

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, Uttarakhand-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, Uttarakhand-248007, India

PLAGIARISM CERTIFICATE

I, **Anshul Ghildiyal**, hereby 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, Uttarakhand-248007, India

Approved by:

Dr. Ravi Kiran Maddali

Department of Mathematics

School of Engineering

UPES

Dehradun, Uttarakhand-248007, India

PLAGIARISM REPORT SUMMARY

BSc Disser

ORIGINALITY REPORT

8%

SIMILARITY INDEX

6%

INTERNET SOURCES

4%

PUBLICATIONS

2%

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 as 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

1. Chapter 1: Introduction

1.1. Scientific Computing and GUIs: A Vital Interface

1.1.1. Bridging the Gap Between Complexity and Usability

1.1.2. Democratizing Access to Scientific Tools

1.1.3. Accelerating Innovation Through Enhanced Collaborations

1.2. Python's Role in Scientific GUI Development

1.2.1. Introduction to GUI Development with Python

1.2.2. Tkinter: Python's Built-in GUI Library

1.2.3. PyQt: Python's Bindings for Qt

1.2.4. Matplotlib and Plotly: Data Visualization in GUI

1.2.5. Advanced GUI Development with PySide and Kivy

1.2.6. Case Studies

1.3. Research Objective

1.4. Methodological Framework

1.5. Technological Stack

1.6. GUI Design and User- Centered Development

1.7. Topics Covered

2. Chapter 2: Mathematical Functions And Operations (with their Python codes)

2.1. Basic Arithmetic Operations

2.2. Trigonometric Equations

2.2.1. Sine Function

2.2.2. Cosine Function

2.2.3. Tangent Function

2.2.4. Cosecant Function, Secant Function, and Cotangent Function

2.3. Exponential and Logarithmic Functions

2.4. Numerical Integration

2.4.1. Introduction

2.4.2. Theoretical Background

2.4.3. Newton Cote's Formula

2.4.3.1. Trapezoidal Rule

2.4.3.2. Simpson's 1/3rd Rule

2.4.3.3. Simpson's 3/8th Rule

2.4.4. Gaussian Quadrature

2.5. Numerical Solution of Differential Equations

2.5.1. Theoretical Background

2.5.2. Numerical Methods for ODEs

2.5.2.1. Euler's Method

2.5.2.2. Runge-Kutta Method

2.5.3. Numerical Methods for PDEs

2.5.3.1. Finite Difference Methods

2.5.3.2. Finite Element Methods

2.6. Numerical Methods

2.6.1. Bisection Method

2.6.2. Regular Falsi Method

2.6.3. Newton Raphson Method

2.6.4. Secant Method

3. The Prototype

LIST OF TABLES

Table 1: Tools and Technologies

Table 2: All the topics that Scientific Calculation deals with

LIST OF FIGURES

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

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

Fig 3: Basic Arithmetic Operation's GUI Result

Fig 4: GUI of calculator which calculates trigonometric function

Fig 5: Trigonometric Function's GUI Result

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

Fig 7: Exponential and Logarithmic Function's Result

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

Fig 9: Graph representing Trapezoidal Rule

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

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

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

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

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

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

Fig 16: Numerical Integration's GUI Result

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

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

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

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

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

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

Fig 21: GUI of calculator which calculates Ordinary Differential Equation

Fig 22: Ordinary Differential Equation's GUI Result

Fig 23: Basic code for solving PDEs using FDM

Fig 24: Result for basic code of FDM

Fig 25: Basic code for solving ODEs using FEM

Fig 26: Result for basic code of FEM

Fig 27: GUI of calculator which calculates partial differential equation

Fig 28: Partial Differential Equation's GUI Result

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

Fig 30: Numerical Method's GUI Result

Fig 31: The Final Prototype

Fig 32: The entire code

Fig 33: Executable File

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.

