Design of GUI for scientific computing using Python

A dissertation submitted in the partial fulfillment of
the requirement for the degree of
Bachelor of Science
in
Mathematics

Submitted by:
Anshul Ghildiyal
(SAP ID: 500097811)

Under the supervision of
Dr. Ravi Kiran Maddali

Department of Mathematics, Applied Science Cluster
School of Engineering
UPES
Dehradun, Uttarakhand-248007, India

May, 2024

DECLARATION

I declare that the thesis entitled "**Design of GUI for scientific computing using Python"** has been prepared by me under the supervision of **Dr. Ravi Kiran Maddali** from **Department of Mathematics, School of Engineering, UPES, Dehradun, India.**

Anshul

Anshul Ghildiyal
Department of Mathematics
School of Engineering
UPES
Dehradun, Uttrakhand-248007, India

CERTIFICATE

I certify that, **Anshul Ghildiyal** has prepared his project entitled "**Design of GUI for scientific computing using Python"** for the award of **B.Sc. (Hons) Mathematics,** under my/our guidance. He has carried out the work at the **Department of Mathematics, School of Engineering, UPES, Dehradun, India.**

Dr. Ravi Kiran Maddali
Department of Mathematics
School of Engineering
UPES
Dehradun, Uttrakhand-248007, India

PLAGIARISM CERTIFICATE

I, **Anshul Ghildiyal**, herby certify that the research dissertation titled "**Design of GUI for scientific computing using Python**" submitted for the partial fulfillment of a B.Sc. degree from University of Petroleum Energy & Studies, Dehradun, India is an original idea and has not been copied/taken verbatim from anyone or from any other sources.

I further certify that this dissertation has been checked for plagiarism through a plagiarism detection tool " " approved by the university and "**8% (should be below 15%)**" percentage similarities have been found as shown in the attached report (given in the next page).

Anshul

Anshul Ghildiyal
Department of Mathematics
School of Engineering
UPES
Dehradun, Uttrakhand-248007, India

Approved by:

Dr. Ravi Kiran Maddali
Department of Mathematics
School of Engineering
UPES
Dehradun, Uttrakhand-248007, India

PLAGIARISM REPORT SUMMARY

BSc Disser

ORIGINALITY REPORT

8%	6%	4%	2%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	www.archive.org Internet Source	1%
2	www.rizvicollege.edu.in Internet Source	1%
3	fastercapital.com Internet Source	<1%
4	www.seab.gov.sg Internet Source	<1%
5	Rosario Russo, Alberto Clarich, Carlo Poloni, Enrico Nobile. "Optimization of a Boomerang shape using modeFRONTIER", 12th AIAA Aviation Technology, Integration, and Operations (ATIO) Conference and 14th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference, 2012 Publication	<1%
6	mathquerry.blogspot.com Internet Source	<1%
7	B. S. Grewal. "Chapter 8: Numerical Differentiation and Integration", Walter de	<1%

Abstract

Graphical User Interfaces (GUIs) help to bridge the gap between complex scientific computing methods and end users by offering an intuitive platform for interaction. In the world of scientific computing, where precision and efficiency are crucial, building GUIs that cater to the different needs of researchers and practitioners is critical. This dissertation provides a thorough examination of the design and implementation of GUIs for scientific computing using Python, a versatile and frequently used programming language.

The dissertation begins by looking at the theoretical foundations of GUI design principles, such as usability, user experience (UX), and human-computer interaction (HCI). Using known literature and procedures, a conceptual framework is created to guide the design process, guaranteeing that effective and user-centric interfaces designed specifically for mathematicians and scientists.

The dissertation relies heavily on Python, a strong and versatile programming language known for its ease of use and comprehension. Python's rich libraries, particularly Tkinter, PyQt, and wxPython, provide strong foundations for GUI development, allowing for smooth integration with scientific computing tools such as NumPy, SciPy, and Matplotlib. A comparison examination of these libraries reveals their strengths, shortcomings, and applicability for various application domains, allowing developers to make more educated judgments when choosing the best toolset for their project.

The dissertation also goes into the practical aspects of GUI implementation, covering major components such widgets, event handling, layout management, and data display. Using real-life examples and case studies, best practices for designing responsive and aesthetically beautiful interfaces is explained, highlighting the necessity of iterative design methods and user feedback in refining the user experience.

Furthermore, the dissertation delves into advanced themes in GUI creation, such as the incorporation of interactive components like sliders, buttons, and input fields, as well as ways for improving performance and scalability via asynchronous programming and multithreading. GUI apps can use Python's concurrent programming features to run computationally heavy operations in the background while remaining responsive and interactive.

The development of prototype programs aimed at various fields such as numerical analysis, data visualization, and computational modeling demonstrates the practical significance of GUI design for scientific computing. These applications are practical demonstrations of how GUIs may democratize access to complicated mathematical algorithms, empowering researchers, educators, and practitioners to explore complex phenomena with unprecedented ease and efficiency.

To sum up, this dissertation advances the rapidly developing subject of GUI design for scientific computing by fusing strategic implementation techniques with theoretical understanding. Through the utilization of Python's abundance of libraries and adaptability, programmers can create graphical user interfaces (GUIs) that surpass the conventional limits of scientific computing, promoting creativity and cooperation in several interdisciplinary fields. GUIs have the ability to completely transform the way mathematicians and scientists work with computer tools, advancing cutting-edge research and discovery through constant improvement and adaption to changing user needs.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

A critical frontier in the effort to make complicated computational tools both efficient and accessible is the nexus of scientific computing and user interface design. An increasing amount of complex computation and large-scale data processing are used in scientific research, therefore intuitive user interfaces are becoming essential. In this environment, Graphical User Interfaces (GUIs) are essential because they provide an interactive and visual way to interface with large datasets and intricate algorithms. In this dissertation, the Python programming language—which is well-known for its readability, efficiency, and rich ecosystem of scientific libraries—is used to design and construct a graphical user interface (GUI) that is specifically suited for scientific computing applications.

1. Scientific Computing and GUIs: A Vital Interface:

Scientific computing has become a vital tool for analysing and simulating complicated problems in a variety of scientific disciplines. Proficiency in complex algorithmic and large-scale data computation is essential for tasks ranging from climate modelling to theoretical physics. However, when paired with graphical user interfaces (GUIs), the potential of scientific computing is greatly expanded. With the help of graphical user interfaces (GUIs), complex computations may now be accessed by a wider range of users. This talk examines how GUIs improve usability, increase accessibility, and spur innovation as a crucial interface in scientific computing. The way that users interact with software systems has changed significantly as a result of the advent of GUIs. Graphical user interfaces (GUIs) are crucial instruments in scientific computing because they translate intricate computational operations into intuitive visual representations. This is especially helpful for users who need access to strong computational tools but may not have a lot of programming experience. Therefore, the design of a good graphical user interface (GUI) for scientific applications needs to balance ease of use with the depth of capability needed for scientific analysis.

1.1 Bridging the Gap Between Complexity and Usability:

1.1.1 Enhancing User Experience:

A GUI's fundamental function is to act as a conduit between the user and intricate computer operations. Conventional scientific computing jobs sometimes include using command-line interfaces (CLIs) to connect with software, which can be intimidating for non-programmers. GUIs provide a visual and easier-to-understand interface for interacting with software by abstracting away these intricacies. GUIs let users enter settings, manage execution, and view outcomes in real time with features like buttons, sliders, and graphical outputs. This improves the software's general usability and accessibility by enabling quick feedback loops and iterative procedures that are less laborious than command-line-based modifications.

1.1.2 Facilitating Advanced Visualization:

Understanding complicated data sets and simulation results requires the use of scientific visualization. Advanced visualization technologies are integrated into GUIs so that users can observe multidimensional data in an understandable fashion. For instance, in fluid dynamics, engineers can optimize designs greatly by using GUI-based tools that depict pressure contours and flow fields across a simulated aircraft surface.

Scientists and engineers may make more informed decisions more rapidly and gain a deeper understanding of their data by directly manipulating and interacting with these visualizations through a graphical user interface (GUI).

1.2 Democratizing Access to Scientific Tools:

1.2.1 Lowering the Learning Curve:

One of the most significant benefits of GUIs in scientific computing is their ability to democratize access. GUIs enable non-specialists to access strong scientific tools by reducing the entry barrier. This accessibility is essential in learning environments, where students studying difficult subjects can gain a great deal from interactive resources that offer instantaneous visual feedback and encourage a more thorough comprehension of the underlying ideas.

Furthermore, graphical user interfaces (GUIs) allow experts from different domains to contribute their skills in multidisciplinary research, even when they may not have specific technical background in computational methods. In more isolated environments, creative ideas might not be conceivable, but in this collaborative context, they can.

1.2.2 Supporting Scalability and Flexibility:

GUIs offer scalable and adaptable solutions in addition to being user-friendly. They frequently have choices to change the degree of control according to the user's skill level. An expert might access advanced options for fine-tuning parameters and algorithms, while a novice might use a GUI in a highly automated mode with preset setups and simplified controls.

This scalability guarantees that the same tool may be used for post-doctoral scholars undertaking cutting-edge research as well as for undergraduates teaching foundational ideas. Additionally, it implies that users can continue to use the same software and explore its deeper functions without having to switch tools as they become more proficient.

1.3 Accelerating Innovation Through Enhanced Collaboration:

1.3.1 Fostering Interdisciplinary Collaboration:

GUIs enable specialists from various domains to collaborate across disciplines by offering a shared platform. This is especially significant for intricate projects where input from other disciplines is required, such as environmental modeling or biomedical engineering. Teams with various scientific backgrounds can collaborate more easily if they have access to a graphical user interface (GUI) that can integrate data from biological sciences, geographical information systems (GIS), and climate models, for instance.

1.3.2 Streamlining Workflow Integration:

Many scientific initiatives include numerous stages: data collecting, processing, modeling, visualization, and analysis. GUIs can simplify this approach by offering tools for combining various stages into a single interface. For example, a GUI may enable a user to import data from multiple sources, process and analyze the data using drag-and-drop tools, and then directly see the results—all within the same software environment. This integration decreases error risk and saves time, resulting in faster iterations and innovations.

2. Python's Role in Scientific GUI Development:

Python, a versatile and strong programming language, has numerous applications in scientific computing. Python is a popular choice among scientists and academics because to its vast ecosystem of libraries and frameworks, which range from data analysis to machine learning and simulation. Graphical User Interfaces (GUIs) serve an important role in making scientific tools accessible and usable. In this post, we will look at Python's use in scientific GUI development, including its benefits, popular libraries, and prominent applications. Python is particularly suited to the development of GUIs in scientific computing due to its simplicity and the powerful capabilities of its libraries. Libraries such as NumPy, SciPy, and Matplotlib offer extensive functionalities for numerical computation, scientific processing, and data visualization, respectively. These tools, when integrated within a GUI framework, allow users to perform robust scientific analyses through simpler, more intuitive interfaces. Python also supports several GUI frameworks, among which PyQt is selected for this project due to its comprehensive features and compatibility with the Qt framework, known for creating scalable and robust applications.

2.1 Introduction to GUI Development with Python:

Graphical User Interfaces (GUIs) allow users to interact with software applications in an easy and intuitive manner. Python provides a number of tools and frameworks for constructing GUIs, each with its own set of strengths and use cases. Tkinter and PyQt are two of the most widely used GUI libraries in the Python ecosystem.

2.2 Tkinter: Python's Built-in GUI Library:

Tkinter, Python's primary GUI toolkit, is included with most Python installs. It provides a simple and user-friendly interface for designing GUI applications. Despite its simplicity, Tkinter is powerful enough to generate elaborate GUIs for scientific applications.

One of the primary benefits of Tkinter is its ease of use. Python developers may rapidly get started with Tkinter thanks to its simple API and abundant documentation. Furthermore, Tkinter's cross-platform nature assures that GUI programs created with Tkinter work perfectly across multiple operating systems without change.

While Tkinter does not have as many advanced features as other GUI libraries, its lightweight design and ease of interface with other Python libraries make it a popular choice for prototyping and small to medium-sized scientific applications.

2.3 PyQt: Python Bindings for Qt:

PyQt is another popular GUI library for Python, providing bindings for the Qt framework. Qt is a powerful and feature-rich C++ framework for developing cross-platform applications with GUIs. PyQt enables Python developers to leverage the capabilities of Qt for building sophisticated and professional-looking GUIs.

One of the main advantages of PyQt is its extensive set of widgets and tools for creating modern and visually appealing user interfaces. PyQt also offers excellent support for multimedia, graphics rendering, and 2D/3D plotting, making it well-suited for scientific applications requiring advanced visualization capabilities.

However, PyQt has a steeper learning curve compared to Tkinter, primarily due to the complexity of the Qt framework and its API. Despite this, many developers prefer PyQt for its flexibility, performance, and extensive documentation.

2.4 Matplotlib and Plotly: Data Visualization in GUIs:

Data visualization is a critical component of scientific computing, enabling researchers to effectively examine and interpret large datasets. Matplotlib and Plotly are two well-known Python libraries for creating interactive and publication-quality plots and charts that can be easily integrated into GUI applications.

Matplotlib has a variety of charting tools for creating static, animated, and interactive displays. Its connection with Tkinter and PyQt enables developers to embed Matplotlib charts directly into GUI windows, allowing users to interact with them via GUI controls.

Plotly, on the other hand, allows for interactive charting and supports web-based GUIs. Plotly's interactive charts may be incorporated in PyQt programs via the PyQtGraph library or viewed in web browsers, making them ideal for creating web-based scientific tools and dashboards.

2.5 Advanced GUI Development with PySide and Kivy:

In addition to Tkinter and PyQt, Python provides various GUI frameworks such as PySide and Kivy, each with its own set of features and strengths.

PySide is an alternate set of Python bindings for the Qt framework that provides similar functionality to PyQt. PySide, on the other hand, is created by the Qt corporation and distributed under the LGPL license, making it more appropriate for commercial applications.

Kivy, on the other hand, is an open-source Python framework for building multi-touch apps with a natural user interface (NUI). Kivy's cross-platform capabilities and support for touch input make it perfect for creating GUI apps for mobile devices and tablets in scientific research and teaching.

2.6 Case Studies: Applications of Python GUIs in Scientific Computing:

To illustrate Python's role in scientific GUI development, let's look at some real-world case studies where Python GUIs have been used effectively in scientific computing:

- **AstroImageJ:** AstroImageJ is an open-source image processing and analysis software for astronomy. It provides a user-friendly GUI built using Java Swing, with extensive integration with Python scripting for advanced customization and automation.
- **PyMOL:** PyMOL is a molecular visualization system used by scientists for molecular modeling and structural biology. It offers a sophisticated GUI built using Tkinter, combined with powerful scripting capabilities in Python for extending its functionality.
- **OpenCV GUI:** OpenCV, a popular computer vision library, offers GUI tools for interactive image processing and computer vision tasks. These GUI applications are typically built using Qt bindings for Python, enabling developers to create custom interfaces for analyzing and manipulating images and videos.

3. Research Objective:

The primary objectives outlined for this dissertation are as follows:

1. User Requirements Analysis: To conduct a comprehensive analysis of the needs and requirements of users engaged in scientific computing, to determine the critical features and functionalities that the GUI must support.
2. GUI Design: To develop a design prototype that integrates user feedback and adheres to best practices in GUI design, ensuring both usability and functionality.
3. Implementation Using Python and PyQt: To construct the GUI using Python and PyQt, focusing on modular design and effective integration of Python's scientific libraries.
4. Evaluation and Refinement: To assess the effectiveness of the GUI through user testing and case studies, and to refine the design based on feedback to meet user expectations effectively.

4. Methodological Framework:

The methodological framework adopted in this dissertation is illustrated in *Figure 1*, which outlines the sequential phases of the research from conception to evaluation.

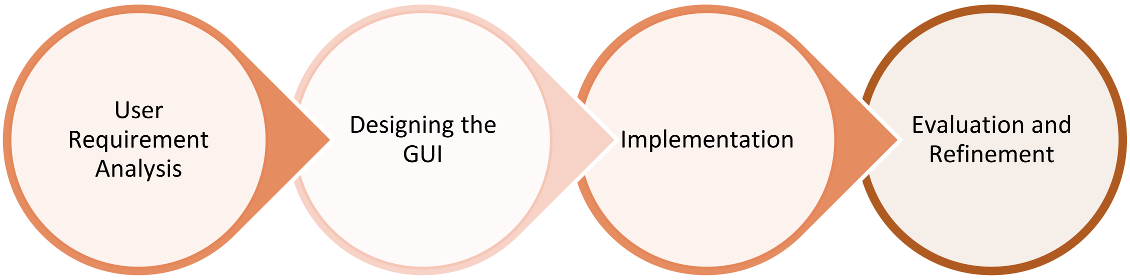


Fig. 1: *A flowchart depicting the stages of the dissertation from user requirements analysis, design, implementation, to evaluation and refinement.*

5. Technological Stack:

The selection of tools and technologies employed in the development of the GUI is critical. *Table 1* summarizes these choices and their justifications.

