid\_y < height:

plot\_grid[grid\_y][grid\_x] = '\*'

# Add labels (very basic)

plot\_lines = []

for i, row in enumerate(plot\_grid):

line = "".join(row)

if i == 0: # Top Y label

line += f" Y={y\_max:.1f}"

elif i == height - 1: # Bottom X label

line += f" X={x\_max:.1f}"

elif i == height // 2: # Middle Y label

line += f" Y={(y\_min + y\_max) / 2:.1f}"

plot\_lines.append(line)

# Add X-axis label at the bottom

plot\_lines.append(f"X={x\_min:.1f}" + " " \* (width - len(f"X={x\_min:.1f}") - len(f"X={x\_max:.1f}")) + f"X={x\_max:.1f}")

return plot\_lines

**7. Unit Tests**

This section contains the complete unit tests for the Mathematical Analysis Tool project, ensuring the reliability and correctness of its core functionalities.

**test\_symbolic\_calc.py N.B: *(underscore between text)***

# tests/test\_symbolic\_calc.py

# Unit tests for the symbolic\_calc module, covering Polynomial class,

# differentiation, integration, limit calculation, and series expansion.

import unittest

import math

import sys # For path manipulation if needed for imports

import os

# Add parent directory to path to allow importing modules from the main project

sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(\_\_file\_\_), '..')))

from symbolic\_calc import Polynomial, differentiate\_polynomial, integrate\_polynomial, \

calculate\_limit\_lhopital, taylor\_maclaurin\_series

class TestPolynomial(unittest.TestCase):

"""Tests the Polynomial class for parsing, string conversion, and evaluation."""

def test\_init\_valid\_polynomials(self):

# Test various valid polynomial inputs

p1 = Polynomial("3\*x^2 + 2\*x - 5")

self.assertEqual(p1.coefficients, {2: 3.0, 1: 2.0, 0: -5.0})

p2 = Polynomial("x^3 - x + 1")

self.assertEqual(p2.coefficients, {3: 1.0, 1: -1.0, 0: 1.0})

p3 = Polynomial("7")

self.assertEqual(p3.coefficients, {0: 7.0})

p4 = Polynomial("-x^2")

self.assertEqual(p4.coefficients, {2: -1.0})

p5 = Polynomial("x")

self.assertEqual(p5.coefficients, {1: 1.0})

p6 = Polynomial("0")

self.assertEqual(p6.coefficients, {0: 0.0})

p7 = Polynomial("2.5\*x^1 - 0.5\*x^0")

self.assertEqual(p7.coefficients, {1: 2.5, 0: -0.5})

p8 = Polynomial("-x^5 + 3\*x^2")

self.assertEqual(p8.coefficients, {5: -1.0, 2: 3.0})

p9 = Polynomial("4\*x^0")

self.assertEqual(p9.coefficients, {0: 4.0})

p10 = Polynomial("3\*x^2 + 0\*x - 5") # Test zero coefficient handling

self.assertEqual(p10.coefficients, {2: 3.0, 0: -5.0})

def test\_init\_invalid\_polynomials(self):

# Test invalid polynomial inputs

with self.assertRaises(ValueError):

Polynomial("3x^2") # Missing '\*'

with self.assertRaises(ValueError):

Polynomial("3\*x^a") # Non-integer exponent

with self.assertRaises(ValueError):

Polynomial("sin(x)") # Non-polynomial function

with self.assertRaises(ValueError):

Polynomial("") # Empty string

with self.assertRaises(ValueError):

Polynomial("3\*x^2 + invalid\_term")

def test\_to\_string(self):

# Test conversion back to string

self.assertEqual(Polynomial("3\*x^2 + 2\*x - 5").to\_string(), "3.0\*x^2 + 2.0\*x - 5.0")

self.assertEqual(Polynomial("x^3 - x + 1").to\_string(), "1.0\*x^3 - 1.0\*x + 1.0")

self.assertEqual(Polynomial("7").to\_string(), "7.0")

self.assertEqual(Polynomial("-x^2").to\_string(), "-1.0\*x^2")

self.assertEqual(Polynomial("x").to\_string(), "1.0\*x")

self.assertEqual(Polynomial("0").to\_string(), "0")

self.assertEqual(Polynomial("2.5\*x - 0.5").to\_string(), "2.5\*x - 0.5")

self.assertEqual(Polynomial("-x^5 + 3\*x^2").to\_string(), "-1.0\*x^5 + 3.0\*x^2")

self.assertEqual(Polynomial("4\*x^0").to\_string(), "4.0")

self.assertEqual(Polynomial("3\*x^2 + 0\*x - 5").to\_string(), "3.0\*x^2 - 5.0")

self.assertEqual(Polynomial("1\*x^2").to\_string(), "1.0\*x^2") # Test explicit 1 coeff

def test\_evaluate(self):