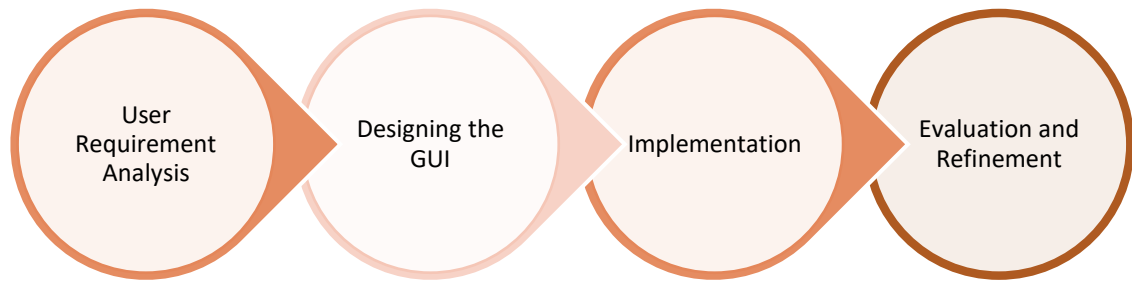


Fig. 1: A flowchart depicting the stages of the dissertation from user requirements analysis, design, implementation, to evaluation and refinement.

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 Siau, 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):

```
299 class BasicCalculator(QWidget):
300     def __init__(self):
301         super().__init__()
302
303         self.setWindowTitle("Basic Calculator")
304         self.setStyleSheet("background-color: #f0f0f0;")
305         self.setGeometry(100, 100, 400, 400)
306
307         # Create the main layout
308         main_layout = QVBoxLayout()
309
310         # Create the display widget
311         self.result_display = QLineEdit()
312         self.result_display.setReadOnly(True)
313         self.result_display.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Preferred)
314         main_layout.addWidget(self.result_display)
315
316         # Create the button grid layout
317         button_grid_layout = QGridLayout()
318
319         # Row 0
320         row0 = self.create_button_row(['', 'C', '='])
321         button_grid_layout.addRow(row0)
322
323         # Row 1
324         row1 = self.create_button_row(['7', '8', '9', '/'])
325         button_grid_layout.addRow(row1)
326
327         # Row 2
328         row2 = self.create_button_row(['4', '5', '6', '*'])
329         button_grid_layout.addRow(row2)
330
331         # Row 3
332         row3 = self.create_button_row(['1', '2', '3', '-'])
333         button_grid_layout.addRow(row3)
334
335         # Row 4
336         row4 = self.create_button_row(['0', '.', '+', '*'])
337         button_grid_layout.addRow(row4)
```

```

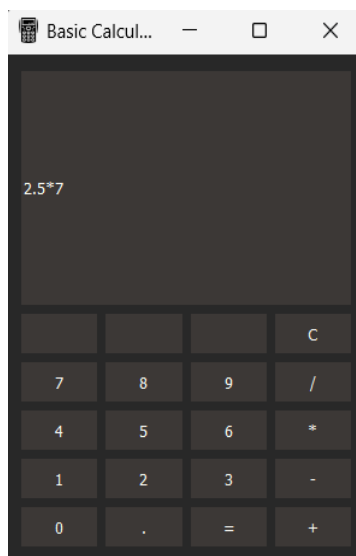
339 # Add the button grid layout to the main layout
340 main_layout.addLayout(button_grid_layout)
341
342 # Set the main layout for the widget
343 self.setLayout(main_layout)
344
345 def create_button_row(self, button_texts):
346     row_layout = QHBoxLayout()
347
348     for button_text in button_texts:
349         button = QPushButton(button_text)
350         button.clicked.connect(self.on_button_click)
351         button.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Preferred)
352         row_layout.addWidget(button)
353
354     return row_layout
355
356 def on_button_click(self):
357     button = self.sender()
358     current_text = self.result_display.text()
359
360     if button.text() == "=":
361         try:
362             result = str(eval(current_text))
363             self.result_display.setText(result)
364         except Exception as e:
365             self.result_display.setText("Error")
366     elif button.text() == "C":
367         self.result_display.clear()
368     else:
369         self.result_display.setText(current_text + button.text())
370
371 def on_clear_click(self):
372     self.result_display.clear()
373