Tool/Framework	**Purpose**	**Reason for Selection**
Python	Programming Language	Extensive libraries, ease of use
PyQt	GUI Development Framework	Comprehensive features, cross-platform
Numpy	Numerical Operations	Optimal performance in large datasets
Scipy	Advanced Scientific Computations	Wide range of scientific algorithms
Plotly	Data Visualization	Versatile plotting capabilities
Matplotlib	Data Visualization	Versatile plotting capabilities
qdarkstlye	Customizing	Introducing Dark theme
Math	Basic operations	Can easily do basic operations
Sympy	Symbolize	Write the code in user readable format

Table 1: *Tools and Technologies*

6. GUI Design and User-Centered Development:

The design phase focuses on a user-centered approach, involving iterative prototyping and testing. This iterative cycle ensures that the GUI not only meets the functional requirements of scientific computing but also addresses the usability needs of its end-users.

7. Topics covered:

S.no.	**Topics**	**Description**
1.	Basic arithmetic operations	Addition, subtract, multiply, division
2.	Trigonometric functions	Sine, cosine, tangent, sec, cosec, cotan
3.	Exponential and logarithmic functions	Exponentiation, ln, log, and their inverses
4.	Numerical Integration and Differentiation	Trapezoidal rule, Simpson's $1/3^{rd}$, ….
5.	Numerical Solutions of Differential Equations	ODE, PDE
6.	Numerical Methods	Solving Transcendental equations

Table 2: *All the topics that Scientific Calculation deals with*

Now let's look into the topics covered by Scientific Calculator in details,

References:

[1] Qingkai Kong, Timmy Siauw, Alexandre M. Bayen, *Python Programming and Numerical Methods*

[2] Svein Linge, Hans Petter Langtangen, *Programming for Computations- A Gentle Introduction to Numerical Simulations with Python*

CHAPTER 2
MATHEMATICAL FUNCTIONS AND OPERATIONS
(with their Python code)

1. Basic Arithmetic Operations:

Basic arithmetic operations are the foundation of mathematical calculation, affecting not only mathematics but also a wide range of real-world applications. This study seeks to provide a thorough overview of addition, subtraction, multiplication, and division, which are the fundamental operations required for any mathematical effort. We will look at their definitions, qualities, applications, and significance in both theoretical and practical settings.

Basic arithmetic operations serve as the foundation for mathematical reasoning and problem solving. Their comprehension is critical for both theoretical mathematical research and practical applications across a wide range of disciplines. By thoroughly investigating addition, subtraction, multiplication, and division, we gain understanding into their properties, applications, and significance, allowing us to approach mathematical problems with confidence and precision.

Now designing a GUI of a calculator which can do basic arithmetic (using PyQt package):

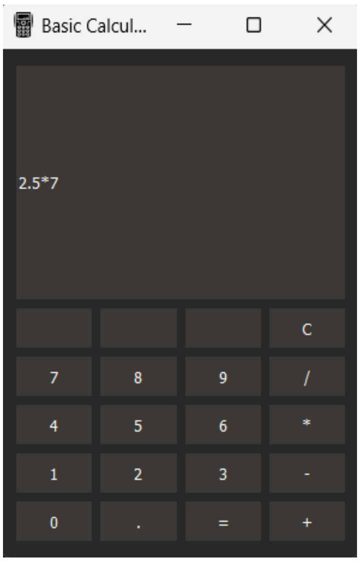
```python
class BasicCalculator(QWidget):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("Basic Calculator")
        self.setWindowIcon(QIcon(r"C:\Users\HP\Desktop\scientific-calculator.png"))
        self.setGeometry(100, 100, 300, 400)

        # Create the main layout
        main_layout = QVBoxLayout()

        # Create the display widget
        self.result_display = QLineEdit()
        self.result_display.setReadOnly(True)
        self.result_display.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Preferred)
        main_layout.addWidget(self.result_display)

        # Create the button grid layout
        button_grid_layout = QVBoxLayout()

        #Row 0
        row0 = self.create_button_row(['', '', ' ', 'C'])
        button_grid_layout.addLayout(row0)

        # Row 1
        row1 = self.create_button_row(['7', '8', '9', '/'])
        button_grid_layout.addLayout(row1)

        # Row 2
        row2 = self.create_button_row(['4', '5', '6', '*'])
        button_grid_layout.addLayout(row2)

        # Row 3
        row3 = self.create_button_row(['1', '2', '3', '-'])
        button_grid_layout.addLayout(row3)

        # Row 4
        row4 = self.create_button_row(['0', '.', '=', '+'])
        button_grid_layout.addLayout(row4)
```

```python
        # Add the button grid layout to the main layout
        main_layout.addLayout(button_grid_layout)

        # Set the main layout for the widget
        self.setLayout(main_layout)

    def create_button_row(self, button_texts):
        row_layout = QHBoxLayout()

        for button_text in button_texts:
            button = QPushButton(button_text)
            button.clicked.connect(self.on_button_click)
            button.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Preferred)
            row_layout.addWidget(button)

        return row_layout

    def on_button_click(self):
        button = self.sender()
        current_text = self.result_display.text()

        if button.text() == "=":
            try:
                result = str(eval(current_text))
                self.result_display.setText(result)
            except Exception as e:
                self.result_display.setText("Error")
        elif button.text() == 'C':
            self.result_display.clear()
        else:
            self.result_display.setText(current_text + button.text())


    def on_clear_click(self):
        self.result_display.clear()
```

Fig 2: *GUI of calculator which can do basic arithmetic operations*

Result-

Input **Output**



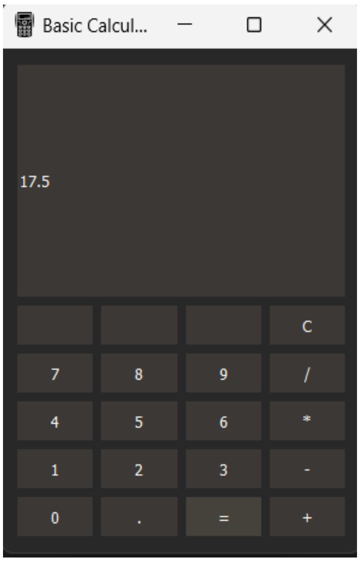


Fig 3: *Basic Arithmetic Operation's GUI Result*

2. Trigonometric Equations:

Trigonometric functions are basic mathematical tools that explain the angles and sides of triangles. They have several applications in domains like physics, engineering, and astronomy. This study examines the primary trigonometric functions: sine, cosine, tangent, cosecant, secant, and cotangent. We will talk about their definitions, attributes, graphical representations, and practical applications.

Now designing a GUI of a calculator which verify trigonometric values (using PyQt package):

```python
class Calculator(QWidget):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("Log, exp & trig Calculator")
        self.setGeometry(100,100,400,400)

        self.expression_lineedit = QLineEdit(self)
        self.expression_lineedit.setPlaceholderText("Enter expression")

        self.result_lineedit = QLineEdit(self)
        self.result_lineedit.setReadOnly(True)

        self.create_buttons()
        self.create_layout()

    def create_buttons(self):
        buttons_layout = QGridLayout()

        buttons = [
            ('7', 0, 0),
            ('8', 0, 1),
            ('9', 0, 2),
            ('4', 1, 0),
            ('5', 1, 1),
            ('6', 1, 2),
            ('1', 2, 0),
            ('2', 2, 1),
            ('3', 2, 2),
            ('0', 3, 0),
            ('.', 3, 2),
            ('+', 0, 3),
            ('-', 1, 3),
            ('*', 2, 3),
            ('/', 3, 3),
            ('^', 4, 0),
            ('ln', 4, 1),
            ('exp', 4, 2),
            ('sin', 5, 0),
            ('cos', 5, 1),
            ('tan', 5, 2),
            ('(', 3, 1),
            (')', 3, 2),
            ('=', 6, 3),
            ('⌫', 4, 3),
            ('AC', 5, 3),
        ]

        for btn_text, row, col in buttons:
            button = QPushButton(btn_text, self)
            button.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Expanding)
            button.clicked.connect(lambda _, text=btn_text: self.button_pressed(text))
            buttons_layout.addWidget(button, row, col)

        self.buttons_layout = buttons_layout

    def create_layout(self):
        main layout = QVBoxLayout()
```

```python
class Calculator(QWidget):

    def create_layout(self):
        main_layout = QVBoxLayout()

        main_layout.addWidget(self.expression_lineedit)
        main_layout.addWidget(self.result_lineedit)
        main_layout.addLayout(self.buttons_layout)

        self.setLayout(main_layout)

        # Set dark theme
        self.setStyleSheet("""
            QWidget {
                background-color: #282828;
                color: #f8f8f2;
            }
            QLineEdit {
                background-color: #3c3836;
                color: #f8f8f2;
                border: none;
            }
            QPushButton {
                background-color: #3c3836;
                color: #f8f8f2;
                border: none;
            }
            QPushButton:hover {
                background-color: #45413b;
            }
        """)

    def button_pressed(self, text):
        current_text = self.expression_lineedit.text()

        if text == '=':
            try:
                result = eval(current_text)
                self.result_lineedit.setText(str(result))
            except Exception as e:
                self.result_lineedit.setText("Error")
        elif text == '⌫':
            new_text = current_text[:-1]
            self.expression_lineedit.setText(new_text)
        elif text == 'AC':
            self.expression_lineedit.clear()
            self.result_lineedit.clear()
        else:
            new_text = current_text + text
            self.expression_lineedit.setText(new_text)
```

Fig 4: *GUI of calculator which calculates trigonometric function*

Result –

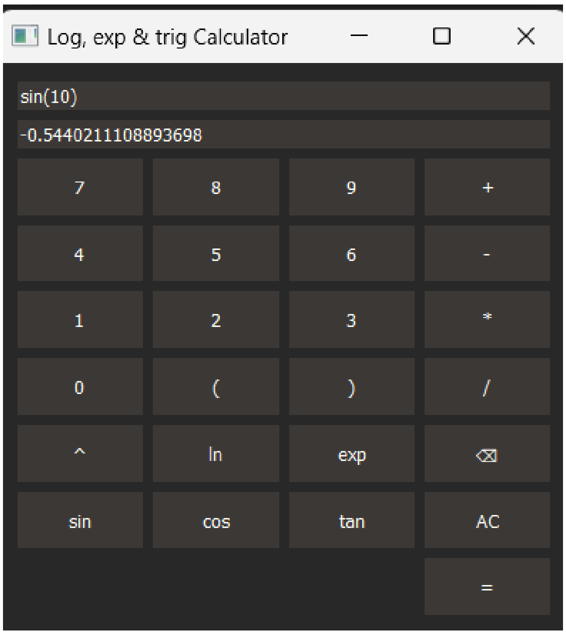


Fig 5: *Trigonometric Function's GUI Result*

3. Exponential and Logarithmic Functions:

Exponential and logarithmic functions are fundamental mathematical concepts with numerous applications in disciplines such as mathematics, physics, engineering, economics, and biology. This report intends to provide a thorough examination of exponential and logarithmic functions, including definitions, properties, graphical representations, and practical applications.

3.1. Exponential Function:

An exponential function is a mathematical function that takes the form f(x) = $a.b^x$, where a and b are constants and b is the exponential function's base. The variable x represents the exponent, which can be any real number. Exponential functions exhibit rapid growth or decay.

3.2. Logarithmic Functions

The logarithm function is the inverse of the exponential function. The definition is as follows: f(x) = log_b(x), where b is the base of the logarithm and x is its argument. Logarithmic functions help you solve exponential equations and represent the rate of growth or decay

3.3.Relationship Between Exponential and Logarithmic Functions:

Exponential and logarithmic functions are closely related through their inverse relationship. Specifically, if $y = b^x$, then x = $log_b y$. This relationship forms the basis for solving equations involving exponential and logarithmic functions.

Exponential and logarithmic functions are fundamental mathematical concepts that have numerous applications in mathematics and related fields. Their properties, graphical representations, and practical significance make them essential tools for modeling, analyzing, and solving problems in a variety of fields, including finance and economics, physics and biology. Mastery of exponential and logarithmic functions allows people to gain insight into complex phenomena, make informed decisions, and contribute to scientific and technological advancements.

Now designing a GUI of a calculator which verify logarithm and exponential values (using PyQt package):

```python
class Calculator(QWidget):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("Log, exp & trig Calculator")
        self.setGeometry(100,100,400,400)

        self.expression_lineedit = QLineEdit(self)
        self.expression_lineedit.setPlaceholderText("Enter expression")

        self.result_lineedit = QLineEdit(self)
        self.result_lineedit.setReadOnly(True)

        self.create_buttons()
        self.create_layout()

    def create_buttons(self):
        buttons_layout = QGridLayout()

        buttons = [
            ('7', 0, 0),
             ('8', 0, 1),
             ('9', 0, 2),
             ('4', 1, 0),
             ('5', 1, 1),
             ('6', 1, 2),
             ('1', 2, 0),
             ('2', 2, 1),
             ('3', 2, 2),
             ('0', 3, 0),
             ('.', 3, 2),
             ('+', 0, 3),
             ('-', 1, 3),
             ('*', 2, 3),
             ('/', 3, 3),
             ('^', 4, 0),
             ('ln', 4, 1),
             ('exp', 4, 2),
             ('sin', 5, 0),
             ('cos', 5, 1),
             ('tan', 5, 2),
             ('(', 3, 1),
             (')', 3, 2),
             ('=', 6, 3),
             ('⌫', 4, 3),
             ('AC', 5, 3),
        ]

        for btn_text, row, col in buttons:
            button = QPushButton(btn_text, self)
            button.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Expanding)
            button.clicked.connect(lambda _, text=btn_text: self.button_pressed(text))
            buttons_layout.addWidget(button, row, col)

        self.buttons_layout = buttons_layout

    def create_layout(self):
        main layout = QVBoxLayout()
```

```python
class Calculator(QWidget):

    def create_layout(self):
        main_layout = QVBoxLayout()

        main_layout.addWidget(self.expression_lineedit)
        main_layout.addWidget(self.result_lineedit)
        main_layout.addLayout(self.buttons_layout)

        self.setLayout(main_layout)

        # Set dark theme
        self.setStyleSheet("""
            QWidget {
                background-color: #282828;
                color: #f8f8f2;
            }
            QLineEdit {
                background-color: #3c3836;
                color: #f8f8f2;
                border: none;
            }
            QPushButton {
                background-color: #3c3836;
                color: #f8f8f2;
                border: none;
            }
            QPushButton:hover {
                background-color: #45413b;
            }
        """)

    def button_pressed(self, text):
        current_text = self.expression_lineedit.text()

        if text == '=':
            try:
                result = eval(current_text)
                self.result_lineedit.setText(str(result))
            except Exception as e:
                self.result_lineedit.setText("Error")
        elif text == '⌫':
            new_text = current_text[:-1]
            self.expression_lineedit.setText(new_text)
        elif text == 'AC':
            self.expression_lineedit.clear()
            self.result_lineedit.clear()
        else:
            new_text = current_text + text
            self.expression_lineedit.setText(new_text)
```

Fig 6: *GUI of calculator which calculate exponential and logarithmic functions*

Result-

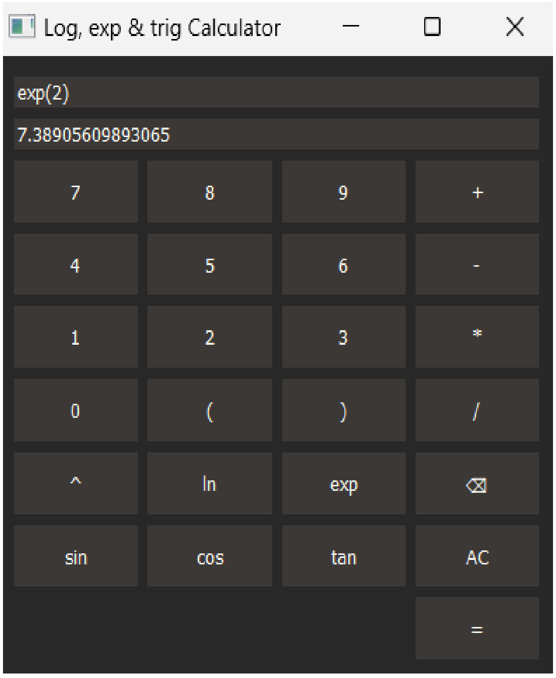


Fig 7: *Exponential and Logarithmic Function's Result*

4. Numerical Integration:

4.1. Introduction

Numerical integration, also known as numerical quadrature, is a fundamental branch of numerical analysis used to approximate definite integrals, particularly when analytical solutions are unfeasible. This technique is used in a variety of scientific and technical domains for modeling and simulation.