# Test polynomial evaluation

p1 = Polynomial("x^2 + 2\*x - 1")

self.assertEqual(p1.evaluate(0), -1.0)

self.assertEqual(p1.evaluate(1), 2.0)

self.assertEqual(p1.evaluate(-1), -2.0)

self.assertEqual(p1.evaluate(2), 7.0)

p2 = Polynomial("5")

self.assertEqual(p2.evaluate(100), 5.0)

p3 = Polynomial("x")

self.assertEqual(p3.evaluate(3), 3.0)

p4 = Polynomial("0")

self.assertEqual(p4.evaluate(5), 0.0)

class TestSymbolicCalculus(unittest.TestCase):

"""Tests differentiation, integration, limits, and series expansion functions."""

def test\_differentiate\_polynomial(self):

# Test differentiation

p1 = Polynomial("x^2 + 2\*x - 5")

d1 = differentiate\_polynomial(p1)

self.assertEqual(d1.to\_string(), "2.0\*x + 2.0")

p2 = Polynomial("3\*x^3")

d2 = differentiate\_polynomial(p2)

self.assertEqual(d2.to\_string(), "9.0\*x^2")

p3 = Polynomial("7")

d3 = differentiate\_polynomial(p3)

self.assertEqual(d3.to\_string(), "0")

p4 = Polynomial("x")

d4 = differentiate\_polynomial(p4)

self.assertEqual(d4.to\_string(), "1.0")

p5 = Polynomial("0")

d5 = differentiate\_polynomial(p5)

self.assertEqual(d5.to\_string(), "0")

p6 = Polynomial("-4\*x^2 + x - 10")

d6 = differentiate\_polynomial(p6)

self.assertEqual(d6.to\_string(), "-8.0\*x + 1.0")

def test\_integrate\_polynomial(self):

# Test integration

p1 = Polynomial("2\*x + 2")

i1 = integrate\_polynomial(p1)

self.assertEqual(i1.to\_string(), "1.0\*x^2 + 2.0\*x") # Note: +C is handled in CLI/report

p2 = Polynomial("9\*x^2")

i2 = integrate\_polynomial(p2)

self.assertEqual(i2.to\_string(), "3.0\*x^3")

p3 = Polynomial("0")

i3 = integrate\_polynomial(p3)

self.assertEqual(i3.to\_string(), "0")

p4 = Polynomial("1")

i4 = integrate\_polynomial(p4)

self.assertEqual(i4.to\_string(), "1.0\*x")

p5 = Polynomial("x^3 - x + 1")

i5 = integrate\_polynomial(p5)

self.assertEqual(i5.to\_string(), "0.25\*x^4 - 0.5\*x^2 + 1.0\*x")

def test\_calculate\_limit\_lhopital(self):

# Test L'Hôpital's Rule

# 0/0 form: (x^2 - 1) / (x - 1) as x->1 should be 2

limit, steps = calculate\_limit\_lhopital("x^2 - 1", "x - 1", 1)

self.assertAlmostEqual(limit, 2.0)

self.assertIn("Indeterminate form (0/0 or inf/inf) detected. Applying L'Hôpital's Rule.", steps[1])

self.assertIn("Derivative of Numerator: 2.0\*x", steps[4])

self.assertIn("Derivative of Denominator: 1.0", steps[5])

# 0/0 form: (x^3 - 8) / (x - 2) as x->2 should be 12

limit, steps = calculate\_limit\_lhopital("x^3 - 8", "x - 2", 2)

self.assertAlmostEqual(limit, 12.0)

# Non-indeterminate form: (x^2 + 1) / (x + 1) as x->1 should be 1

limit, steps = calculate\_limit\_lhopital("x^2 + 1", "x + 1", 1)

self.assertAlmostEqual(limit, 1.0)

self.assertIn("Not an indeterminate form, direct substitution yields a finite value.", steps[0])

# Non-indeterminate form: (x + 1) / (x - 1) as x->1 should be Infinity

limit, steps = calculate\_limit\_lhopital("x + 1", "x - 1", 1)

self.assertEqual(limit, "Infinity")

# Constant / 0: (5) / (x - 1) as x->1 should be Infinity

limit, steps = calculate\_limit\_lhopital("5", "x - 1", 1)

self.assertEqual(limit, "Infinity")

# 0 / non-zero: (x - 1) / (x + 1) as x->1 should be 0

limit, steps = calculate\_limit\_lhopital("x - 1", "x + 1", 1)

self.assertAlmostEqual(limit, 0.0)