```

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

Result-

Input



Output

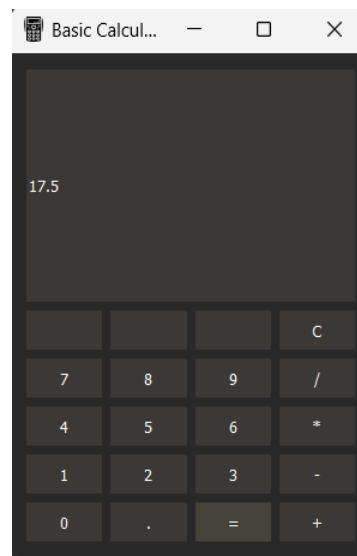


Fig 3: Basic Arithmetic Operation's GUI Result

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):

```
14 class Calculator(QWidget):
15     def __init__(self):
16         super().__init__()
17
18         self.setWindowTitle("Log, exp & trig Calculator")
19         self.setGeometry(100,100,400,400)
20
21         self.expression_lineedit = QLineEdit(self)
22         self.expression_lineedit.setPlaceholderText("Enter expression")
23
24         self.result_lineedit = QLineEdit(self)
25         self.result_lineedit.setReadOnly(True)
26
27         self.create_buttons()
28         self.create_layout()
29
30     def create_buttons(self):
31         buttons_layout = QGridLayout()
32
33         buttons = [
34             ('7', 0, 0),
35             ('8', 0, 1),
36             ('9', 0, 2),
37             ('4', 1, 0),
38             ('5', 1, 1),
39             ('6', 1, 2),
40             ('1', 2, 0),
41             ('2', 2, 1),
42             ('3', 2, 2),
43             ('0', 3, 0),
44             ('.', 3, 1),
45             ('+', 3, 2),
46             ('-', 3, 3),
47             ('*', 4, 0),
48             ('/', 4, 1),
49             ('^', 4, 2),
50             ('ln', 4, 3),
51             ('exp', 4, 4),
52             ('sin', 5, 0),
53             ('cos', 5, 1),
54             ('tan', 5, 2),
55             ('(', 5, 3),
56             (')', 5, 4),
57             ('= ', 6, 0),
58             ('C', 6, 1),
59             ('AC', 6, 2),
60         ]
61
62         for btn_text, row, col in buttons:
63             button = QPushButton(btn_text, self)
64             button.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Expanding)
65             button.clicked.connect(lambda _, text=btn_text: self.button_pressed(text))
66             buttons_layout.addWidget(button, row, col)
67
68         self.buttons_layout = buttons_layout
69
70     def create_layout(self):
71         main_layout = QVBoxLayout()
72
73         main_layout.addWidget(self.expression_lineedit)
74         main_layout.addWidget(self.result_lineedit)
75         main_layout.addLayout(self.buttons_layout)
76
77         self.setLayout(main_layout)
78
79         # Set dark theme
80         self.setStyleSheet("""
81             QWidget {
82                 background-color: #282828;
83                 color: #f8f8f2;
84             }
85             QLineEdit {
86                 background-color: #3c3836;
87                 color: #f8f8f2;
88                 border: none;
89             }
90             QPushButton {
91                 background-color: #3c3836;
92                 color: #f8f8f2;
93                 border: none;
94             }
95             QPushButton:hover {
96                 background-color: #45413b;
97             }
98         """)
99
100     def button_pressed(self, text):
101         current_text = self.expression_lineedit.text()
102
103         if text == '=':
104             try:
105                 result = eval(current_text)
106                 self.result_lineedit.setText(str(result))
107             except Exception as e:
108                 self.result_lineedit.setText("Error")
109         elif text == 'C':
110             new_text = current_text[:-1]
111             self.expression_lineedit.setText(new_text)
112         elif text == 'AC':
113             self.expression_lineedit.clear()
114             self.result_lineedit.clear()
115         else:
116             new_text = current_text + text
117             self.expression_lineedit.setText(new_text)
```