4.2. Methods of Numerical Integration

Newton-Cotes formulae and Gaussian quadrature are two types of numerical integration techniques, with each having a different purpose depending on function behavior and required precision.

4.3. Newton-Cotes Formulas:

These approaches use polynomials to interpolate the function f(x) at evenly or unequally spaced points in the interval.
Let interval [a, b] be divide into n equal subintervals such that
$a = x_0 < x_1 < x_2 \dots < x_n = b$. Clearly, $x_n = x_0 + nh$. Then, we have

$$\int_{x_0}^{x_n} y\,dx = nh\left[y_0 + \frac{n}{2}\Delta y_0 + \frac{n(2n-3)}{12}\Delta^2 y_0 + \frac{n(n-2)^2}{24}\Delta^3 y_{0+\dots}\dots\right] \qquad (1)$$

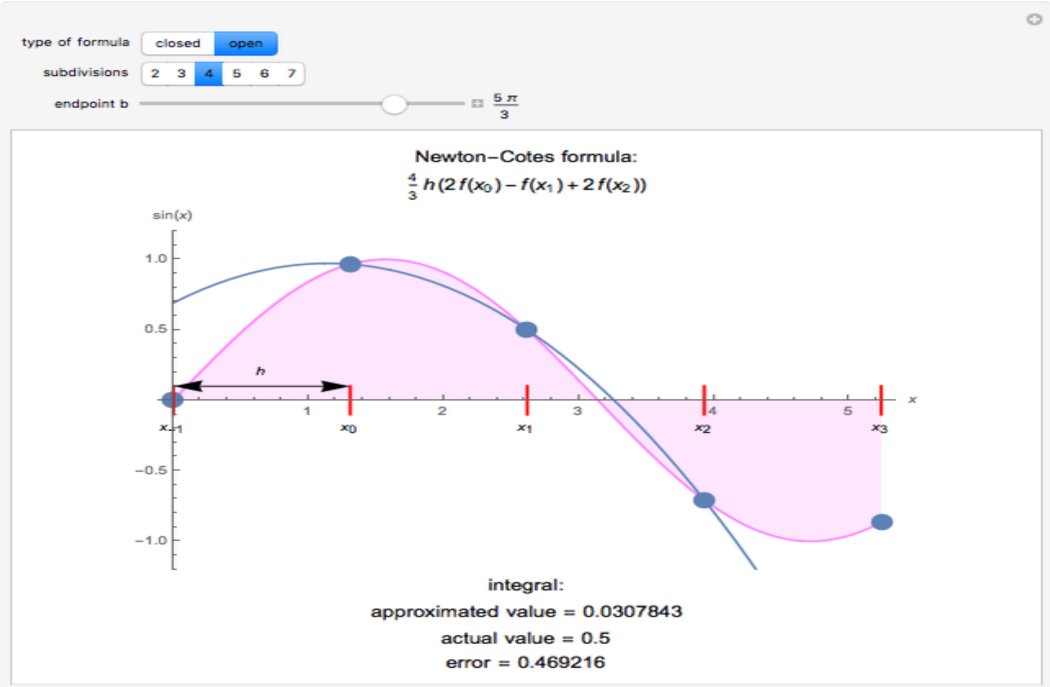


***Fig 8**: Shows the graphical representation of Newton – Cotes Quadrature*

The Newton-Cotes quadrature formula is the name given to this. We can use this simple formula to find the different integration procedures according to the value of n.

- **Trapezoidal Rule:**

All differences greater than the first will become zero when n = 1 is used in the Newton-Cotes quadrature general formula above, giving us

$$\int_{x_0}^{x_n} y\,dx = \frac{h}{2}[y_0 + 2(y_1 + y_2 + \cdots + y_{n\cdot 1}) + y_n]$$

It is referred to as the *trapezoidal rule*

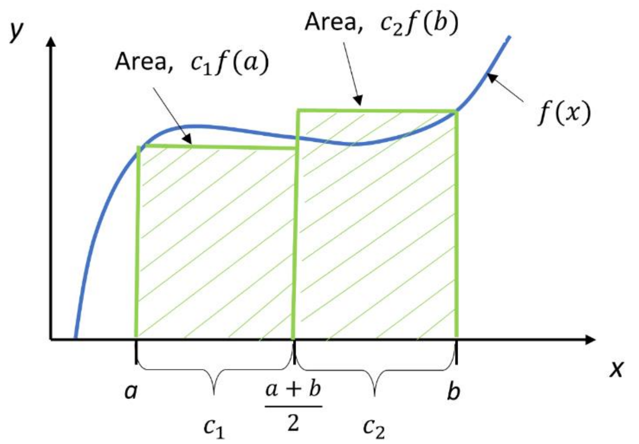
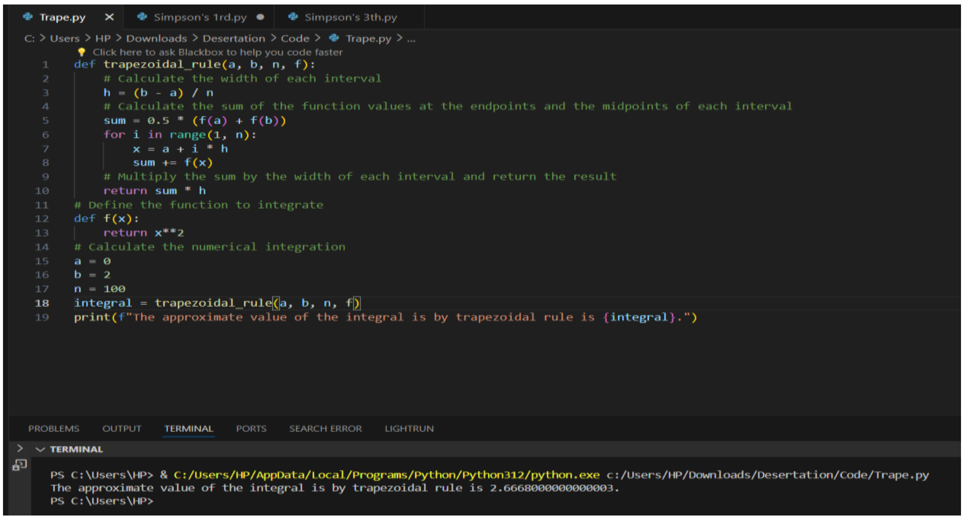


***Fig 9:** Graph representing Trapezoidal Rule*

The area then roughly equals the total of the areas of the n trapeziums created, circumscribed by the curve y = f(x), the ordinates x = x_0 and x = x_n, and the axis.

Code:




```python
def trapezoidal_rule(a, b, n, f):
    # Calculate the width of each interval
    h = (b - a) / n
    # Calculate the sum of the function values at the endpoints and the midpoints of each interval
    sum = 0.5 * (f(a) + f(b))
    for i in range(1, n):
        x = a + i * h
        sum += f(x)
    # Multiply the sum by the width of each interval and return the result
    return sum * h
# Define the function to integrate
def f(x):
    return x**2
# Calculate the numerical integration
a = 0
b = 2
n = 100
integral = trapezoidal_rule(a, b, n, f)
print(f"The approximate value of the integral is by trapezoidal rule is {integral}.")
```

```
PS C:\Users\HP> & C:/Users/HP/AppData/Local/Programs/Python/Python312/python.exe c:/Users/HP/Downloads/Desertation/Code/Trape.py
The approximate value of the integral is by trapezoidal rule is 2.6668000000000003.
PS C:\Users\HP>
```

***Fig 10**: Basic code doing numerical integration (using trapezoidal rule)*

- **Simpson's 1/3rd Rule:**

In the Newton-Cotes quadrature general formula above, let n = 2, and substitute $\frac{n}{2}$ arcs of second-degree polynomials or parabolas for the curve. Then, we have

$$\int_{x_0}^{x_n} y\,dx = \frac{h}{3}[y_0 + 4(y_1 + y_3 + y_5 + \cdots + y_{n-1}) + 2(y_2 + y_4 + \cdots y_{n-2}) + y_n]$$

It is referred to as Simpson's 1/3rd rule.

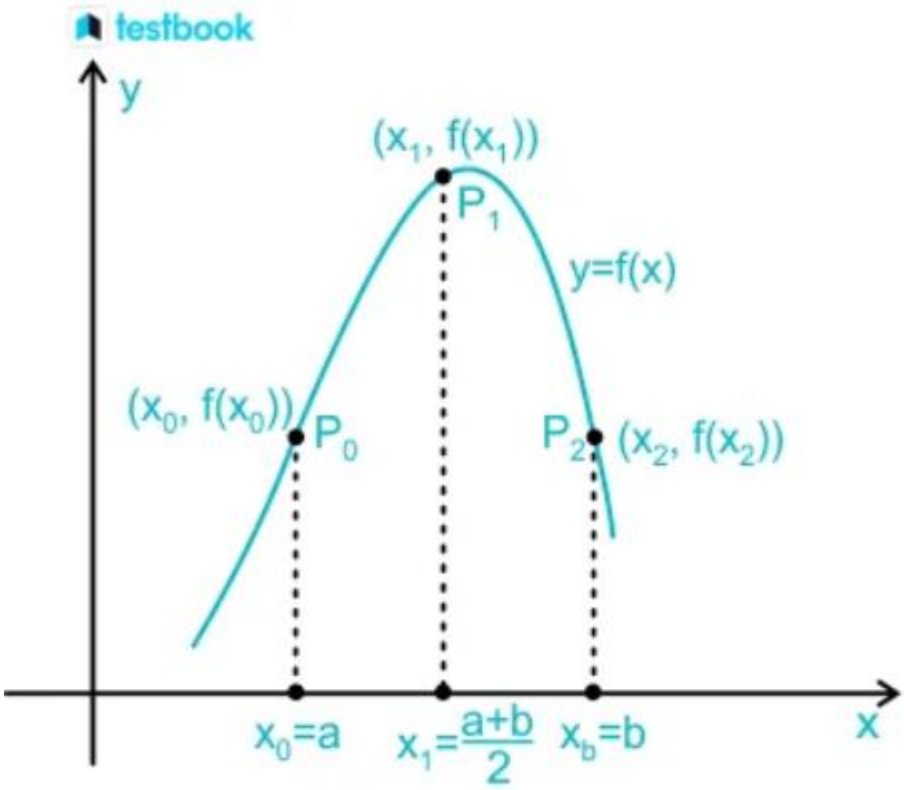
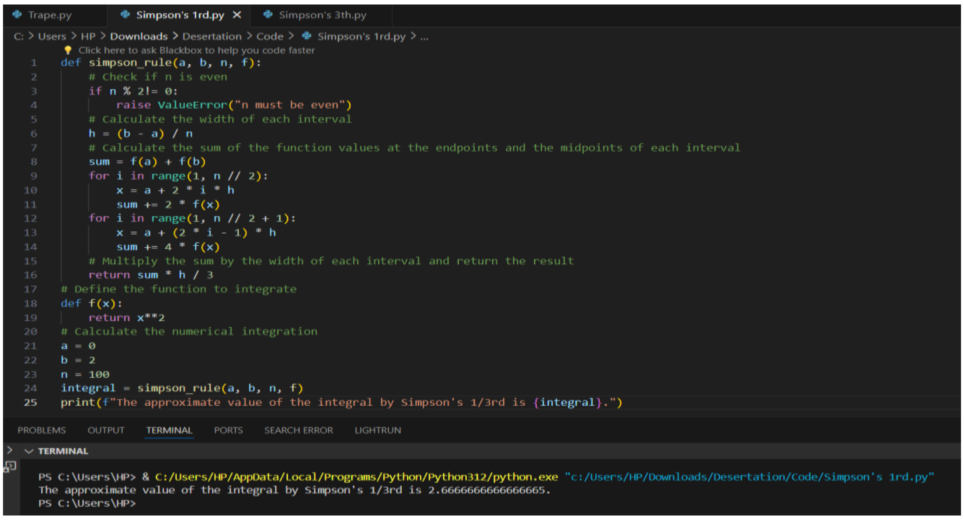


Fig 11: *Graph representing Simpson's 1/3rd Rule*

Note that the total range must be divided into an even number of subintervals of width h in order to comply with this condition.

Code:



```
Trape.py    Simpson's 1rd.py ×    Simpson's 3th.py
C: > Users > HP > Downloads > Desertation > Code > Simpson's 1rd.py > ...
Click here to ask Blackbox to help you code faster
def simpson_rule(a, b, n, f):
    # Check if n is even
    if n % 2!= 0:
        raise ValueError("n must be even")
    # Calculate the width of each interval
    h = (b - a) / n
    # Calculate the sum of the function values at the endpoints and the midpoints of each interval
    sum = f(a) + f(b)
    for i in range(1, n // 2):
        x = a + 2 * i * h
        sum += 2 * f(x)
    for i in range(1, n // 2 + 1):
        x = a + (2 * i - 1) * h
        sum += 4 * f(x)
    # Multiply the sum by the width of each interval and return the result
    return sum * h / 3
# Define the function to integrate
def f(x):
    return x**2
# Calculate the numerical integration
a = 0
b = 2
n = 100
integral = simpson_rule(a, b, n, f)
print(f"The approximate value of the integral by Simpson's 1/3rd is {integral}.")

PROBLEMS    OUTPUT    TERMINAL    PORTS    SEARCH ERROR    LIGHTRUN
TERMINAL
PS C:\Users\HP> & C:/Users/HP/AppData/Local/Programs/Python/Python312/python.exe "c:/Users/HP/Downloads/Desertation/Code/Simpson's 1rd.py"
The approximate value of the integral by Simpson's 1/3rd is 2.6666666666666665.
PS C:\Users\HP>
```

Fig 12: *Basic code doing numerical integration (using Simpson's 1/3rd rule)*

- **Simson's 3/8th Rule:**

By selecting n = 3, we can observe that all differences larger than the third will become zero in the Newton-Cotes quadrature general formula above, and we obtain

$$\int_{x_0}^{x_n} y\,dx = \frac{3h}{8}(y_0 + 3y, +3y_2 + 2y_3 + \cdots + 2y_{n-3} + 3y_{n-2} + 3y_{n-1} + y_n)$$

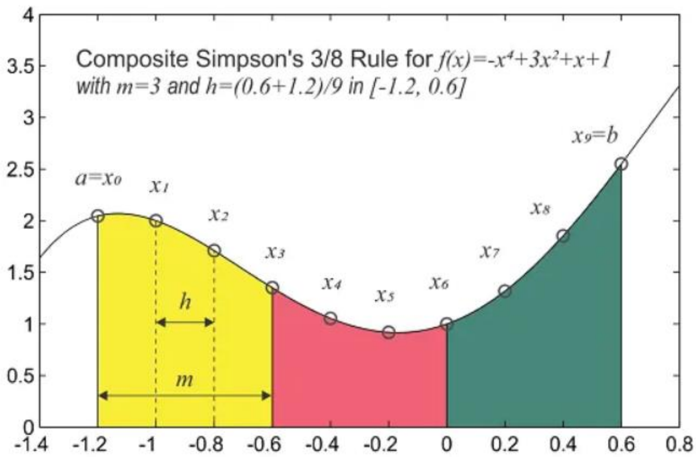


***Fig 13:** Graph representing Simpson's 3/8th Rule*

This rule, known as Simpson's (3/8th)-rule, is not as precise as Simpson's rule; the main term contributing to this formula's mistake is $-\left(\frac{3}{80}\right) h^5 y^{iv}(\bar{x})$.

Code:

```
Trape.py    Simpson's 1rd.py    Simpson's 3th.py
C: > Users > HP > Downloads > Desertation > Code > Simpson's 3th.py > ...
Click here to ask Blackbox to help you code faster
def simpson_38_rule(a, b, n, f):
    # Check if n is a multiple of 3
    if n % 3!= 0:
        raise ValueError("n must be a multiple of 3")
    # Calculate the width of each interval
    h = (b - a) / n
    # Calculate the sum of the function values at the endpoints and the midpoints of each interval
    sum = f(a) + f(b)
    for i in range(1, n // 3):
        x = a + 3 * i * h
        sum += 3 * f(x)
    for i in range(1, n // 3 + 1):
        x = a + (3 * i - 1) * h
        sum += 2 * f(x)
    # Multiply the sum by the width of each interval and return the result
    return sum * h * 3 / 8
# Define the function to integrate
def f(x):
    return x**2
# Calculate the numerical integration
a = 0
b = 2
n = 102
integral = simpson_38_rule(a, b, n, f)
print(f"The approximate value of the integral by Simpson's 3/8th is {integral}.")

PROBLEMS    OUTPUT    TERMINAL    PORTS    SEARCH ERROR    LIGHTRUN
TERMINAL
PS C:\Users\HP> & C:/Users/HP/AppData/Local/Programs/Python/Python312/python.exe "c:/Users/HP/Downloads/Desertation/Code/Simpson's 3th.py"
The approximate value of the integral by Simpson's 3/8th is 1.6621011149557858.
PS C:\Users\HP>
```

***Fig 14:** Basic code doing numerical integration (using Simpson's 3/8th rule)*

4.4. Gaussian Quadrature:

This method enhances accuracy by optimally choosing both the evaluation points and their weights. It is particularly effective when the integrand is well-behaved over the interval [-1, 1], using roots of orthogonal polynomials like Legendre polynomials.