# Inf/Inf form: (2\*x^2 + x) / (x^2 + 1) as x->inf should be 2

limit, steps = calculate\_limit\_lhopital("2\*x^2 + x", "x^2 + 1", float('inf'))

self.assertAlmostEqual(limit, 2.0)

self.assertIn("Indeterminate form (0/0 or inf/inf) detected. Applying L'Hôpital's Rule.", steps[1])

# Inf/Inf form: (x^3) / (x^2) as x->inf should be Infinity

limit, steps = calculate\_limit\_lhopital("x^3", "x^2", float('inf'))

self.assertEqual(limit, "Infinity")

# 0/0 form requiring multiple applications: (x^2 - 2x + 1) / (x^2 - 1) as x->1 should be 0

# This is (x-1)^2 / (x-1)(x+1) = (x-1)/(x+1) -> 0/2 = 0

limit, steps = calculate\_limit\_lhopital("x^2 - 2\*x + 1", "x^2 - 1", 1)

self.assertAlmostEqual(limit, 0.0)

self.assertGreater(len(steps), 5) # Should show multiple differentiation steps

# Test case where denominator derivative becomes zero and numerator is non-zero

limit, steps = calculate\_limit\_lhopital("x", "x^2", float('inf'))

self.assertEqual(limit, 0.0) # (1) / (2x) -> 0 as x->inf

# Test case where numerator derivative becomes zero and denominator is non-zero

limit, steps = calculate\_limit\_lhopital("x^2", "x^3", float('inf'))

self.assertEqual(limit, 0.0) # (2x) / (3x^2) -> (2) / (6x) -> 0 as x->inf

def test\_taylor\_maclaurin\_series(self):

# Test Maclaurin series (center = 0)

p1 = Polynomial("x^2")

series, steps = taylor\_maclaurin\_series(p1, 0, 2)

self.assertEqual(series, "0.0 + 0.0\*x + 1.0\*x^2") # Should simplify to "x^2" in a real app, but this is raw output

# Check for correct coefficients

self.assertIn("f^(0)(0) = 0.0", steps)

self.assertIn("f^(1)(0) = 0.0", steps)

self.assertIn("f^(2)(0) = 2.0", steps)

p2 = Polynomial("x^3 - 2\*x + 1")

series, steps = taylor\_maclaurin\_series(p2, 0, 3)

self.assertEqual(series, "1.0 - 2.0\*x + 0.0\*x^2 + 1.0\*x^3")

p3 = Polynomial("5")

series, steps = taylor\_maclaurin\_series(p3, 0, 5)

self.assertEqual(series, "5.0")

# Test Taylor series (center != 0)

p4 = Polynomial("x^2")

series, steps = taylor\_maclaurin\_series(p4, 1, 2) # (x-1)^2 + 2(x-1) + 1 = x^2 - 2x + 1 + 2x - 2 + 1 = x^2

self.assertEqual(series, "1.0 + 2.0\*(x - 1) + 1.0\*(x - 1)^2")

p5 = Polynomial("x^3")

series, steps = taylor\_maclaurin\_series(p5, 2, 3) # (x-2)^3 + 6(x-2)^2 + 12(x-2) + 8

self.assertEqual(series, "8.0 + 12.0\*(x - 2) + 6.0\*(x - 2)^2 + 1.0\*(x - 2)^3")

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

**test\_numerical\_solver.py N.B: *(underscore between text)***

# tests/test\_numerical\_solver.py

# Unit tests for the numerical\_solver module, covering Euler's method for ODEs.

import unittest

import sys # For path manipulation if needed for imports

import os

# Add parent directory to path to allow importing modules from the main project

sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(\_\_file\_\_), '..')))

from numerical\_solver import solve\_ode\_euler

class TestNumericalSolver(unittest.TestCase):

"""Tests the numerical ODE solver using Euler's method."""

def test\_solve\_ode\_euler\_simple\_constant(self):

# dy/dx = 2, y(0) = 0. Solution: y = 2x

# x=0, y=0 -> (0,0)

# x=0.5, y=1 -> (0.5,1)

# x=1.0, y=2 -> (1.0,2)

solution = solve\_ode\_euler("2", x0=0, y0=0, h=0.5, num\_steps=2)

expected\_solution = [(0.0, 0.0), (0.5, 1.0), (1.0, 2.0)]

self.assertEqual(len(solution), len(expected\_solution))

for i in range(len(solution)):

self.assertAlmostEqual(solution[i][0], expected\_solution[i][0])

self.assertAlmostEqual(solution[i][1], expected\_solution[i][1])

def test\_solve\_ode\_euler\_linear(self):