Fig 4: GUI of calculator which calculates trigonometric function

Result –

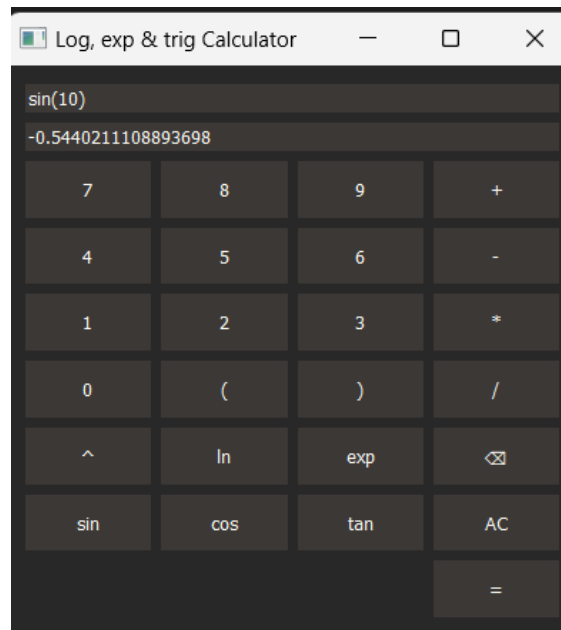


Fig 5: Trigonometric Function's GUI Result

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 $f(x) = a \cdot b^x$, 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):

```
14 class Calculator(QWidget):
15     def __init__(self):
16         super().__init__()
17
18         self.setWindowTitle("Log, exp & trig Calculator")
19         self.setGeometry(100,100,400,400)
20
21         self.expression_lineedit = QLineEdit(self)
22         self.expression_lineedit.setPlaceholderText("Enter expression")
23
24         self.result_lineedit = QLineEdit(self)
25         self.result_lineedit.setReadOnly(True)
26
27         self.create_buttons()
28         self.create_layout()
29
30     def create_buttons(self):
31         buttons_layout = QGridLayout()
32
33         buttons = [
34             ('7', 0, 0),
35             ('8', 0, 1),
36             ('9', 0, 2),
37             ('4', 1, 0),
38             ('5', 1, 1),
39             ('6', 1, 2),
40             ('1', 2, 0),
41             ('2', 2, 1),
42             ('3', 2, 2),
43             ('0', 3, 0),
44             ('+', 3, 1),
45             ('-', 3, 2),
46             ('*', 3, 3),
47             ('/', 3, 4),
48             ('^', 4, 0),
49             ('ln', 4, 1),
50             ('exp', 4, 2),
51             ('sin', 4, 3),
52             ('cos', 4, 4),
53             ('tan', 5, 0),
54             ('(', 5, 1),
55             (')', 5, 2),
56             ('= ', 5, 3),
57             ('CE', 5, 4),
58             ('AC', 6, 0)
59         ]
60
61         for btn_text, row, col in buttons:
62             button = QPushButton(btn_text, self)
63             button.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Expanding)
64             button.clicked.connect(lambda _, text=btn_text: self.button_pressed(text))
65             buttons_layout.addWidget(button, row, col)
66
67         self.buttons_layout = buttons_layout
68
69     def create_layout(self):
70         main_layout = QVBoxLayout()
71
72         main_layout.addWidget(self.expression_lineedit)
73         main_layout.addWidget(self.result_lineedit)
74         main_layout.addLayout(self.buttons_layout)
75
76         self.setLayout(main_layout)
77
78         # Set dark theme
79         self.setStyleSheet("""
80         QWidget {
81             background-color: #282828;
82             color: #f8f8f2;
83         }
84         QLineEdit {
85             background-color: #3c3836;
86             color: #f8f8f2;
87             border: none;
88         }
89         QPushButton {
90             background-color: #3c3836;
91             color: #f8f8f2;
92             border: none;
93         }
94         QPushButton:hover {
95             background-color: #45433b;
96         }
97         """)
98
99     def button_pressed(self, text):
100         current_text = self.expression_lineedit.text()
101
102         if text == '=':
103             try:
104                 result = eval(current_text)
105                 self.result_lineedit.setText(str(result))
106             except Exception as e:
107                 self.result_lineedit.setText("Error")
108
109         elif text == 'CE':
110             new_text = current_text[:-1]
111             self.expression_lineedit.setText(new_text)
112
113         elif text == 'AC':
114             self.expression_lineedit.clear()
115             self.result_lineedit.clear()
116
117         else:
118             new_text = current_text + text
119             self.expression_lineedit.setText(new_text)
```