Numerical integration provides essential tools for approximating integrals where analytical solutions are limited or infeasible. Through methods like the Newton-Cotes formulas and Gaussian quadrature, practitioners can tackle a wide array of integral problems, facilitating advanced scientific and engineering analyses. As computational techniques evolve, so does the scope for applying these methods to more complex and varied problems, underscoring their ongoing relevance in technological advancement.

Now designing a GUI of a calculator which calculate equations using numerical integration [used Simpson's $\frac{3}{8}^{th}$ rule] (using PyQt package):

```python
import sys
from PyQt5.QtWidgets import QApplication, QWidget, QVBoxLayout, QGridLayout, QLabel, QLineEdit, QPushButton
import math
from sympy import symbols


def simpsons_3_8_rule(f, a, b, n):
    """
    Numerical integration using Simpson's 3/8th rule.
    f: Function to integrate.
    a: Lower limit of integration.
    b: Upper limit of integration.
    n: Number of sub-intervals.
    """
    h = (b - a) / n
    result = f(a) + f(b)
    for i in range(1, n):
        if i % 3 == 0:
            result += 2 * f(a + i * h)
        else:
            result += 3 * f(a + i * h)
    result *= 3 * h / 8
    return result


class IntegrationApp(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Numerical Integration using Simpson's 3/8th Rule")
        self.init_ui()

    def init_ui(self):
        layout = QVBoxLayout()

        self.equation_label = QLabel("Enter the equation (use 'x' as variable):")
        self.equation_input = QLineEdit()

        self.lower_limit_label = QLabel("Lower limit:")
        self.lower_limit_input = QLineEdit()
        self.lower_limit_input.setReadOnly(True)

        self.upper_limit_label = QLabel("Upper limit:")
        self.upper_limit_input = QLineEdit()
        self.upper_limit_input.setReadOnly(True)

        self.subintervals_label = QLabel("Number of subintervals:")
        self.subintervals_input = QLineEdit()
        self.subintervals_input.setReadOnly(True)

        self.result_label = QLabel("Result:")
        self.result_display = QLabel("")

        self.calculate_button = QPushButton("Calculate")
        self.calculate_button.clicked.connect(self.calculate_integration)

        self.clear_button = QPushButton("Clear")
        self.clear_button.clicked.connect(self.clear_inputs)

```

```python

        self.backspace_button = QPushButton("Backspace")
        self.backspace_button.clicked.connect(self.backspace_input)

        self.edit_limit_button = QPushButton("Edit Limits")
        self.edit_limit_button.clicked.connect(self.enable_limit_editing)

        self.grid_layout = QGridLayout()
        buttons = ['7', '8', '9', '+', '4', '5', '6', '-', '1', '2', '3', '*', '0', '.', 'x', '/',
                   'x^2', '(', ')', 'log', 'exp']
        positions = [(i, j) for i in range(5) for j in range(4)]

        for position, button in zip(positions, buttons):
            btn = QPushButton(button)
            if button == 'x^2':
                btn.clicked.connect(self.insert_x_squared)
            else:
                btn.clicked.connect(lambda _, text=button: self.add_to_equation_input(text))
            self.grid_layout.addWidget(btn, *position)

        grid_widget = QWidget()
        grid_widget.setLayout(self.grid_layout)

        layout.addWidget(self.equation_label)
        layout.addWidget(self.equation_input)
        layout.addWidget(self.lower_limit_label)
        layout.addWidget(self.lower_limit_input)
        layout.addWidget(self.upper_limit_label)
        layout.addWidget(self.upper_limit_input)
        layout.addWidget(self.subintervals_label)
        layout.addWidget(self.subintervals_input)
        layout.addWidget(grid_widget)  # Add the grid layout widget
        layout.addWidget(self.calculate_button)
        layout.addWidget(self.clear_button)
        layout.addWidget(self.backspace_button)
        layout.addWidget(self.edit_limit_button)
        layout.addWidget(self.result_label)
        layout.addWidget(self.result_display)

        self.setLayout(layout)

    def add_to_equation_input(self, text):
        current_text = self.equation_input.text()

        # Handle arithmetic operations
        if text in ['+', '-', '*', '/']:
            # Add space before and after the operator
            text = ' ' + text + ' '

        self.equation_input.setText(current_text + text)

    def insert_x_squared(self):
        current_text = self.equation_input.text()
        x = symbols('x')
        x_squared = x**2
        self.equation_input.setText(current_text + str(x_squared))

    def enable_limit_editing(self):
        self.lower_limit_input.setReadOnly(False)
        self.upper_limit_input.setReadOnly(False)
        self.subintervals_input.setReadOnly(False)

    def clear_inputs(self):
        self.equation_input.clear()
        self.lower_limit_input.clear()
        self.upper_limit_input.clear()
        self.subintervals_input.clear()
        self.result_display.clear()

    def backspace_input(self):
        self.equation_input.backspace()

    def calculate_integration(self):
        try:
            equation = self.equation_input.text()
            lower_limit = float(self.lower_limit_input.text())
            upper_limit = float(self.upper_limit_input.text())
            n = int(self.subintervals_input.text())

            # Define the function to integrate
            def f(x):
                return eval(equation)

            # Handle 'log' and 'exp' functions
            equation = equation.replace('log', 'math.log')
            equation = equation.replace('exp', 'math.exp')

            result = simpsons_3_8_rule(f, lower_limit, upper_limit, n)
            self.result_display.setText(str(result))

        except Exception as e:
            self.result_display.setText("Error: " + str(e))


if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = IntegrationApp()
    window.show()
    sys.exit(app.exec_())
```

Fig 15: *GUI of calculator which integrate a function using numerical integration (Simpson's 3/8th Rule)*

Result –

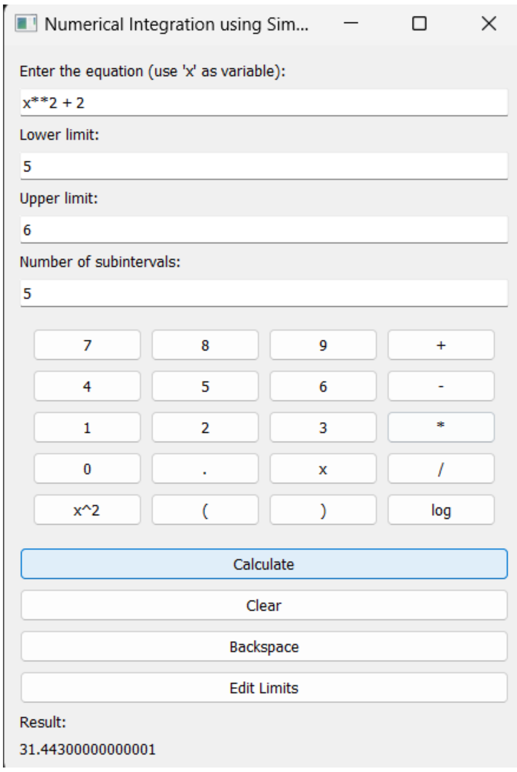


Fig 16: *Numerical Integration's GUI Result*

5. Numerical Solutions of Differential Equations:

Computational mathematics and engineering both depend on numerical solutions to differential equations. Differential equations are crucial for modeling dynamical systems in fields like physics, engineering, biology, and economics because they explain how values change over time or space. An overview of numerical techniques for approximating solutions to partial and ordinary differential equations, together with their computational considerations and applications, is given in this paper.

5.1. Numerical Methods for ODEs

The solution to ODEs is estimated at discrete points inside the domain via numerical techniques. Among the techniques frequently employed are:

5.1.1. Euler's Method

One of the most basic and straightforward numerical techniques for resolving ordinary differential equations (ODEs) is the Euler's Method. Iteratively progressing from an initial condition, it produces an approximate solution.

5.1.1.1. Code:

```
Euler's method.py
C: > Users > HP > Downloads > Desertation > Code > Euler's method.py > ...
Click here to ask Blackbox to help you code faster
import numpy as np
import matplotlib.pyplot as plt
def euler_method(f, x0, y0, xmax, h):
    x_values = np.arange(x0, xmax+h, h)
    y_values = [y0]
    for i in range(1, len(x_values)):
        y_next = y_values[-1] + h * f(x_values[i-1], y_values[-1])
        y_values.append(y_next)
    return x_values, y_values
# Example usage:
def f(x, y):
    return x + y
x0 = 0  # Initial x
y0 = 1  # Initial y
xmax = 2  # Maximum value of x
h = 0.1  # Step siz
x_values, y_values = euler_method(f, x0, y0, xmax, h)
# Print values
print("x values:", x_values)
print("y values:", y_values)
# Plotting
plt.plot(x_values, y_values, label="Euler's method")
plt.scatter(x_values, y_values, color='red', label="Computed Points")
plt.xlabel('x')
plt.ylabel('y')
plt.title('Solution of dy/dx = x + y using Euler\'s method')
plt.legend()
plt.grid(True)
plt.show()
```

Fig 17: *Basic code for solving ODEs using Euler's Method*

Result –

```
TERMINAL
PS C:\Users\HP> & C:/Users/HP/AppData/Local/Programs/Python/Python312/python.exe "c:/Users/HP/Downloads/Desertation/Code/Euler's method.py"
x values: [0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.  1.1 1.2 1.3 1.4 1.5 1.6 1.7
 1.8 1.9 2. ]
y values: [1, 1.1, 1.2200000000000002, 1.362, 1.5282, 1.72102, 1.943122, 2.1974342, 2.48717762, 2.8158953820000003, 3.1874849202, 3.6062334122200004, 4.0768567534420001, 4.6045424287
86201, 5.19499667166482, 5.854496338831303, 6.589945972714433, 7.408940569985877, 8.319834626984465, 9.331818089682912, 10.454999898651202]
```

Fig 18(a): *Result for the basic code of Euler's method*

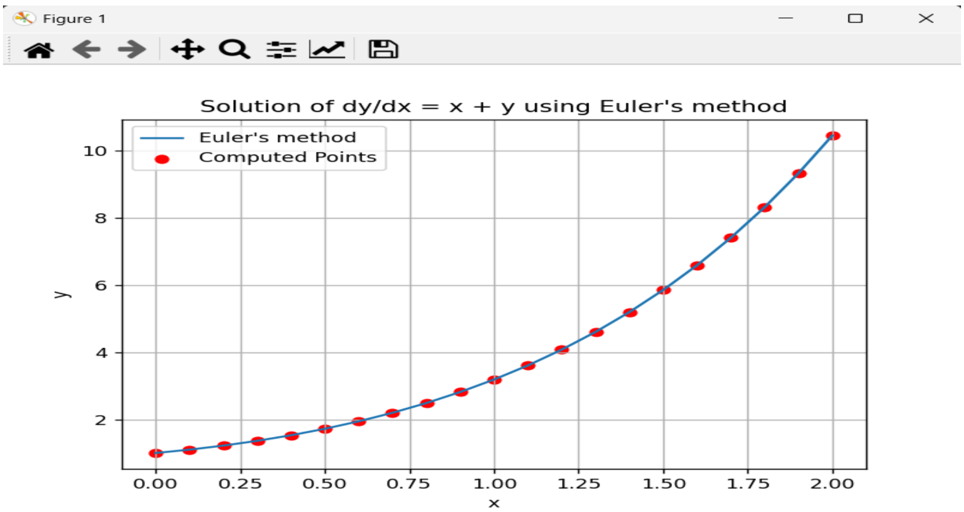


Fig 18(b): *Graph for the basic code of Euler's method*

5.1.2. Runge-Kutta Methods

A class of numerical algorithms called Runge-Kutta (RK) techniques is superior than Euler's method in solving ODEs. The most used version is the fourth-order Runge-Kutta (RK4) algorithm, which achieves a good balance between processing efficiency and accuracy.

5.1.2.1. Code:

```
Runge-Kutta Method.py ×
C: > Users > HP > Downloads > Desertation > Code > Runge-Kutta Method.py > ...
    Click here to ask Blackbox to help you code faster
import numpy as np
import matplotlib.pyplot as plt
def runge_kutta_method(f, x0, y0, xmax, h):
    x_values = np.arange(x0, xmax+h, h)
    y_values = [y0]
    for i in range(1, len(x_values)):
        k1 = h * f(x_values[i-1], y_values[-1])
        k2 = h * f(x_values[i-1] + 0.5 * h, y_values[-1] + 0.5 * k1)
        k3 = h * f(x_values[i-1] + 0.5 * h, y_values[-1] + 0.5 * k2)
        k4 = h * f(x_values[i-1] + h, y_values[-1] + k3)
        y_next = y_values[-1] + (k1 + 2*k2 + 2*k3 + k4) / 6
        y_values.append(y_next)
    return x_values, y_values
# Example usage:
def f(x, y):
    return x + y
x0 = 0  # Initial x
y0 = 1  # Initial y
xmax = 2  # Maximum value of x
h = 0.1  # Step size
x_values, y_values = runge_kutta_method(f, x0, y0, xmax, h)
# Print values
print("x values:", x_values)
print("y values:", y_values)
# Plotting
plt.plot(x_values, y_values, label="Runge-Kutta method")
plt.scatter(x_values, y_values, color='red', label="Computed Points")
plt.xlabel('x')
plt.ylabel('y')
plt.title('Solution of dy/dx = x + y using Runge-Kutta method')
plt.legend()
plt.grid(True)
plt.show()
```

***Fig 19:** Basic code for solving ODEs using Runge-Kutta Method*

Result –

```
TERMINAL
PS C:\Users\HP> & C:/Users/HP/AppData/Local/Programs/Python/Python312/python.exe "c:/Users/HP/Downloads/Desertation/Code/Runge-Kutta Method.py"
x values: [0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.  1.1 1.2 1.3 1.4 1.5 1.6 1.7
 1.8 1.9 2. ]
y values: [1, 1.1103416666666668, 1.242805141701389, 1.3997169941250756, 1.5836484801613715, 1.7974412771936765, 2.0442359241838663, 2.327503253193554, 2.65107912658 4631, 3.01920282
7560142, 3.436559488270332, 3.9083269801179634, 4.44022773555612, 5.038586020027671, 5.710391227242331, 6.463367831270763, 7.306052695558702, 8.247880512594524, 9.299278229337851, 1
0.471769403449171, 11.77808953475109]
```

***Fig 20(a):** Result for basic code of Runge-Kutta Method*

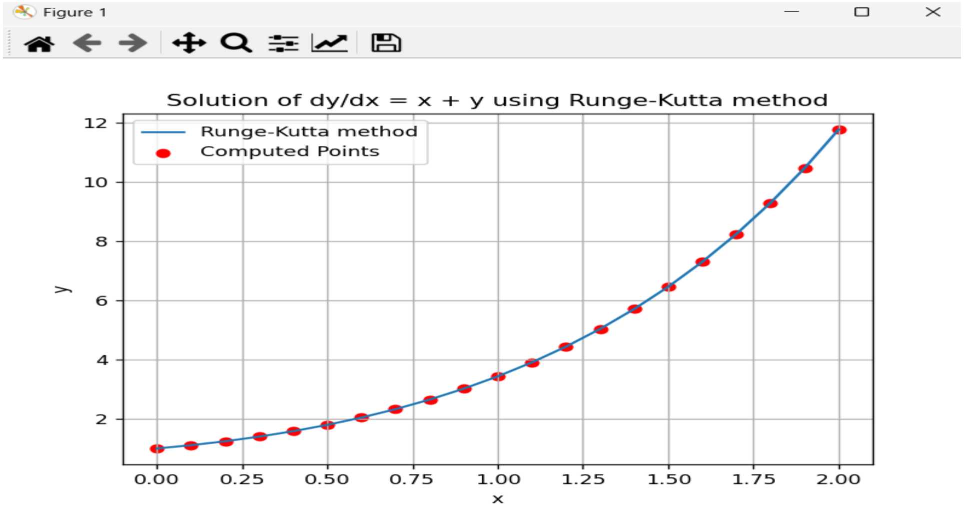


Fig 20(b): *Graph for the basic code of Runge-Kutta Method*

Now designing a GUI of a calculator which calculate ordinary differential equations using numerical methods (using PyQt package):