# dy/dx = x, y(0) = 0. Solution: y = 0.5\*x^2

# x=0, y=0 -> (0,0)

# x=0.1, y=0 + 0.1\*0 = 0 -> (0.1,0)

# x=0.2, y=0 + 0.1\*0.1 = 0.01 -> (0.2,0.01)

solution = solve\_ode\_euler("x", x0=0, y0=0, h=0.1, num\_steps=2)

expected\_solution = [(0.0, 0.0), (0.1, 0.0), (0.2, 0.01)]

self.assertEqual(len(solution), len(expected\_solution))

for i in range(len(solution)):

self.assertAlmostEqual(solution[i][0], expected\_solution[i][0], places=5)

self.assertAlmostEqual(solution[i][1], expected\_solution[i][1], places=5)

def test\_solve\_ode\_euler\_quadratic(self):

# dy/dx = x^2, y(0) = 0. Solution: y = x^3 / 3

# x=0, y=0 -> (0,0)

# x=1, y=0 + 1\*0^2 = 0 -> (1,0)

# x=2, y=0 + 1\*1^2 = 1 -> (2,1)

solution = solve\_ode\_euler("x^2", x0=0, y0=0, h=1, num\_steps=2)

expected\_solution = [(0.0, 0.0), (1.0, 0.0), (2.0, 1.0)]

self.assertEqual(len(solution), len(expected\_solution))

for i in range(len(solution)):

self.assertAlmostEqual(solution[i][0], expected\_solution[i][0], places=5)

self.assertAlmostEqual(solution[i][1], expected\_solution[i][1], places=5)

def test\_solve\_ode\_euler\_with\_initial\_y(self):

# dy/dx = 1, y(0) = 5. Solution: y = x + 5

solution = solve\_ode\_euler("1", x0=0, y0=5, h=1, num\_steps=1)

expected\_solution = [(0.0, 5.0), (1.0, 6.0)]

self.assertEqual(len(solution), len(expected\_solution))

for i in range(len(solution)):

self.assertAlmostEqual(solution[i][0], expected\_solution[i][0], places=5)

self.assertAlmostEqual(solution[i][1], expected\_solution[i][1], places=5)

def test\_solve\_ode\_euler\_negative\_h(self):

# dy/dx = 1, y(0) = 0. h = -0.5, num\_steps = 2. Should go backwards

solution = solve\_ode\_euler("1", x0=0, y0=0, h=-0.5, num\_steps=2)

expected\_solution = [(0.0, 0.0), (-0.5, -0.5), (-1.0, -1.0)]

self.assertEqual(len(solution), len(expected\_solution))

for i in range(len(solution)):

self.assertAlmostEqual(solution[i][0], expected\_solution[i][0], places=5)

self.assertAlmostEqual(solution[i][1], expected\_solution[i][1], places=5)

def test\_solve\_ode\_euler\_invalid\_function\_string(self):

# Test with a non-polynomial function string

with self.assertRaises(ValueError):

solve\_ode\_euler("sin(x)", x0=0, y0=0, h=0.1, num\_steps=1)

with self.assertRaises(ValueError):

solve\_ode\_euler("", x0=0, y0=0, h=0.1, num\_steps=1)

with self.assertRaises(ValueError):

solve\_ode\_euler("x^a", x0=0, y0=0, h=0.1, num\_steps=1)

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

**test\_function\_analysis.py N.B: *(underscore between text)***

# tests/test\_function\_analysis.py

# Unit tests for the function\_analysis module, covering continuity and differentiability checks.

import unittest

import sys # For path manipulation if needed for imports

import os

# Add parent directory to path to allow importing modules from the main project

sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(\_\_file\_\_), '..')))

from function\_analysis import analyze\_continuity\_differentiability

class TestFunctionAnalysis(unittest.TestCase):

"""Tests the function analysis capabilities."""

def test\_analyze\_continuity\_differentiability\_polynomial(self):

# Test a simple polynomial function

func\_str = "x^2 + 2\*x - 1"

point = 3.0

result\_str, steps = analyze\_continuity\_differentiability(func\_str, point)

self.assertIn(f"The function f(x) = 1.0\*x^2 + 2.0\*x - 1.0 is both continuous and differentiable at x = {point}", result\_str)

self.assertIn("Polynomial functions are continuous at all points in their domain.", steps)

self.assertIn("The derivative of a polynomial function exists for all real numbers.", steps)

self.assertIn("f'(x) = 2.0\*x + 2.0", steps) # Check for derivative string

def test\_analyze\_continuity\_differentiability\_constant(self):