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

Result-

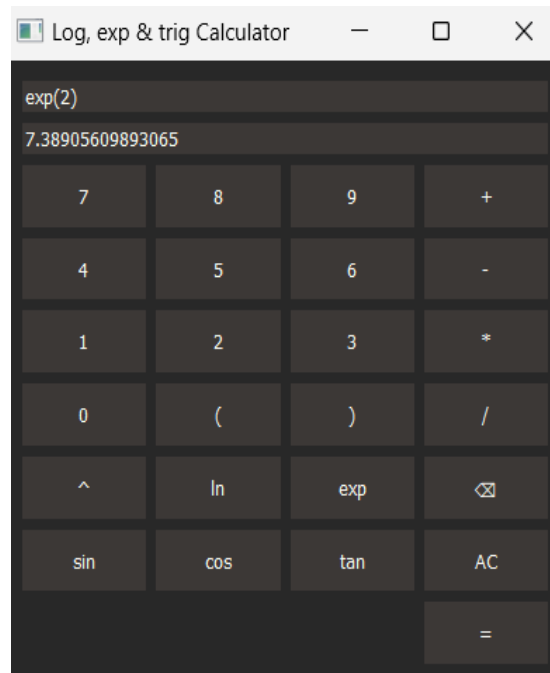


Fig 7: Exponential and Logarithmic Function's Result

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

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

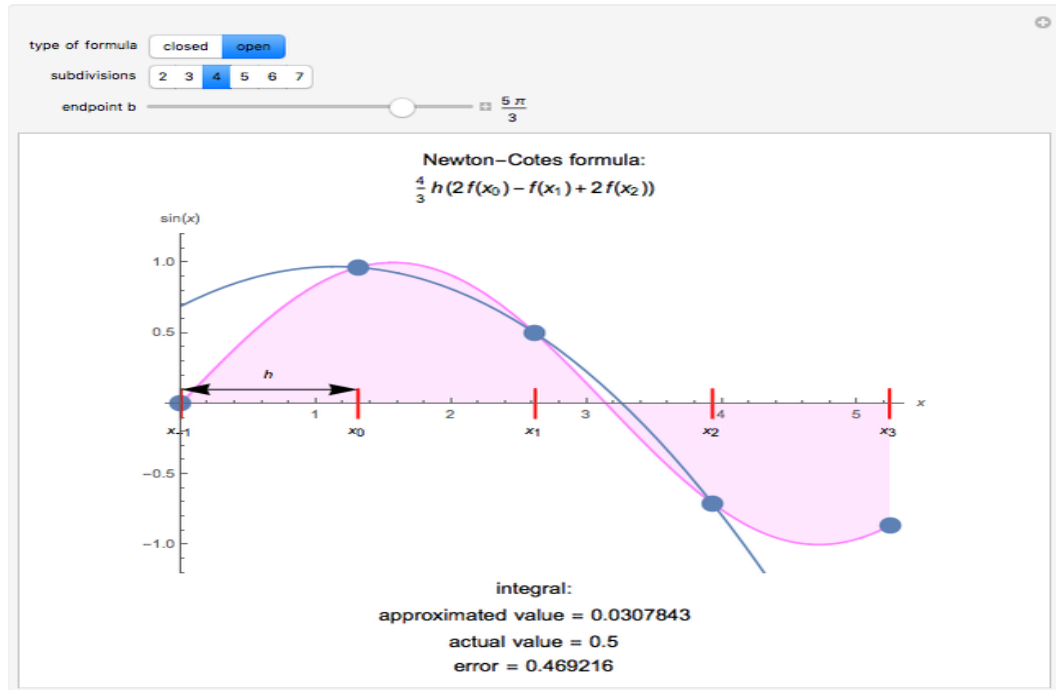


Fig 8: Shows the graphical representation of Newton – Cotes Quadrature

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

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

It is referred to as the *trapezoidal rule*

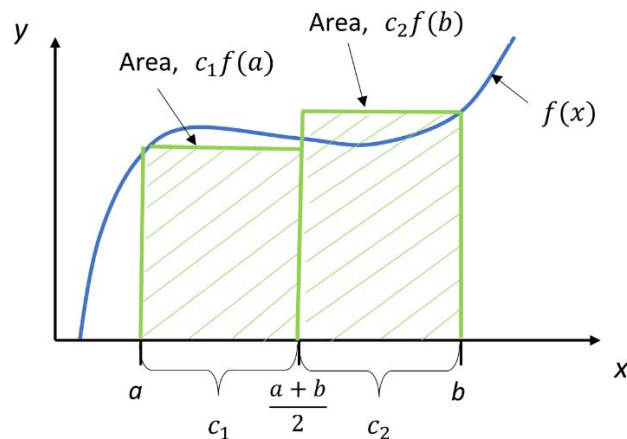


Fig 9: Graph representing Trapezoidal Rule

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:

```

1 def trapezoidal_rule(a, b, n, f):
2     # Calculate the width of each interval
3     h = (b - a) / n
4     # Calculate the sum of the function values at the endpoints and the midpoints of each interval