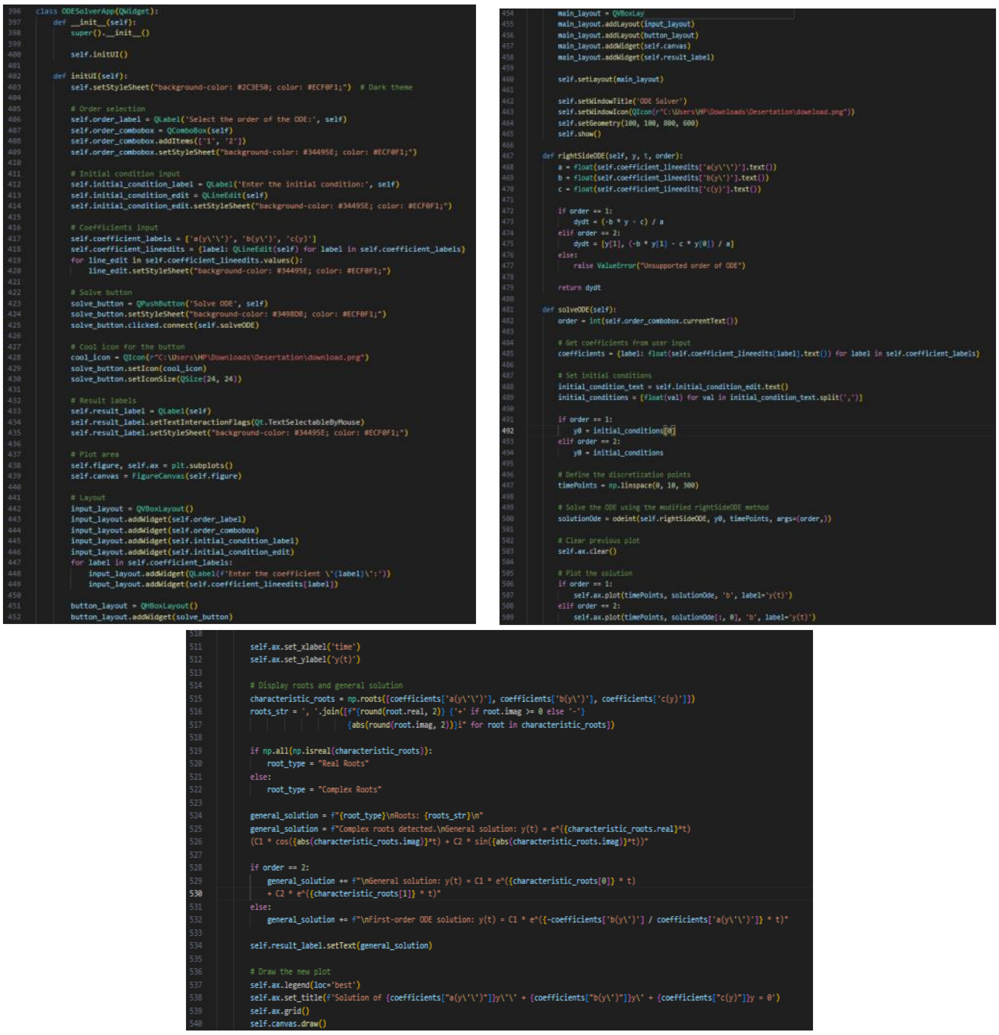


Fig 21: *GUI of calculator which calculates Ordinary Differential Equation*

Result-

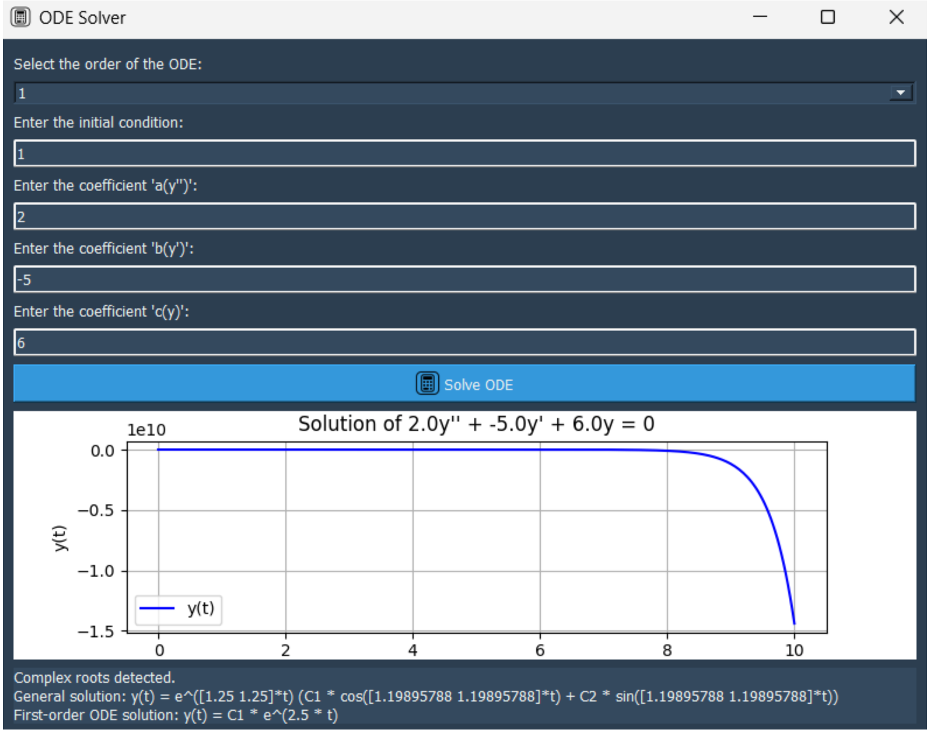


Fig 22: *Ordinary Differential Equation's GUI Result*

5.2. Numerical Methods for PDEs

PDEs are more difficult to solve numerically because they are greater dimensional. Here are some common approaches:

5.2.1. Finite Difference Methods

By employing finite differences to approximate derivatives, finite difference methods (FDMs) are numerical techniques for solving partial differential equations (PDEs). These techniques replace derivatives with difference quotients and divide the spatial domain into a grid of points.

5.2.1.1. Limitations:

- To maintain stability and accuracy, grid characteristics such as grid size and time step must be carefully tuned.
- It can be computationally demanding for big systems or fine discretization, particularly for high-dimensional situations.

5.2.1.2. Code:

PDM.py

C: > Users > HP > Downloads > Desertation > Code > PDM.py > ...

Click here to ask Blackbox to help you code faster

```python
import numpy as np
import matplotlib.pyplot as plt
# Parameters
L = 1.0  # Length of the rod
T = 0.1  # Total time
Nx = 100  # Number of spatial grid points
Nt = 1000  # Number of time steps
alpha = 0.01  # Thermal diffusivity
# Grid spacing
dx = L / (Nx - 1)
dt = T / Nt
# Initial condition
def initial_condition(x):
    return np.sin(np.pi * x)
# Boundary conditions
def boundary_condition_left(t):
    return 0.0
def boundary_condition_right(t):
    return 0.0
# Finite Difference Method
def solve_heat_equation(Nx, Nt, alpha, dx, dt):
    # Initialize solution matrix
    u = np.zeros((Nt+1, Nx))
    # Initial condition
    u[0, :] = initial_condition(np.linspace(0, L, Nx))
    # Boundary conditions
    u[:, 0] = boundary_condition_left(np.linspace(0, T, Nt+1))
    u[:, -1] = boundary_condition_right(np.linspace(0, T, Nt+1))
    # Finite Difference Iteration
    for n in range(Nt):
        for i in range(1, Nx-1):
            u[n+1, i] = u[n, i] + alpha * dt / dx**2 * (u[n, i+1] - 2*u[n, i] + u[n, i-1])
    return u
# Solve the heat equation
u = solve_heat_equation(Nx, Nt, alpha, dx, dt)
# Plotting
x_values = np.linspace(0, L, Nx)
t_values = np.linspace(0, T, Nt+1)
X, T = np.meshgrid(x_values, t_values)
fig = plt.figure(figsize=(10, 6))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, T, u, cmap='viridis')
ax.set_xlabel('Position')
ax.set_ylabel('Time')
ax.set_zlabel('Temperature')
ax.set_title('Solution of 1D Heat Equation using Finite Difference Method')
plt.show()
```

__Fig 23:__ Basic code for solving PDEs using FDM

Result –

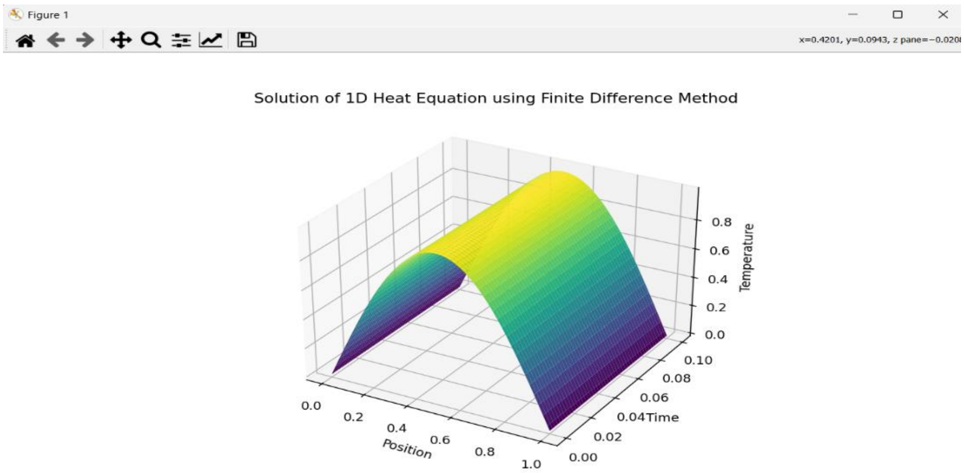


__Fig 24:__ Result for basic code of FDM

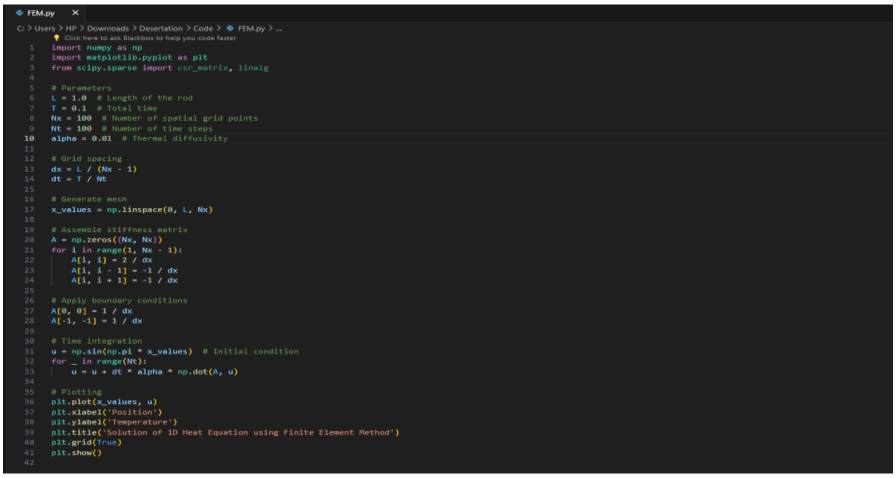
5.2.2. Finite Element Methods (FEM)

By breaking the domain up into smaller, more manageable components, numerical techniques known as Finite Element Methods (FEM) are used to approximate solutions to partial differential equations (PDEs). FEM's adaptability and ability to handle complicated geometries and boundary conditions make it a popular tool in physics and engineering.

5.2.2.1. Limitations:

- To attain precise results, a thorough mesh creation and element selection are required.
- Computing costs can be excessively high when dealing with large numbers of elements or high-dimensional environments.
- Implementing and solving the resulting linear systems can be challenging and resource-intensive.

5.2.2.2. Code:



```
import numpy as np
import matplotlib.pyplot as plt
from scipy.sparse import csr_matrix, linalg

# Parameters
L = 1.0  # Length of the rod
T = 0.1  # Total time
Nx = 100  # Number of spatial grid points
Nt = 100  # Number of time steps
alpha = 0.01  # Thermal diffusivity

# Grid spacing
dx = L / (Nx - 1)
dt = T / Nt

# Generate mesh
x_values = np.linspace(0, L, Nx)

# Assemble stiffness matrix
A = np.zeros((Nx, Nx))
for i in range(1, Nx - 1):
    A[i, i] = 2 / dx
    A[i, i - 1] = -1 / dx
    A[i, i + 1] = -1 / dx

# Apply boundary conditions
A[0, 0] = 1 / dx
A[-1, -1] = 1 / dx

# Time integration
u = np.sin(np.pi * x_values)  # Initial condition
for _ in range(Nt):
    u = u + dt * alpha * np.dot(A, u)

# Plotting
plt.plot(x_values, u)
plt.xlabel('Position')
plt.ylabel('Temperature')
plt.title('Solution of 1D Heat Equation using Finite Element Method')
plt.grid(True)
plt.show()
```

Fig 25: *Basic code for solving ODEs using FEM*

Result-

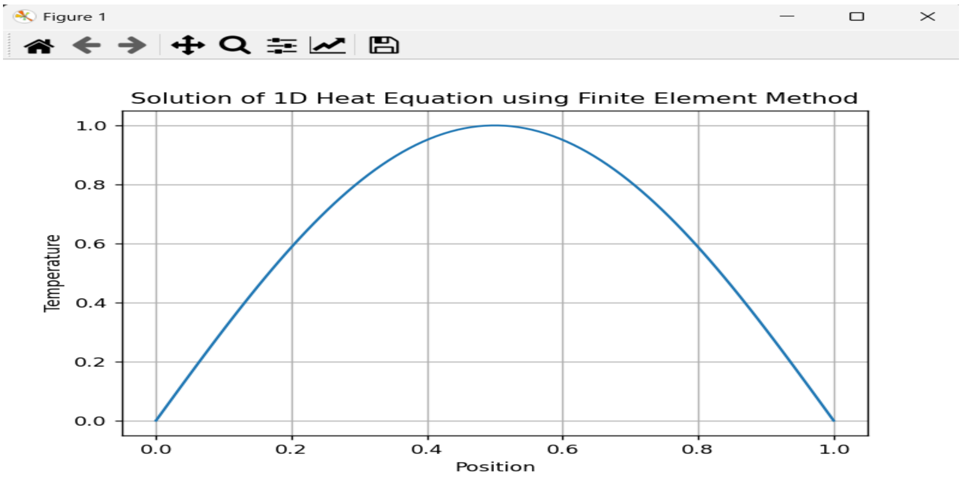


Fig 26: *Result for basic code of FEM*

Now designing a GUI of a calculator which calculate partial differential equations using numerical methods (using PyQt package):

```python
import sys
import numpy as np
import matplotlib.pyplot as plt
from PyQt5.QtWidgets import *
from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as FigureCanvas

class PDESolver:
    def __init__(self, equation_type, solver_type):
        self.equation_type = equation_type
        self.solver_type = solver_type

    def solve(self, equation, **kwargs):
        # Implement Finite Element Method solver based on the equation and solver type
        # Placeholder implementation
        x = np.linspace(0, 1, 100)
        y = np.sin(np.pi * x)
        return x, y

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("FEM Solver")
        self.setGeometry(100, 100, 800, 600)

        self.central_widget = QWidget()
        self.setCentralWidget(self.central_widget)

        self.layout = QVBoxLayout()
        self.central_widget.setLayout(self.layout)

        self.equation_label = QLabel("Equation Type:")
        self.layout.addWidget(self.equation_label)
        self.equation_combo = QComboBox()
        self.equation_combo.addItems(["Custom Equation", "Heat Equation", "Wave Equation"])
        self.equation_combo.currentIndexChanged.connect(self.on_equation_combo_changed)
        self.layout.addWidget(self.equation_combo)

        self.equation_text_edit = QTextEdit()
        self.layout.addWidget(self.equation_text_edit)

        self.solver_label = QLabel("Solver Type:")
        self.layout.addWidget(self.solver_label)
        self.solver_combo = QComboBox()
        self.solver_combo.addItems(["FEM"])  # Add more solvers as needed
        self.layout.addWidget(self.solver_combo)

        self.params_label = QLabel("Parameters:")
        self.layout.addWidget(self.params_label)
        self.params_edit = QLineEdit()
        self.layout.addWidget(self.params_edit)

        self.solve_button = QPushButton("Solve")
        self.solve_button.clicked.connect(self.solve)
        self.layout.addWidget(self.solve_button)

        self.clear_button = QPushButton("Clear")
        self.clear_button.clicked.connect(self.clear)
        self.layout.addWidget(self.clear_button)

        self.canvas = FigureCanvas(plt.figure())
        self.layout.addWidget(self.canvas)

    def on_equation_combo_changed(self, index):
        if index == 0:
            self.equation_text_edit.setEnabled(True)
        else:
            self.equation_text_edit.setEnabled(False)

    def solve(self):
        equation_type = self.equation_combo.currentText()
        solver_type = self.solver_combo.currentText()
        if equation_type == "Custom Equation":
            equation = self.equation_text_edit.toPlainText()
        else:
            equation = None
        params = self.params_edit.text()

        fem_solver = FEMSolver(equation_type, solver_type)
        x, y = fem_solver.solve(equation, parameters=params)

        self.plot_solution(x, y)

    def plot_solution(self, x, y):
        ax = self.canvas.figure.add_subplot(111)
        ax.clear()
        ax.plot(x, y)
        ax.set_xlabel("X")
        ax.set_ylabel("Y")
        ax.set_title("Finite Element Method Solution")

        # Add annotations (example: mark max value)
        max_index = np.argmax(y)
        max_x = x[max_index]
        max_y = y[max_index]
        ax.annotate(f'Max Value: {max_y:.2f}', xy=(max_x, max_y), xytext=(max_x, max_y + 0.5),
                    arrowprops=dict(facecolor='black', shrink=0.05),
                    )

        # Show values on specific points
        for i in range(0, len(x), len(x)//10):  # Show values at 10 equally spaced points
            ax.text(x[i], y[i], f'({x[i]:.2f}, {y[i]:.2f})', ha='center', va='bottom')

        self.canvas.draw()

    def clear(self):
        self.canvas.figure.clear()
        self.canvas.draw()

if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = MainWindow()
    window.show()
    sys.exit(app.exec_())
```

***Fig 27:** GUI of calculator which calculates partial differential equation*

Result-

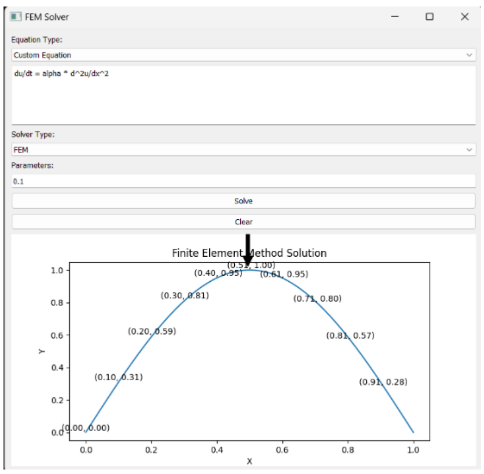


***Fig 28:** Partial Differential Equation's GUI Result*

Numerical solutions to differential equations are essential for modeling and simulating complicated systems in a variety of scientific and engineering areas. Using numerical approaches adapted to the topic at hand, practitioners can acquire insights into the behavior of dynamic systems, anticipate future outcomes, and optimize designs. As computational tools improve, numerical solutions continue to fuel innovation and discovery, pushing the limits of what is feasible in science, engineering, and technology.