# Test a constant function

func\_str = "5"

point = -10.0

result\_str, steps = analyze\_continuity\_differentiability(func\_str, point)

self.assertIn(f"The function f(x) = 5.0 is both continuous and differentiable at x = {point}", result\_str)

self.assertIn("Polynomial functions are continuous at all points in their domain.", steps)

self.assertIn("The derivative of a polynomial function exists for all real numbers.", steps)

self.assertIn("f'(x) = 0", steps) # Derivative of a constant is 0

def test\_analyze\_continuity\_differentiability\_linear(self):

# Test a linear function

func\_str = "-3\*x + 7"

point = 0.0

result\_str, steps = analyze\_continuity\_differentiability(func\_str, point)

self.assertIn(f"The function f(x) = -3.0\*x + 7.0 is both continuous and differentiable at x = {point}", result\_str)

self.assertIn("Polynomial functions are continuous at all points in their domain.", steps)

self.assertIn("The derivative of a polynomial function exists for all real numbers.", steps)

self.assertIn("f'(x) = -3.0", steps) # Derivative of -3x+7 is -3

def test\_analyze\_continuity\_differentiability\_invalid\_function\_string(self):

# Test with an invalid function string (non-polynomial)

with self.assertRaises(ValueError):

analyze\_continuity\_differentiability("sin(x)", 0.0)

with self.assertRaises(ValueError):

analyze\_continuity\_differentiability("", 0.0)

with self.assertRaises(ValueError):

analyze\_continuity\_differentiability("x^a", 1.0)

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

**test\_report\_gen.py*(\*underscore between text)***

# tests/test\_report\_gen.py

# Unit tests for the report\_gen module, verifying report formatting,

# file creation, and ASCII plotting integration.

import unittest

import os

import shutil

import sys # For path manipulation if needed for imports

from unittest import mock # Import the mock library

# Add parent directory to path to allow importing modules from the main project

sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(\_\_file\_\_), '..')))

from report\_gen import ReportGenerator

from utils import ascii\_plotter # To ensure the plotter is mocked correctly

class TestReportGenerator(unittest.TestCase):

"""Tests the ReportGenerator's ability to format and save calculus reports."""

def setUp(self):

# Set up a temporary output directory for reports

self.test\_output\_dir = "test\_reports\_temp"

self.report\_generator = ReportGenerator(output\_dir=self.test\_output\_dir)

os.makedirs(self.test\_output\_dir, exist\_ok=True)

# Sample data for testing report generation

self.operation\_type = "Symbolic Differentiation"

self.input\_function = "x^2 + 2\*x - 5"

self.result = "The derivative is: 2.0\*x + 2.0"

self.steps = [

"Input Function: x^2 + 2\*x - 5",

"Applying power rule (d/dx(ax^n) = anx^(n-1)) to each term.",

"Resulting Derivative: 2.0\*x + 2.0"

]

self.plot\_data = {

"original": [(-2, 0), (-1, -2), (0, -1), (1, 2), (2, 7)],

"derived": [(-2, -2), (-1, 0), (0, 2), (1, 4), (2, 6)]

}

def tearDown(self):

# Clean up the temporary output directory after tests

if os.path.exists(self.test\_output\_dir):

shutil.rmtree(self.test\_output\_dir)

def test\_generate\_report\_full\_data(self):

# Test report generation with a complete set of data including steps and plot

report\_path = self.report\_generator.generate\_report(

self.operation\_type, self.input\_function, self.result,

steps=self.steps, plot\_data=self.plot\_data, file\_name="diff\_report.txt"

)

self.assertIsNotNone(report\_path)

self.assertTrue(os.path.exists(report\_path))

with open(report\_path, 'r', encoding='utf-8') as f:

content = f.read()

self.assertIn("MATHEMATICAL ANALYSIS REPORT - SYMBOLIC DIFFERENTIATION", content)

self.assertIn(f"Operation Type: {self.operation\_type}", content)

self.assertIn(f"Input Function(s): {self.input\_function}", content)

self.assertIn(f"Result:\n {self.result}", content)

self.assertIn("--- Step-by-Step Explanation ---", content)

self.assertIn("Applying power rule", content)

self.assertIn("--- ASCII Plot ---", content)

self.assertIn("Plot for Original:", content)

self.assertIn("Plot for Derived:", content)

self.assertIn("END OF REPORT", content)

def test\_generate\_report\_no\_steps\_no\_plot(self):

# Test report generation without optional steps or plot data

report\_path = self.report\_generator.generate\_report(

"Symbolic Integration", "x^2", "Result: x^3/3 + C",

steps=None, plot\_data=None, file\_name="int\_report.txt"