5     sum = 0.5 * (f(a) + f(b))
6     for i in range(1, n):
7         x = a + i * h
8         sum += f(x)
9     # Multiply the sum by the width of each interval and return the result
10    return sum * h
11
12 # Define the function to integrate
13 def f(x):
14     return x**2
15
16 # Calculate the numerical integration
17 a = 0
18 b = 2
19 n = 100
20 integral = trapezoidal_rule(a, b, n, f)
21 print(f"The approximate value of the integral is by trapezoidal rule is {integral}.")

```

PROBLEMS OUTPUT TERMINAL PORTS SEARCH ERROR LIGHTTRUN

> TERMINAL

```

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

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

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

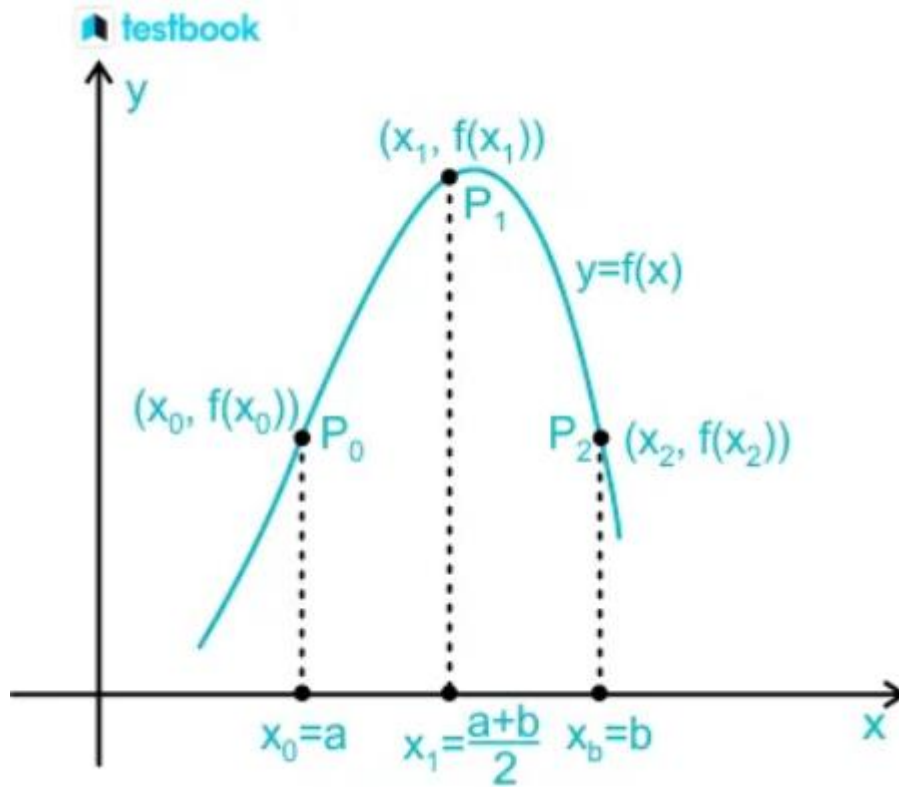


Fig 11: Graph representing Simpson's 1/3rd Rule

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:

```

1  def simpson_rule(a, b, n, f):
2      # Check if n is even
3      if n % 2 != 0:
4          raise ValueError("n must be even")
5      # Calculate the width of each interval
6      h = (b - a) / n
7      # Calculate the sum of the function values at the endpoints and the midpoints of each interval
8      sum = f(a) + f(b)
9      for i in range(1, n // 2):
10         x = a + 2 * i * h
11         sum += 2 * f(x)
12     for i in range(1, n // 2 + 1):
13         x = a + (2 * i - 1) * h
14         sum += 4 * f(x)
15     # Multiply the sum by the width of each interval and return the result
16     return sum * h / 3
17 # Define the function to integrate
18 def f(x):
19     return x**2
20 # Calculate the numerical integration
21 a = 0
22 b = 2
23 n = 100
24 integral = simpson_rule(a, b, n, f)
25 print(f"The approximate value of the integral by Simpson's 1/3rd is {integral}.")

```

PROBLEMS OUTPUT TERMINAL PORTS SEARCH ERROR LIGHTRUN

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

$$\int_{x_0}^{x_n} y dx = \frac{3h}{8} (y_0 + 3y_1 + 3y_2 + 2y_3 + \cdots + 2y_{n-3} + 3y_{n-2} + 3y_{n-1} + y_n)$$

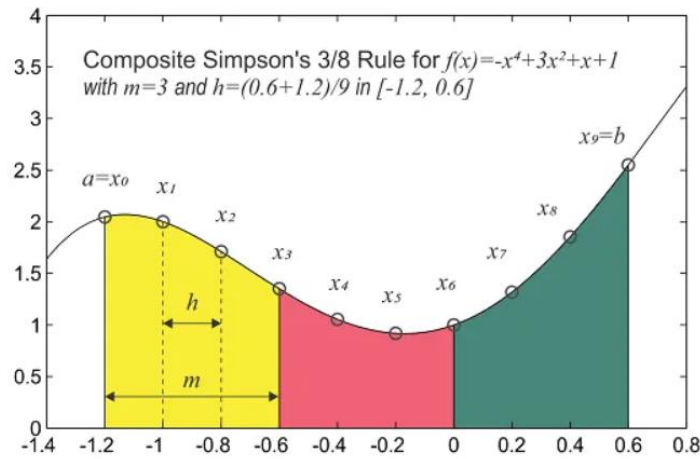


Fig 13: Graph representing Simpson's 3/8th Rule

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:

```

1 def simpson_38_rule(a, b, n, f):
2     # Check if n is a multiple of 3
3     if n % 3 != 0:
4         raise ValueError("n must be a multiple of 3")
5     # Calculate the width of each interval
6     h = (b - a) / n
7     # Calculate the sum of the function values at the endpoints and the midpoints of each interval
8     sum = f(a) + f(b)
9     for i in range(1, n // 3):
10        x = a + 3 * i * h
11        sum += 3 * f(x)
12    for i in range(1, n // 3 + 1):
13        x = a + (2 * i - 1) * h
14        sum += 2 * f(x)
15    # Multiply the sum by the width of each interval and return the result
16    return sum * h * 3 / 8
17
18 # Define the function to integrate
19 def f(x):
20     return x**2
21
22 # Calculate the numerical integration
23 a = 0
24 b = 2
25 n = 102
26 integral = simpson_38_rule(a, b, n, f)
27 print(f"The approximate value of the integral by Simpson's 3/8th is {integral}.")

```

PROBLEMS OUTPUT TERMINAL PORTS SEARCH ERROR LIGHTRUN

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 $\frac{3}{8}$ rule] (using PyQt package):