6. Numerical Methods

Numerical methods are essential tools in computational mathematics because they provide approaches for addressing mathematical problems when analytical solutions are difficult or impossible to find. Newton-Raphson, Secant, Bisection, and Regula Falsi numerical methods are very effective and versatile in solving root-finding issues. This study delves deeply into these methodologies, discussing their concepts, algorithms, applications, and computational considerations.

6.1. Bisection Method

The Bisection technique is a straightforward and reliable root-finding procedure based on the intermediate value theorem that ensures convergence to a root inside a certain interval when the function changes sign.

6.1.1. Formula

$$x_r = \frac{a+b}{2}$$

6.2. Regular-Falsi Method

The Regula Falsi method is a variation on the Bisection method that employs linear interpolation to determine the root.

6.2.1. Formula

$$x_1 = a - \frac{f(a)}{f(b) - f(a)}(b - a)$$

6.3. Newton-Raphson Method

The Newton-Raphson technique is an iterative root-finding algorithm that use the concept of linear approximation to modify an initial prediction in order to converge on the root of a specific function.

6.3.1. Formula

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

6.4. Secant Method

The Secant technique is an iterative root-finding algorithm that approximates the derivative using finite differences, making it appropriate for situations in which the derivative is not known.

6.4.1. Formula

$$\frac{x_{i-1} f_i - x_i f_{i-1}}{f_i - f_{i-1}}$$

Now designing a GUI of a calculator which calculate partial differential equations using numerical methods (using PyQt package):

```python
class TranscendentalEquationSolver(QWidget):
    def __init__(self):
        super().__init__()

        self.initUI()

    def initUI(self):
        # Widgets
        self.equation_label = QLabel("Enter the transcendental equation (use 'x' as the variable):")
        self.equation_input = QLineEdit(self)

        self.guess_label = QLabel("Enter an initial guess for the root:")
        self.guess_input = QDoubleSpinBox(self)
        self.guess_input.setRange(-1000, 1000)
        self.guess_input.setSingleStep(0.1)

        self.result_text = QTextEdit(self)
        self.result_text.setReadOnly(True)

        self.solve_button = QPushButton("Solve", self)
        self.solve_button.clicked.connect(self.solve)

        self.clear_button = QPushButton("Clear", self)
        self.clear_button.clicked.connect(self.clear)

        # Layout
        layout = QVBoxLayout(self)
        layout.addWidget(self.equation_label)
        layout.addWidget(self.equation_input)
        layout.addWidget(self.guess_label)
        layout.addWidget(self.guess_input)
        layout.addWidget(self.solve_button)
        layout.addWidget(self.clear_button)
        layout.addWidget(self.result_text)

        self.setLayout(layout)

        self.setGeometry(300, 300, 400, 300)
        self.setWindowTitle('Transcendental Equation Solver')

        # Apply qdarkstyle globally
        self.setStyleSheet(qdarkstyle.load_stylesheet_pyqt5())

    def solve(self):
        equation_str = self.equation_input.text()
        x = sp.symbols('x')
        equation = sp.sympify(equation_str)
        derivative = sp.diff(equation, x)
        x0 = self.guess_input.value()

        max_iterations = 20
        tolerance = 1e-6

        for i in range(max_iterations):
            f_x = equation.subs(x, x0)
            f_prime_x = derivative.subs(x, x0)
            x0 = x0 - f_x.evalf() / f_prime_x.evalf()

            self.result_text.append(f"Iteration {i + 1}: x = {x0}")

            if sp.N(f_x.evalf()).is_zero:
                self.result_text.append(f"Converged to root: {x0}")
                break
        else:
            self.result_text.append("Maximum number of iterations reached. Solution may not have converged.")

    def clear(self):
        # Clear the input fields and result text
        self.guess_input.clear()
        self.result_text.clear()
```

Fig 29: *GUI of calculator which calculates transcendental equation using numerical methods*

Result-

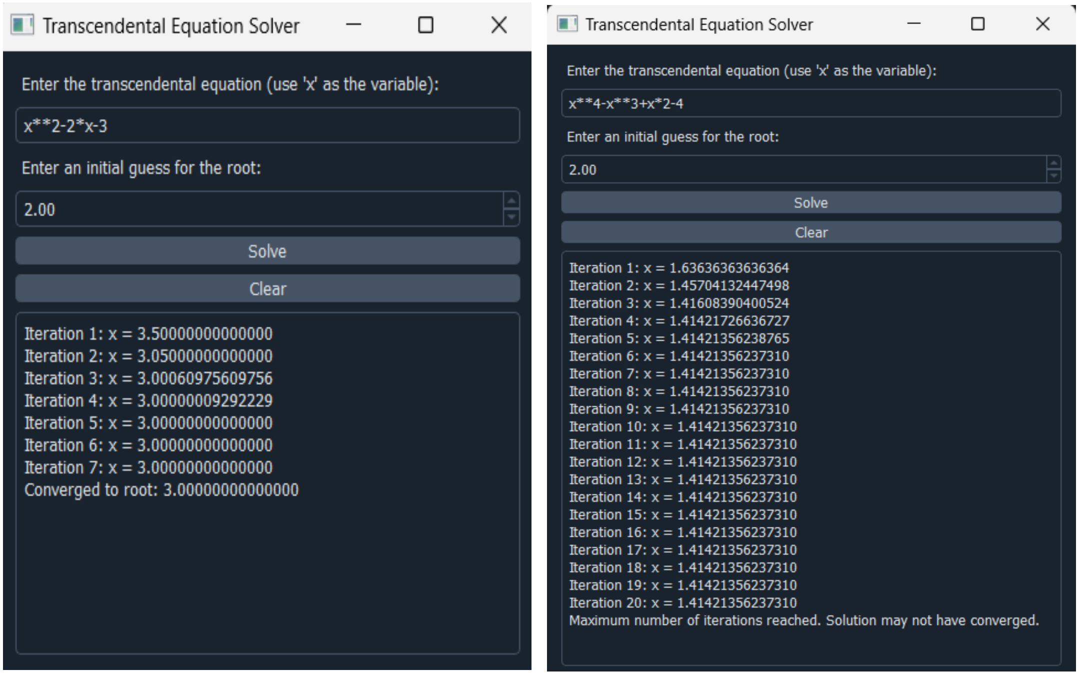


Fig 30: *Numerical Method's GUI Result*

These are the detailed information on all the topics covered in the GUI of Scientific Computing.

References:

[1] S. C. Malik, Savita Arora, *Mathematical Analysis*
[2] S.S. Sastry, *Introductory Methods Of Numerical Analysis*
[3] Gajendra Purohit, *Youtube*

CHAPTER 3
THE PROTOTYPE

Combining all the mathematical methods mentioned in the previous chapter with their respective codes, we get a prototype which look like this:

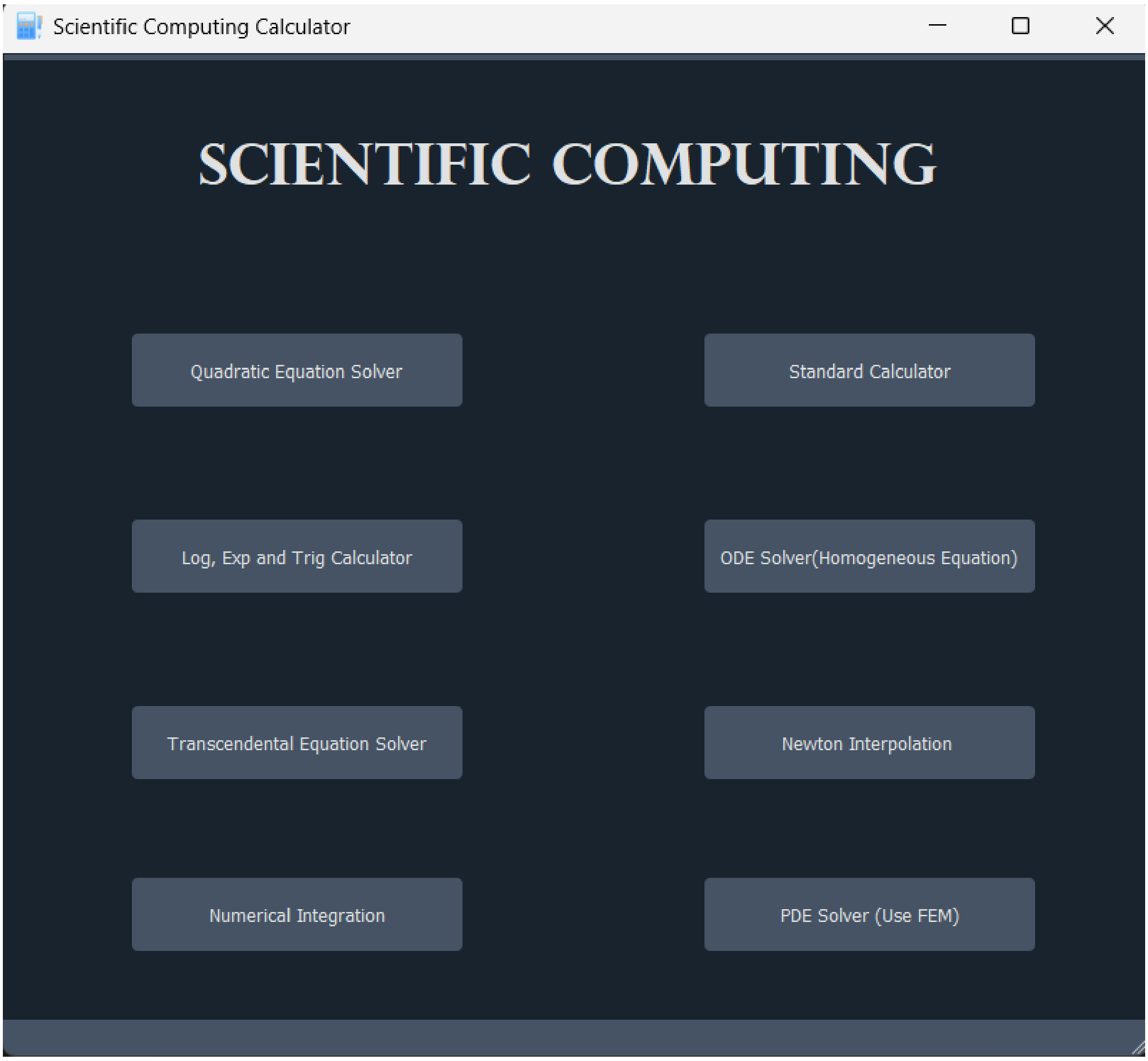


Fig 31: *The Final Prototype*

The code for this whole GUI is given below:

```python
import sys
from math import log, exp, sin, cos, tan, radians, degrees, log10, isnan
from PyQt5 import QtCore, QtGui, QtWidgets
from PyQt5.QtWidgets import*
from PyQt5.QtGui import*
from PyQt5.QtCore import *
from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as FigureCanvas
from scipy.integrate import odeint
import qdarkstyle
import numpy as np
import matplotlib.pyplot as plt
import sympy as sp

class Calculator(QWidget):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("Log, exp & trig Calculator")
        self.setGeometry(100,100,400,400)

        self.expression_lineedit = QLineEdit(self)
        self.expression_lineedit.setPlaceholderText("Enter expression")

        self.result_lineedit = QLineEdit(self)
        self.result_lineedit.setReadOnly(True)

        self.create_buttons()
        self.create_layout()

    def create_buttons(self):
        buttons_layout = QGridLayout()

        buttons = [
          ('7', 0, 0),
           ('8', 0, 1),
           ('9', 0, 2),
           ('4', 1, 0),
           ('5', 1, 1),
           ('6', 1, 2),
           ('1', 2, 0),
           ('2', 2, 1),
           ('3', 2, 2),
           ('0', 3, 0),
           ('.', 3, 2),
           ('+', 0, 3),
           ('-', 1, 3),
           ('*', 2, 3),
           ('/', 3, 3),
           ('^', 4, 0),
           ('ln', 4, 1),
           ('exp', 4, 2),
           ('sin', 5, 0),
           ('cos', 5, 1),
           ('tan', 5, 2),
           ('(', 3, 1),
           (')', 3, 2),
           ('=', 6, 3),
           ('⌫', 4, 3),
           ('AC', 5, 3),
        ]

        for btn_text, row, col in buttons:
            button = QPushButton(btn_text, self)
            button.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Expanding)
            button.clicked.connect(lambda _, text=btn_text: self.button_pressed(text))
            buttons_layout.addWidget(button, row, col)

        self.buttons_layout = buttons_layout

    def create_layout(self):
        main_layout = QVBoxLayout()

        main_layout.addWidget(self.expression_lineedit)
        main_layout.addWidget(self.result_lineedit)
        main_layout.addLayout(self.buttons_layout)

        self.setLayout(main_layout)

        # Set dark theme
        self.setStyleSheet("""
            QWidget {
                background-color: #282828;
                color: #f8f8f2;
            }
            QLineEdit {
                background-color: #3c3836;
                color: #f8f8f2;
                border: none;
            }
            QPushButton {
                background-color: #3c3836;
                color: #f8f8f2;
                border: none;
            }
            QPushButton:hover {
                background-color: #45413b;
            }
        """)

    def button_pressed(self, text):
        current_text = self.expression_lineedit.text()

        if text == '=':
            try:
                result = eval(current_text)
                self.result_lineedit.setText(str(result))
            except Exception as e:
                self.result_lineedit.setText("Error")
        elif text == '⌫':
            new_text = current_text[:-1]
            self.expression_lineedit.setText(new_text)
        elif text == 'AC':
            self.expression_lineedit.clear()
            self.result_lineedit.clear()
        else:
            new_text = current_text + text
            self.expression_lineedit.setText(new_text)

class NewtonInterpolationCalculator(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Newton Interpolation Calculator")
        self.setGeometry(250, 350, 500, 300)
        self.initUI()

    def initUI(self):
        layout = QVBoxLayout()

        self.data_points_layout = QVBoxLayout()
        self.add_data_point_button = QPushButton("Add Data Point")
        self.add_data_point_button.clicked.connect(self.add_data_point)
        layout.addLayout(self.data_points_layout)
        layout.addWidget(self.add_data_point_button)

        self.x_interp_edit = QLineEdit()
        self.x_interp_edit.setPlaceholderText("Enter x for interpolation")
        self.interpolated_value_label = QLabel("Interpolated Value:")
        layout.addWidget(self.x_interp_edit)
        layout.addWidget(self.interpolated_value_label)

        self.calculate_button = QPushButton("Calculate")
        self.calculate_button.clicked.connect(self.calculate_interpolation)
        layout.addWidget(self.calculate_button)

        self.setLayout(layout)

        self.data_points = []
```

```python
    def add_data_point(self):
        data_point_layout = QHBoxLayout()
        x_edit = QLineEdit()
        y_edit = QLineEdit()
        data_point_layout.addWidget(QLabel("x:"))
        data_point_layout.addWidget(x_edit)
        data_point_layout.addWidget(QLabel("y:"))
        data_point_layout.addWidget(y_edit)
        self.data_points_layout.addLayout(data_point_layout)
        self.data_points.append((x_edit, y_edit))

    def calculate_interpolation(self):
        x_values = []
        y_values = []
        for x_edit, y_edit in self.data_points:
            x_value = float(x_edit.text())
            y_value = float(y_edit.text())
            x_values.append(x_value)
            y_values.append(y_value)

        x_interp = float(self.x_interp_edit.text())

        x_values = np.array(x_values)
        y_values = np.array(y_values)

        interpolated_value = self.newton_interpolation(x_values, y_values, x_interp)
        self.interpolated_value_label.setText(f"Interpolated Value: {interpolated_value}")

    def newton_interpolation(self, x, y, x_interp):
        coef = self.divided_difference(x, y)
        n = len(x)
        result = coef[-1]
        for i in range(n - 2, -1, -1):
            result = result * (x_interp - x[i]) + coef[i]
        return result

    def divided_difference(self, x, y):
        n = len(y)
        coef = np.copy(y)
        for j in range(1, n):
            coef[j:] = (coef[j:] - coef[j - 1:-1]) / (x[j:] - x[:-j])
        return coef

class QuadraticEquationSolver(QWidget):
    def __init__(self):
        super().__init__()

        self.setWindowTitle('Quadratic Equation Solver')
        self.setWindowIcon(QIcon(r"C:\Users\HP\Desktop\scientific-calculator.png"))
        self.setGeometry(100, 100, 400, 250)
        self.setStyleSheet("background-color: #232323; color: white;")

        # Header label with FontAwesome icon
        header_label = QLabel(self)
        header_label.setText("<font size='5' color='#4CAF50'><i class='fab fa-fort-awesome'></i></font> <font size='5'>Quadratic Equation Solver</font>")
        header_label.setAlignment(Qt.AlignmentFlag.AlignCenter)

        # Labels for coefficients
        label_a = QLabel('Coefficient a(x^2):', self)
        label_b = QLabel('Coefficient b(x):', self)
        label_c = QLabel('Coefficient c:', self)

        # Input fields for coefficients
        self.a_input = QLineEdit(self)
        self.b_input = QLineEdit(self)
        self.c_input = QLineEdit(self)

        # Label to display the result
        self.result_label = QLabel(self)
        self.result_label.setAlignment(Qt.AlignmentFlag.AlignCenter)

        # Solve button with hover effect
        solve_button = QPushButton('Solve', self)
        solve_button.clicked.connect(self.solve_quadratic)
        solve_button.setStyleSheet('QPushButton {background-color: #4CAF50; color: white; padding: 8px 16px; border: none; border-radius: 4px;}'
                                   'QPushButton:hover {background-color: #45a049;}')

        # Set size policy for each widget to expand to fill available space
        header_label.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Preferred)
        label_a.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Preferred)
        self.a_input.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Preferred)
        label_b.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Preferred)
        self.b_input.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Preferred)
        label_c.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Preferred)
        self.c_input.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Preferred)
        solve_button.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Preferred)

        # Create layouts
        input_layout = QVBoxLayout()
        input_layout.addWidget(label_a)
        input_layout.addWidget(self.a_input)
        input_layout.addWidget(label_b)
        input_layout.addWidget(self.b_input)
        input_layout.addWidget(label_c)
        input_layout.addWidget(self.c_input)

        result_layout = QVBoxLayout()
        result_layout.addWidget(self.result_label)

        button_layout = QHBoxLayout()
        button_layout.addWidget(solve_button)

        main_layout = QVBoxLayout(self)
        main_layout.addWidget(header_label)
        main_layout.addLayout(input_layout)
        main_layout.addLayout(result_layout)
        main_layout.addLayout(button_layout)
        main_layout.setAlignment(Qt.AlignmentFlag.AlignTop)

    def solve_quadratic(self):
        try:
            a = float(self.a_input.text())
            b = float(self.b_input.text())
            c = float(self.c_input.text())

            # Calculate the discriminant
            discriminant = b**2 - 4*a*c

            if discriminant > 0:
                root1 = (-b + discriminant**0.5) / (2*a)
                root2 = (-b - discriminant**0.5) / (2*a)
                result = f"Roots: {root1:.2f}, {root2:.2f}"
            elif discriminant == 0:
                root1 = root2 = -b / (2*a)
                result = f"Roots: {root1:.2f}"
            else:
                real_part = -b / (2*a)
                imaginary_part = (-discriminant)**0.5 / (2*a)
                result = f"Roots: {real_part:.2f} + {imaginary_part:.2f}i, {real_part:.2f} - {imaginary_part:.2f}i"

            self.animate_result(result)

        except ValueError:
            self.show_error("Invalid Input", "Please enter valid numerical coefficients.")

    def show_error(self, title, message):
        error_box = QMessageBox()
        error_box.setIcon(QMessageBox.Critical)
        error_box.setWindowTitle(title)
        error_box.setText(message)
        error_box.exec_()

    @pyqtSlot(str)
    def animate_result(self, result):
        self.result_label.setText(result)
        animation = QPropertyAnimation(self.result_label, b'opacity')
        animation.setStartValue(0)
        animation.setEndValue(1)
```

```python
        animation.setDuration(1000)

class BasicCalculator(QWidget):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("Basic Calculator")
        self.setWindowIcon(QIcon(r"C:\Users\HP\Desktop\scientific-calculator.png"))
        self.setGeometry(100, 100, 300, 400)

        # Set dark theme
        self.setStyleSheet("""
            QWidget {
                background-color: #282828;
                color: #f8f8f2;
            }
            QPushButton {
                background-color: #3c3836;
                color: #f8f8f2;
                border: none;
            }
            QPushButton:hover {
                background-color: #45413b;
            }
            QLineEdit {
                background-color: #3c3836;
                color: #f8f8f2;
                border: none;
            }
        """)

        # Create the main layout
        main_layout = QVBoxLayout()

        # Create the display widget
        self.result_display = QLineEdit()
        self.result_display.setReadOnly(True)
        self.result_display.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Preferred)
        main_layout.addWidget(self.result_display)

        # Create the button grid layout
        button_grid_layout = QVBoxLayout()

        #Row 0
        row0 = self.create_button_row(['', '', ' ', 'C'])
        button_grid_layout.addLayout(row0)

        # Row 1
        row1 = self.create_button_row(['7', '8', '9', '/'])
        button_grid_layout.addLayout(row1)

        # Row 2
        row2 = self.create_button_row(['4', '5', '6', '*'])
        button_grid_layout.addLayout(row2)

        # Row 3
        row3 = self.create_button_row(['1', '2', '3', '-'])
        button_grid_layout.addLayout(row3)

        # Row 4
        row4 = self.create_button_row(['0', '.', '=', '+'])
        button_grid_layout.addLayout(row4)

        # Add the button grid layout to the main layout
        main_layout.addLayout(button_grid_layout)

        # Set the main layout for the widget
        self.setLayout(main_layout)

    def create_button_row(self, button_texts):
        row_layout = QHBoxLayout()

        for button_text in button_texts:
            button = QPushButton(button_text)
            button.clicked.connect(self.on_button_click)
            button.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Preferred)
            row_layout.addWidget(button)

        return row_layout

    def on_button_click(self):
        button = self.sender()
        current_text = self.result_display.text()

        if button.text() == "=":
            try:
                result = str(eval(current_text))
                self.result_display.setText(result)
            except Exception as e:
                self.result_display.setText("Error")
        elif button.text() == 'C':
            self.result_display.clear()
        else:
            self.result_display.setText(current_text + button.text())


    def on_clear_click(self):
        self.result_display.clear()
class ODESolverApp(QWidget):
    def __init__(self):
        super().__init__()

        self.initUI()

    def initUI(self):
        self.setStyleSheet("background-color: #2C3E50; color: #ECF0F1;")  # Dark theme

        # Order selection
        self.order_label = QLabel('Select the order of the ODE:', self)
        self.order_combobox = QComboBox(self)
        self.order_combobox.addItems(['1', '2'])
        self.order_combobox.setStyleSheet("background-color: #34495E; color: #ECF0F1;")

        # Initial condition input
        self.initial_condition_label = QLabel('Enter the initial condition:', self)
        self.initial_condition_edit = QLineEdit(self)
        self.initial_condition_edit.setStyleSheet("background-color: #34495E; color: #ECF0F1;")

        # Coefficients input
        self.coefficient_labels = ['a(y\'\')', 'b(y\')', 'c(y)']
        self.coefficient_lineedits = {label: QLineEdit(self) for label in self.coefficient_labels}
        for line_edit in self.coefficient_lineedits.values():
            line_edit.setStyleSheet("background-color: #34495E; color: #ECF0F1;")

        # Solve button
        solve_button = QPushButton('Solve ODE', self)
        solve_button.setStyleSheet("background-color: #3498DB; color: #ECF0F1;")
        solve_button.clicked.connect(self.solveODE)

        # Cool icon for the button
        cool_icon = QIcon(r"C:\Users\HP\Downloads\Desertation\download.png")
        solve_button.setIcon(cool_icon)
        solve_button.setIconSize(QSize(24, 24))

        # Result labels
        self.result_label = QLabel(self)
        self.result_label.setTextInteractionFlags(Qt.TextSelectableByMouse)
        self.result_label.setStyleSheet("background-color: #34495E; color: #ECF0F1;")

        # Plot area
        self.figure, self.ax = plt.subplots()
        self.canvas = FigureCanvas(self.figure)

        # Layout
        input_layout = QVBoxLayout()
        input_layout.addWidget(self.order_label)
        input_layout.addWidget(self.order_combobox)
        input_layout.addWidget(self.initial_condition_label)
        input_layout.addWidget(self.initial_condition_edit)
```

```python
        for label in self.coefficient_labels:
            input_layout.addWidget(QLabel(f'Enter the coefficient \'{label}\':'))
            input_layout.addWidget(self.coefficient_lineedits[label])

        button_layout = QHBoxLayout()
        button_layout.addWidget(solve_button)

        main_layout = QVBoxLayout()
        main_layout.addLayout(input_layout)
        main_layout.addLayout(button_layout)
        main_layout.addWidget(self.canvas)
        main_layout.addWidget(self.result_label)

        self.setLayout(main_layout)

        self.setWindowTitle('ODE Solver')
        self.setWindowIcon(QIcon(r"C:\Users\HP\Downloads\Desertation\download.png"))
        self.setGeometry(100, 100, 800, 600)
        self.show()

    def rightSideODE(self, y, t, order):
        a = float(self.coefficient_lineedits['a(y\'\')'].text())
        b = float(self.coefficient_lineedits['b(y\')'].text())
        c = float(self.coefficient_lineedits['c(y)'].text())

        if order == 1:
            dydt = (-b * y - c) / a
        elif order == 2:
            dydt = [y[1], (-b * y[1] - c * y[0]) / a]
        else:
            raise ValueError("Unsupported order of ODE")

        return dydt

    def solveODE(self):
        order = int(self.order_combobox.currentText())

        # Get coefficients from user input
        coefficients = {label: float(self.coefficient_lineedits[label].text()) for label in self.coefficient_labels}

        # Set initial conditions
        initial_condition_text = self.initial_condition_edit.text()
        initial_conditions = [float(val) for val in initial_condition_text.split(',')]

        if order == 1:
            y0 = initial_conditions[0]
        elif order == 2:
            y0 = initial_conditions

        # Define the discretization points
        timePoints = np.linspace(0, 10, 300)

        # Solve the ODE using the modified rightSideODE method
        solutionOde = odeint(self.rightSideODE, y0, timePoints, args=(order,))

        # Clear previous plot
        self.ax.clear()

        # Plot the solution
        if order == 1:
            self.ax.plot(timePoints, solutionOde, 'b', label='y(t)')
        elif order == 2:
            self.ax.plot(timePoints, solutionOde[:, 0], 'b', label='y(t)')

        self.ax.set_xlabel('time')
        self.ax.set_ylabel('y(t)')

        # Display roots and general solution
        characteristic_roots = np.roots([coefficients['a(y\'\')'], coefficients['b(y\')'], coefficients['c(y)']])
        roots_str = ', '.join([f"{round(root.real, 2)} {'+' if root.imag >= 0 else '-'} {abs(round(root.imag, 2))}i" for root in characteristic_roots])

        if np.all(np.isreal(characteristic_roots)):
            root_type = "Real Roots"
        else:
            root_type = "Complex Roots"

        general_solution = f"{root_type}\nRoots: {roots_str}\n"
        general_solution = f"Complex roots detected.\nGeneral solution: y(t) = e^({characteristic_roots.real}*t) (C1 * cos({abs(characteristic_roots.imag)}*t) + C2 * sin({abs(characteristic_roots.imag)}*t))"

        if order == 2:
            general_solution += f"\nGeneral solution: y(t) = C1 * e^({characteristic_roots[0]} * t) + C2 * e^({characteristic_roots[1]} * t)"
        else:
            general_solution += f"\nFirst-order ODE solution: y(t) = C1 * e^({-coefficients['b(y\')'] / coefficients['a(y\'\')']} * t)"

        self.result_label.setText(general_solution)

        # Draw the new plot
        self.ax.legend(loc='best')
        self.ax.set_title(f'Solution of {coefficients["a(y\'\')"]}y\'\' + {coefficients["b(y\')"]}y\' + {coefficients["c(y)"]}y = 0')
        self.ax.grid()
        self.canvas.draw()

class TranscendentalEquationSolver(QWidget):
    def __init__(self):
        super().__init__()

        self.initUI()

    def initUI(self):
        # Widgets
        self.equation_label = QLabel("Enter the transcendental equation (use 'x' as the variable):")
        self.equation_input = QLineEdit(self)

        self.guess_label = QLabel("Enter an initial guess for the root:")
        self.guess_input = QDoubleSpinBox(self)
        self.guess_input.setRange(-1000, 1000)
        self.guess_input.setSingleStep(0.1)

        self.result_text = QTextEdit(self)
        self.result_text.setReadOnly(True)

        self.solve_button = QPushButton("Solve", self)
        self.solve_button.clicked.connect(self.solve)

        self.clear_button = QPushButton("Clear", self)
        self.clear_button.clicked.connect(self.clear)

        # Layout
        layout = QVBoxLayout(self)
        layout.addWidget(self.equation_label)
        layout.addWidget(self.equation_input)
        layout.addWidget(self.guess_label)
        layout.addWidget(self.guess_input)
        layout.addWidget(self.solve_button)
        layout.addWidget(self.clear_button)
        layout.addWidget(self.result_text)

        self.setLayout(layout)

        self.setGeometry(300, 300, 400, 300)
        self.setWindowTitle('Transcendental Equation Solver')

        # Apply qdarkstyle globally
        self.setStyleSheet(qdarkstyle.load_stylesheet_pyqt5())

    def solve(self):
        equation_str = self.equation_input.text()
        x = sp.symbols('x')
        equation = sp.sympify(equation_str)
        derivative = sp.diff(equation, x)
        x0 = self.guess_input.value()

        max_iterations = 20
        tolerance = 1e-6

        for i in range(max_iterations):
            f_x = equation.subs(x, x0)
            f_prime_x = derivative.subs(x, x0)
```

```python
            x0 = x0 - f_x.evalf() / f_prime_x.evalf()

            self.result_text.append(f"Iteration {i + 1}: x = {x0}")

            if sp.N(f_x.evalf()).is_zero:
                self.result_text.append(f"Converged to root: {x0}")
                break
        else:
            self.result_text.append("Maximum number of iterations reached. Solution may not have converged.")

    def clear(self):
        # Clear the input fields and result text
        self.guess_input.clear()
        self.result_text.clear()

class NumericalIntegration(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Numerical Integration using Simpson's 3/8th Rule")
        self.init_ui()

    def init_ui(self):
        layout = QVBoxLayout()

        self.equation_label = QLabel("Enter the equation (use 'x' as variable):")
        self.equation_input = QLineEdit()

        self.lower_limit_label = QLabel("Lower limit:")
        self.lower_limit_input = QLineEdit()
        self.lower_limit_input.setReadOnly(True)

        self.upper_limit_label = QLabel("Upper limit:")
        self.upper_limit_input = QLineEdit()
        self.upper_limit_input.setReadOnly(True)

        self.subintervals_label = QLabel("Number of subintervals:")
        self.subintervals_input = QLineEdit()
        self.subintervals_input.setReadOnly(True)

        self.result_label = QLabel("Result:")
        self.result_display = QLabel("")

        self.calculate_button = QPushButton("Calculate")
        self.calculate_button.clicked.connect(self.calculate_integration)

        self.clear_button = QPushButton("Clear")
        self.clear_button.clicked.connect(self.clear_inputs)

        self.backspace_button = QPushButton("Backspace")
        self.backspace_button.clicked.connect(self.backspace_input)
        self.edit_limit_button = QPushButton("Edit Limits")
        self.edit_limit_button.clicked.connect(self.enable_limit_editing)

        self.grid_layout = QGridLayout()
        buttons = ['7', '8', '9', '+', '4', '5', '6', '-', '1', '2', '3', '*', '0', '.', 'x', '/',
                   'x^2', '(', ')', 'log', 'exp']
        positions = [(i, j) for i in range(5) for j in range(4)]

        for position, button in zip(positions, buttons):
            btn = QPushButton(button)
            if button == 'x^2':
                btn.clicked.connect(self.insert_x_squared)
            else:
                btn.clicked.connect(lambda _, text=button: self.add_to_equation_input(text))
            self.grid_layout.addWidget(btn, *position)

        grid_widget = QWidget()
        grid_widget.setLayout(self.grid_layout)

        layout.addWidget(self.equation_label)
        layout.addWidget(self.equation_input)
        layout.addWidget(self.lower_limit_label)
        layout.addWidget(self.lower_limit_input)
        layout.addWidget(self.upper_limit_label)
        layout.addWidget(self.upper_limit_input)
        layout.addWidget(self.subintervals_label)
        layout.addWidget(self.subintervals_input)
        layout.addWidget(grid_widget)  # Add the grid layout widget
        layout.addWidget(self.calculate_button)
        layout.addWidget(self.clear_button)
        layout.addWidget(self.backspace_button)
        layout.addWidget(self.edit_limit_button)
        layout.addWidget(self.result_label)
        layout.addWidget(self.result_display)

        self.setLayout(layout)

    def simpsons_3_8_rule(f, a, b, n):
        """
        Numerical integration using Simpson's 3/8th rule.
        f: Function to integrate.
        a: Lower limit of integration.
        b: Upper limit of integration.
        n: Number of sub-intervals.
        """
        h = (b - a) / n
        result = f(a) + f(b)
        for i in range(1, n):
            if i % 3 == 0:
                result += 2 * f(a + i * h)
            else:
                result += 3 * f(a + i * h)
        result *= 3 * h / 8
        return result

    def add_to_equation_input(self, text):
        current_text = self.equation_input.text()

        # Handle arithmetic operations
        if text in ['+', '-', '*', '/']:
            # Add space before and after the operator
            text = ' ' + text + ' '

        self.equation_input.setText(current_text + text)

    def insert_x_squared(self):
        current_text = self.equation_input.text()
        x = symbols('x')
        x_squared = x**2
        self.equation_input.setText(current_text + str(x_squared))

    def enable_limit_editing(self):
        self.lower_limit_input.setReadOnly(False)
        self.upper_limit_input.setReadOnly(False)
        self.subintervals_input.setReadOnly(False)

    def clear_inputs(self):
        self.equation_input.clear()
        self.lower_limit_input.clear()
        self.upper_limit_input.clear()
        self.subintervals_input.clear()
        self.result_display.clear()

    def backspace_input(self):
        self.equation_input.backspace()

    def calculate_integration(self):
        try:
            equation = self.equation_input.text()
            lower_limit = float(self.lower_limit_input.text())
            upper_limit = float(self.upper_limit_input.text())
            n = int(self.subintervals_input.text())

            # Define the function to integrate
            def f(x):
                return eval(equation)