)

self.assertIsNotNone(report\_path)

self.assertTrue(os.path.exists(report\_path))

with open(report\_path, 'r', encoding='utf-8') as f:

content = f.read()

self.assertIn("MATHEMATICAL ANALYSIS REPORT - SYMBOLIC INTEGRATION", content)

self.assertIn("No detailed steps available for this operation.", content)

self.assertIn("No plot generated for this operation.", content)

def test\_generate\_report\_default\_file\_name(self):

# Test that a report is generated with a default timestamped file name

report\_path = self.report\_generator.generate\_report(

"Calculate Limit", "x/x", "Result: 1.0"

)

self.assertIsNotNone(report\_path)

self.assertTrue(os.path.exists(report\_path))

self.assertTrue(os.path.basename(report\_path).startswith("calculate\_limit\_report\_"))

self.assertTrue(os.path.basename(report\_path).endswith(".txt"))

@mock.patch('builtins.open')

def test\_generate\_report\_io\_error(self, mock\_open):

# Test error handling when the report file cannot be written due to an IOError

mock\_open.side\_effect = IOError("Simulated permission denied error")

report\_path = self.report\_generator.generate\_report(

self.operation\_type, self.input\_function, self.result, file\_name="error\_report.txt"

)

self.assertIsNone(report\_path)

mock\_open.assert\_called\_once()

@mock.patch('utils.ascii\_plotter') # Mock the ascii\_plotter function

def test\_generate\_report\_with\_plot\_data\_calls\_plotter(self, mock\_ascii\_plotter):

# Configure mock\_ascii\_plotter to return dummy lines

mock\_ascii\_plotter.return\_value = ["dummy plot line 1", "dummy plot line 2"]

report\_path = self.report\_generator.generate\_report(

self.operation\_type, self.input\_function, self.result,

steps=self.steps, plot\_data=self.plot\_data, file\_name="plot\_test\_report.txt"

)

self.assertIsNotNone(report\_path)

self.assertTrue(os.path.exists(report\_path))

# Assert that ascii\_plotter was called for each set of points

self.assertEqual(mock\_ascii\_plotter.call\_count, 2) # Called for 'original' and 'derived'

with open(report\_path, 'r', encoding='utf-8') as f:

content = f.read()

self.assertIn("dummy plot line 1", content)

self.assertIn("dummy plot line 2", content)

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

**test\_utils.py**

# tests/test\_utils.py

# Unit tests for the utils module, covering InputValidator and polynomial parsing.

import unittest

import sys # For path manipulation if needed for imports

import os

from unittest import mock # For mocking input

# Add parent directory to path to allow importing modules from the main project

sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(\_\_file\_\_), '..')))

from utils import InputValidator, parse\_polynomial\_string, ascii\_plotter

class TestInputValidator(unittest.TestCase):

"""Tests the InputValidator static methods."""

@mock.patch('builtins.input', side\_effect=['5'])

def test\_get\_valid\_int\_input\_valid(self, mock\_input):

self.assertEqual(InputValidator.get\_valid\_int\_input("Enter int: "), 5)

@mock.patch('builtins.input', side\_effect=['abc', '10'])

def test\_get\_valid\_int\_input\_invalid\_then\_valid(self, mock\_input):

self.assertEqual(InputValidator.get\_valid\_int\_input("Enter int: "), 10)

@mock.patch('builtins.input', side\_effect=['3'])

def test\_get\_valid\_int\_input\_min\_max\_valid(self, mock\_input):

self.assertEqual(InputValidator.get\_valid\_int\_input("Enter int: ", 1, 5), 3)

@mock.patch('builtins.input', side\_effect=['0', '6', '4'])

def test\_get\_valid\_int\_input\_min\_max\_invalid\_then\_valid(self, mock\_input):

self.assertEqual(InputValidator.get\_valid\_int\_input("Enter int: ", 1, 5), 4)

@mock.patch('builtins.input', side\_effect=['3.14'])

def test\_get\_valid\_float\_input\_valid(self, mock\_input):

self.assertEqual(InputValidator.get\_valid\_float\_input("Enter float: "), 3.14)

@mock.patch('builtins.input', side\_effect=['xyz', '2.71'])

def test\_get\_valid\_float\_input\_invalid\_then\_valid(self, mock\_input):

self.assertEqual(InputValidator.get\_valid\_float\_input("Enter float: "), 2.71)

@mock.patch('builtins.input', side\_effect=['1.0'])

def test\_get\_valid\_float\_input\_min\_max\_valid(self, mock\_input):

self.assertEqual(InputValidator.get\_valid\_float\_input("Enter float: ", 0.5, 1.5), 1.0)

@mock.patch('builtins.input', side\_effect=['no'])

def test\_get\_yes\_no\_input\_no(self, mock\_input):

self.assertFalse(InputValidator.get\_yes\_no\_input("Continue?"))

@mock.patch('builtins.input', side\_effect=['YES'])

def test\_get\_yes\_no\_input\_yes(self, mock\_input):

self.assertTrue(InputValidator.get\_yes\_no\_input("Continue?"))

@mock.patch('builtins.input', side\_effect=['maybe', 'y'])

def test\_get\_yes\_no\_input\_invalid\_then\_valid(self, mock\_input):

self.assertTrue(InputValidator.get\_yes\_no\_input("Continue?"))

class TestPolynomialParser(unittest.TestCase):

"""Tests the parse\_polynomial\_string utility function."""

def test\_parse\_simple\_terms(self):

self.assertEqual(parse\_polynomial\_string("3\*x^2"), {2: 3.0})

self.assertEqual(parse\_polynomial\_string("x^3"), {3: 1.0})

self.assertEqual(parse\_polynomial\_string("2\*x"), {1: 2.0})

self.assertEqual(parse\_polynomial\_string("x"), {1: 1.0})

self.assertEqual(parse\_polynomial\_string("5"), {0: 5.0})

self.assertEqual(parse\_polynomial\_string("-x"), {1: -1.0})

self.assertEqual(parse\_polynomial\_string("-4\*x^2"), {2: -4.0})

self.assertEqual(parse\_polynomial\_string("0"), {0: 0.0})

self.assertEqual(parse\_polynomial\_string("1\*x^0"), {0: 1.0})

self.assertEqual(parse\_polynomial\_string("1\*x"), {1: 1.0})

def test\_parse\_multiple\_terms(self):

self.assertEqual(parse\_polynomial\_string("3\*x^2 + 2\*x - 5"), {2: 3.0, 1: 2.0, 0: -5.0})

self.assertEqual(parse\_polynomial\_string("x^3 - x + 1"), {3: 1.0, 1: -1.0, 0: 1.0})

self.assertEqual(parse\_polynomial\_string("-x^2 + 7\*x^5 - 10"), {2: -1.0, 5: 7.0, 0: -10.0})

self.assertEqual(parse\_polynomial\_string("2\*x - 3\*x^2 + 1"), {1: 2.0, 2: -3.0, 0: 1.0})

self.assertEqual(parse\_polynomial\_string("x^2 + x + 1"), {2: 1.0, 1: 1.0, 0: 1.0})

def test\_parse\_with\_spaces(self):

self.assertEqual(parse\_polynomial\_string(" 3 \* x ^ 2 + 2 \* x - 5 "), {2: 3.0, 1: 2.0, 0: -5.0})

def test\_parse\_float\_coefficients(self):

self.assertEqual(parse\_polynomial\_string("0.5\*x^2 - 1.5\*x + 0.25"), {2: 0.5, 1: -1.5, 0: 0.25})

def test\_parse\_zero\_coefficients(self):

# Terms with zero coefficients should be handled correctly (not necessarily removed by parser)

# The Polynomial class \_\_init\_\_ removes them.

coeffs = parse\_polynomial\_string("3\*x^2 + 0\*x - 5")

self.assertIn(2, coeffs)

self.assertIn(1, coeffs)

self.assertIn(0, coeffs)

self.assertEqual(coeffs[1], 0.0)

def test\_parse\_invalid\_formats(self):

with self.assertRaises(ValueError):

parse\_polynomial\_string("3x^2") # Missing '\*'

with self.assertRaises(ValueError):

parse\_polynomial\_string("x^a") # Non-integer exponent

with self.assertRaises(ValueError):

parse\_polynomial\_string("sin(x)") # Non-polynomial

with self.assertRaises(ValueError):

parse\_polynomial\_string("") # Empty string

with self.assertRaises(ValueError):

parse\_polynomial\_string("3\*x^2 + invalid")

with self.assertRaises(ValueError):

parse\_polynomial\_string("x\*\*2") # Invalid exponentiation operator

class TestASCIIPlotter(unittest.TestCase):

"""Tests the ascii\_plotter utility function."""

def test\_simple\_line(self):

points = [(0, 0), (1, 1), (2, 2)]

plot\_lines = ascii\_plotter(points, 0, 2, 0, 2, width=10, height=5)

self.assertIsInstance(plot\_lines, list)

self.assertGreater(len(plot\_lines), 0)