```

```python
            # Handle 'log' and 'exp' functions
            equation = equation.replace('log', 'math.log')
            equation = equation.replace('exp', 'math.exp')

            result = simpsons_3_8_rule(f, lower_limit, upper_limit, n)
            self.result_display.setText(str(result))

        except Exception as e:
            self.result_display.setText("Error: " + str(e))

class PDESolver(QWidget):
    def __init__(self, equation_type, solver_type):
        self.equation_type = equation_type
        self.solver_type = solver_type

    def solve(self, equation, **kwargs):
        x = np.linspace(0, 1, 100)
        y = np.sin(np.pi * x)
        return x, y

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("FEM Solver")
        self.setGeometry(100, 100, 800, 600)

        self.central_widget = QWidget()
        self.setCentralWidget(self.central_widget)

        self.layout = QVBoxLayout()
        self.central_widget.setLayout(self.layout)

        self.equation_label = QLabel("Equation Type:")
        self.layout.addWidget(self.equation_label)
        self.equation_combo = QComboBox()
        self.equation_combo.addItems(["Custom Equation", "Heat Equation", "Wave Equation"])  # Add more equations as needed
        self.equation_combo.currentIndexChanged.connect(self.on_equation_combo_changed)
        self.layout.addWidget(self.equation_combo)

        self.equation_text_edit = QTextEdit()
        self.layout.addWidget(self.equation_text_edit)

        self.solver_label = QLabel("Solver Type:")
        self.layout.addWidget(self.solver_label)
        self.solver_combo = QComboBox()
        self.solver_combo.addItems(["FEM"])  # Add more solvers as needed
        self.layout.addWidget(self.solver_combo)

        self.params_label = QLabel("Parameters:")
        self.layout.addWidget(self.params_label)
        self.params_edit = QLineEdit()
        self.layout.addWidget(self.params_edit)

        self.solve_button = QPushButton("Solve")
        self.solve_button.clicked.connect(self.solve)
        self.layout.addWidget(self.solve_button)

        self.clear_button = QPushButton("Clear")
        self.clear_button.clicked.connect(self.clear)
        self.layout.addWidget(self.clear_button)

        self.canvas = FigureCanvas(plt.figure())
        self.layout.addWidget(self.canvas)

    def on_equation_combo_changed(self, index):
        if index == 0:
            self.equation_text_edit.setEnabled(True)
        else:
            self.equation_text_edit.setEnabled(False)

    def solve(self):
        equation_type = self.equation_combo.currentText()
        solver_type = self.solver_combo.currentText()
        if equation_type == "Custom Equation":
            equation = self.equation_text_edit.toPlainText()
        else:
            equation = None
        params = self.params_edit.text()

        fem_solver = FEMSolver(equation_type, solver_type)
        x, y = fem_solver.solve(equation, parameters=params)

        self.plot_solution(x, y)

    def plot_solution(self, x, y):
        ax = self.canvas.figure.add_subplot(111)
        ax.clear()
        ax.plot(x, y)
        ax.set_xlabel("X")
        ax.set_ylabel("Y")
        ax.set_title("Finite Element Method Solution")

        # Add annotations (example: mark max value)
        max_index = np.argmax(y)
        max_x = x[max_index]
        max_y = y[max_index]
        ax.annotate(f'Max Value: {max_y:.2f}', xy=(max_x, max_y), xytext=(max_x, max_y + 0.5),
                    arrowprops=dict(facecolor='black', shrink=0.05),
                    )

        # Show values on specific points
        for i in range(0, len(x), len(x)//10):  # Show values at 10 equally spaced points
            ax.text(x[i], y[i], f'({x[i]:.2f}, {y[i]:.2f})', ha='center', va='bottom')

        self.canvas.draw()

    def clear(self):
        self.canvas.figure.clear()
        self.canvas.draw()

class Ui_MainWindow(object):
    def setupUi(self, MainWindow):
        MainWindow.setObjectName("MainWindow")
        MainWindow.setWindowIcon(QIcon(r"C:\Users\HP\Desktop\pen.png"))
        MainWindow.resize(800, 700)
        MainWindow.setStyleSheet(qdarkstyle.load_stylesheet_pyqt5())
        self.centralwidget = QtWidgets.QWidget(MainWindow)
        self.centralwidget.setObjectName("centralwidget")
        self.label = QtWidgets.QLabel(self.centralwidget)
        self.label.setGeometry(QtCore.QRect(30, 20, 731, 101))
        font = QtGui.QFont()
        font.setFamily("Perpetua Titling MT")
        font.setPointSize(24)
        font.setBold(True)
        font.setWeight(75)
        self.label.setFont(font)
        self.label.setAlignment(QtCore.Qt.AlignCenter)
        self.label.setObjectName("label")
        self.qes = QtWidgets.QPushButton(self.centralwidget)
        self.qes.setGeometry(QtCore.QRect(90, 190, 231, 51))
        self.qes.setObjectName("qes")
        self.qes.clicked.connect(self.show_quadratic_solver)
        self.sc = QtWidgets.QPushButton(self.centralwidget)
        self.sc.setGeometry(QtCore.QRect(490, 190, 231, 51))
        self.sc.setObjectName("sc")
        self.sc.clicked.connect(self.show_basic_calculator)
        self.let = QtWidgets.QPushButton(self.centralwidget)
        self.let.setGeometry(QtCore.QRect(90, 320, 231, 51))
        self.let.setObjectName("let")
        self.let.clicked.connect(self.show_Calculator)
        self.ode = QtWidgets.QPushButton(self.centralwidget)
        self.ode.setGeometry(QtCore.QRect(490, 320, 231, 51))
        self.ode.setObjectName("ode")
        self.ode.clicked.connect(self.show_ODESolverApp)
        self.tes = QtWidgets.QPushButton(self.centralwidget)
        self.tes.setGeometry(QtCore.QRect(90, 450, 231, 51))
        self.tes.setObjectName("tes")
```

```python
        self.tes.clicked.connect(self.show_TranscendentalEquationSolver)
        self.nic = QtWidgets.QPushButton(self.centralwidget)
        self.nic.setGeometry(QtCore.QRect(490, 450, 231, 51))
        self.nic.setObjectName("nic")
        self.nic.clicked.connect(self.show_NewtonInterpolationCalculator)
        self.ni = QtWidgets.QPushButton(self.centralwidget)
        self.ni.setGeometry(QtCore.QRect(90, 570, 231, 51))
        self.ni.setObjectName("ni")
        self.ni.clicked.connect(self.show_NumericalIntegration)
        self.pde = QtWidgets.QPushButton(self.centralwidget)
        self.pde.setGeometry(QtCore.QRect(490, 570, 231, 51))
        self.pde.setObjectName("pdi")
        self.pde.clicked.connect(self.show_PDESolver)
        MainWindow.setCentralWidget(self.centralwidget)
        self.menubar = QtWidgets.QMenuBar(MainWindow)
        self.menubar.setGeometry(QtCore.QRect(0, 0, 800, 26))
        self.menubar.setObjectName("menubar")
        MainWindow.setMenuBar(self.menubar)
        self.statusbar = QtWidgets.QStatusBar(MainWindow)
        self.statusbar.setObjectName("statusbar")
        MainWindow.setStatusBar(self.statusbar)

        self.retranslateUi(MainWindow)
        QtCore.QMetaObject.connectSlotsByName(MainWindow)



    def retranslateUi(self, MainWindow):
        _translate = QtCore.QCoreApplication.translate
        MainWindow.setWindowTitle(_translate("MainWindow", "Scientific Computing Calculator"))
        self.label.setText(_translate("MainWindow", "SCIENTIFIC COMPUTING"))
        self.qes.setText(_translate("MainWindow", "Quadratic Equation Solver"))
        self.let.setText(_translate("MainWindow", "Log, Exp and Trig Calculator"))
        self.sc.setText(_translate("MainWindow", "Standard Calculator"))
        self.ode.setText(_translate("MainWindow", "ODE Solver(Homogeneous Equation)"))
        self.tes.setText(_translate("MainWindow", "Transcendental Equation Solver"))
        self.nic.setText(_translate("MainWindow", "Newton Interpolation "))
        self.ni.setText(_translate("MainWindow", "Numerical Integration"))
        self.pde.setText(_translate("Mainwindow","PDE Solver (Use FEM)"))


    def show_quadratic_solver(self):
        self.quadratic_solver = QuadraticEquationSolver()
        self.quadratic_solver.show()

    def show_basic_calculator(self):
        self.basic_calculator = BasicCalculator()
        self.basic_calculator.show()

    def show_Calculator(self):
        self.Calculator = Calculator()
        self.Calculator.show()

    def show_ODESolverApp(self):
        self.ODESolverApp = ODESolverApp()
        self.ODESolverApp.show()

    def show_TranscendentalEquationSolver(self):
        self.TranscendentalEquationSolver = TranscendentalEquationSolver()
        self.TranscendentalEquationSolver.show()

    def show_NewtonInterpolationCalculator(self):
        self.NewtonInterpolationCalculator = NewtonInterpolationCalculator()
        self.NewtonInterpolationCalculator.show()

    def show_NumericalIntegration(self):
        self.NumericalIntegration = NumericalIntegration()
        self.NumericalIntegration.show()

    def show_PDESolver(self):
        self.PDESolver = PDESolver()
        self.PDESolver.show()

if __name__ == "__main__":
    app = QApplication(sys.argv)
    MainWindow = QtWidgets.QMainWindow()
    ui = Ui_MainWindow()
    ui.setupUi(MainWindow)
    MainWindow.show()
    sys.exit(app.exec_())
```

Fig 32: *The entire code*

Now we are going to convert this python code to an executable file so that it can be a standalone application. To do so we first got to install pyinstall using pip command in command prompt (at its respective PATH)

```
pip install pyinstaller
```

Then we are going to write a code to convert our python file (.py) into executable file (.exe)

```
pyinstaller Scientific_Calculator.py --onefile
```

PyInstaller converts Python scripts (.py files) to standalone executable (.exe) files by analysing the script, calculating its dependencies, and combining them with a Python interpreter to produce a single executable. It employs hooks to manage non-Python libraries and modules. PyInstaller first evaluates the code structure and scans the import statements to determine the script's dependencies. It then creates a bundled package including the script, its dependencies, and the Python interpreter. Finally, it creates the executable file using a specific technique (such as one-folder or one-file mode), allowing the Python script to run on machines without a separate Python installation.

So now we have an executable file that runs the same as shown below:

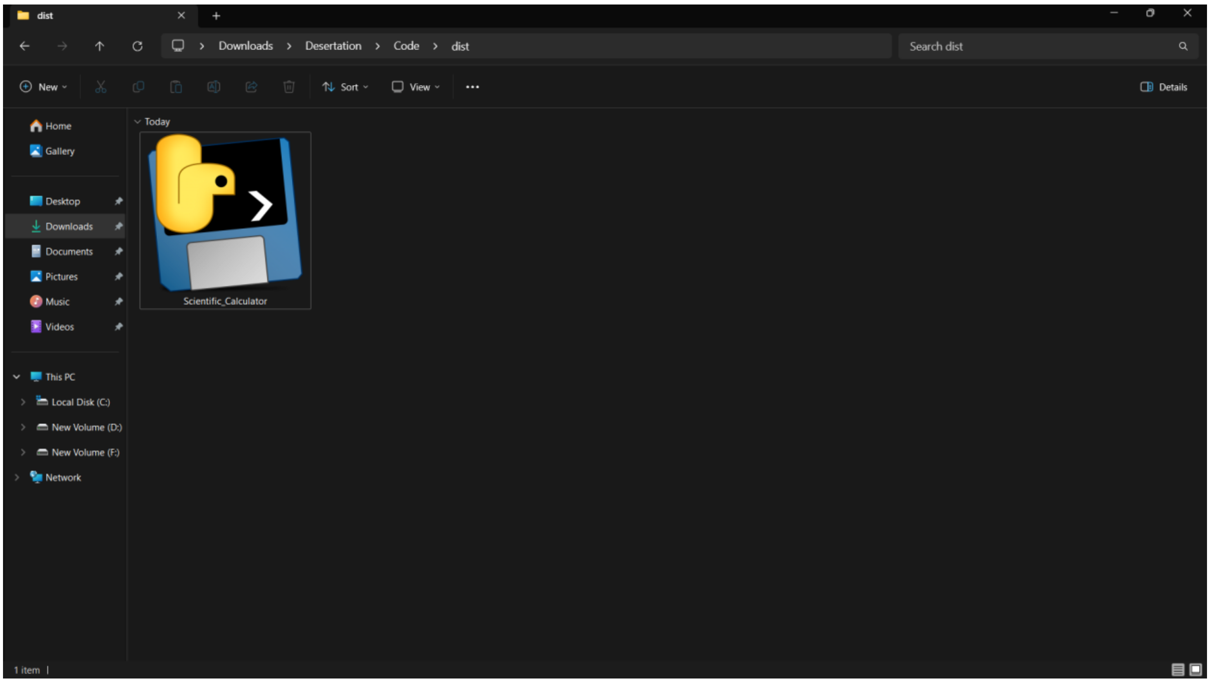


Fig 33: *Executable File*

References:

[1] StackOverflow
[2] GeeksForGeeks
[3] Github