# Check if the '\*' characters are roughly on the diagonal

# This is a basic check, exact pixel mapping is hard for ASCII

self.assertIn("\*", plot\_lines[4]) # (0,0) -> bottom-left

self.assertIn("\*", plot\_lines[3]) # (1,1) -> middle

self.assertIn("\*", plot\_lines[2]) # (2,2) -> top-right

def test\_horizontal\_line(self):

points = [(0, 1), (1, 1), (2, 1)]

plot\_lines = ascii\_plotter(points, 0, 2, 0, 2, width=10, height=5)

self.assertIsInstance(plot\_lines, list)

# Check if '\*' are on the same y-level (row)

self.assertIn("\*", plot\_lines[2]) # Should be on the same row for y=1

def test\_empty\_points(self):

points = []

plot\_lines = ascii\_plotter(points, 0, 10, 0, 10, width=10, height=5)

self.assertIsInstance(plot\_lines, list)

# Should still draw axes

self.assertIn('|', plot\_lines[0])

self.assertIn('-', plot\_lines[4])

def test\_single\_point(self):

points = [(5, 5)]

plot\_lines = ascii\_plotter(points, 0, 10, 0, 10, width=10, height=5)

self.assertIsInstance(plot\_lines, list)

# Check if '\*' is present somewhere

found\_star = False

for line in plot\_lines:

if '\*' in line:

found\_star = True

break

self.assertTrue(found\_star)

def test\_ranges\_equal(self):

# Test case where x\_min == x\_max or y\_min == y\_max

points = [(0,0)]

plot\_lines = ascii\_plotter(points, 0, 0, 0, 0, width=10, height=10)

self.assertIsInstance(plot\_lines, list)

self.assertGreater(len(plot\_lines), 0)

# Should still produce a grid, even if scaled to a single point

found\_star = False

for line in plot\_lines:

if '\*' in line:

found\_star = True

break

self.assertTrue(found\_star)

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

## 8. Conclusion

The Mathematical Analysis Tool project stands as a testament to the successful development of a robust, modular, and highly functional Command Line Interface (CLI) application for fundamental calculus operations. By meticulously adhering to the stringent requirement of utilizing solely the Python standard library, this project has not only delivered a practical analytical tool but has also profoundly demonstrated a deep understanding of core mathematical algorithms and advanced software engineering principles.

A significant achievement of this endeavor lies in overcoming the inherent challenges posed by the "standard library only" constraint. Unlike projects that leverage powerful external libraries like SymPy for symbolic computation, this tool necessitated the ground-up implementation of parsing logic for polynomial expressions, as well as the fundamental rules for symbolic differentiation and integration. This bespoke development process demanded a precise and nuanced approach to handling mathematical operations, parsing complex input strings, and managing coefficient-exponent relationships, showcasing a thorough grasp of the underlying mathematical concepts and their algorithmic translation. Similarly, the numerical solution for differential equations, the iterative application of L'Hôpital's rule, and the generation of Taylor/Maclaurin series were crafted from first principles, ensuring complete control and transparency over each computational step.

The project's architecture, characterized by its clear class hierarchies and modular design across `symbolic\_calc.py`, `numerical\_solver.py`, `function\_analysis.py`, `report\_gen.py`, and `utipls.py`, exemplifies best practices in object-oriented programming. This modularity enhances readability, simplifies maintenance, and facilitates future expansion. Furthermore, the comprehensive error handling mechanisms, coupled with robust input validation, contribute significantly to the application's reliability and user-friendliness, ensuring graceful recovery from invalid inputs and unexpected scenarios. The inclusion of a dedicated suite of unit tests for critical functions rigorously validates the correctness and precision of the implemented algorithms, reinforcing the project's commitment to quality assurance.

Beyond its technical merits, the Mathematical Analysis Tool serves as a valuable educational resource. Its ability to generate detailed, step-by-step reports, complete with explanations and rudimentary ASCII graphical representations, demystifies complex calculus concepts. This transparency allows users to not only obtain results but also to understand the process behind them, fostering a deeper learning experience.

Looking forward, this project lays a solid foundation for numerous potential enhancements. Future iterations could explore expanding symbolic capabilities to include trigonometric or logarithmic functions (albeit with increased complexity given the standard library constraint), implementing more advanced numerical methods for ODEs (e.g., Runge-Kutta), or even venturing into multivariate calculus. The current framework is well-positioned to integrate such features, demonstrating its scalability and adaptability.

In conclusion, the Mathematical Analysis Tool represents a successful convergence of mathematical theory and practical software development. It stands as a testament to the power of fundamental programming principles in creating sophisticated solutions, even within constrained environments, and offers a robust, extensible platform for engaging with the fascinating world of calculus.