```
1 import sys
2 from PyQt5.QtWidgets import QApplication, QWidget, QVBoxLayout, QGridLayout, QLabel, QLineEdit, QPushButton
3 import math
4 from sympy import symbols
5
6
7 def simpsons_3_8_rule(f, a, b, n):
8     """
9     Numerical integration using Simpson's 3/8th rule.
10    f: Function to integrate.
11    a: Lower limit of integration.
12    b: Upper limit of integration.
13    n: Number of sub-intervals.
14    """
15    h = (b - a) / n
16    result = f(a) + f(b)
17    for i in range(1, n):
18        if i % 3 == 0:
19            result += 2 * f(a + i * h)
20        else:
21            result += 3 * f(a + i * h)
22    result *= h / 8
23    return result
24
25
26 class IntegrationApp(QWidget):
27     def __init__(self):
28         super().__init__()
29         self.setWindowTitle("Numerical Integration using Simpson's 3/8th Rule")
30         self.init_ui()
31
32     def init_ui(self):
33         layout = QVBoxLayout()
34
35         self.equation_label = QLabel("Enter the equation (use 'x' as variable):")
36         self.equation_input = QLineEdit()
37
38         self.lower_limit_label = QLabel("Lower limit:")
39         self.lower_limit_input = QLineEdit()
40         self.lower_limit_input.setReadOnly(True)
41
42         self.upper_limit_label = QLabel("Upper limit:")
43         self.upper_limit_input = QLineEdit()
44         self.upper_limit_input.setReadOnly(True)
45
46         self.subintervals_label = QLabel("Number of subintervals:")
47         self.subintervals_input = QLineEdit()
48         self.subintervals_input.setReadOnly(True)
49
50         self.result_label = QLabel("Result:")
51         self.result_display = QLabel("")
52
53         self.calculate_button = QPushButton("Calculate")
54         self.calculate_button.clicked.connect(self.calculate_integration)
55
56         self.clear_button = QPushButton("Clear")
57         self.clear_button.clicked.connect(self.clear_inputs)
58
```

```

58 self.backspace_button = QPushButton("Backspace")
59 self.backspace_button.clicked.connect(self.backspace_input)
60
61 self.edit_limit_button = QPushButton("Edit Limits")
62 self.edit_limit_button.clicked.connect(self.enable_limit_editing)
63
64 self.grid_layout = QGridLayout()
65 buttons = ['7', '8', '9', '+', '4', '5', '6', '-', '1', '2', '3', '*', '0', '.', 'x', '/'],
66           ['x^2', '(', ')', 'log', 'exp']
67 positions = [(i, j) for i in range(5) for j in range(4)]
68
69 for position, button in zip(positions, buttons):
70     btn = QPushButton(button)
71     if button == 'x^2':
72         btn.clicked.connect(self.insert_x_squared)
73     else:
74         btn.clicked.connect(lambda _, text=button: self.add_to_equation_input(text))
75     self.grid_layout.addWidget(btn, *position)
76
77 grid_widget = QWidget()
78 grid_widget.setLayout(self.grid_layout)
79
80 layout.addWidget(self.equation_label)
81 layout.addWidget(self.equation_input)
82 layout.addWidget(self.lower_limit_label)
83 layout.addWidget(self.lower_limit_input)
84 layout.addWidget(self.upper_limit_label)
85 layout.addWidget(self.upper_limit_input)
86 layout.addWidget(self.subintervals_label)
87 layout.addWidget(self.subintervals_input)
88 layout.addWidget(grid_widget) # Add the grid layout widget
89 layout.addWidget(self.calculate_button)
90 layout.addWidget(self.clear_button)
91 layout.addWidget(self.backspace_button)
92 layout.addWidget(self.edit_limit_button)
93 layout.addWidget(self.result_label)
94 layout.addWidget(self.result_display)
95
96 self.setLayout(layout)
97
98 def add_to_equation_input(self, text):
99     current_text = self.equation_input.text()
100
101     # Handle arithmetic operations
102     if text in ['+', '-', '*', '/']:
103         # Add space before and after the operator
104         text = ' ' + text + ' '
105
106     self.equation_input.setText(current_text + text)
107
108 def insert_x_squared(self):
109     current_text = self.equation_input.text()
110     x = symbols('x')
111     x_squared = x**2
112     self.equation_input.setText(current_text + str(x_squared))
113
114 def enable_limit_editing(self):
115     self.lower_limit_input.setReadOnly(False)
116     self.upper_limit_input.setReadOnly(False)
117     self.subintervals_input.setReadOnly(False)
118
119 def clear_inputs(self):
120     self.equation_input.clear()
121     self.lower_limit_input.clear()
122     self.upper_limit_input.clear()
123     self.subintervals_input.clear()
124     self.result_display.clear()
125
126 def backspace_input(self):
127     self.equation_input.backspace()
128
129 def calculate_integration(self):
130     try:
131         equation = self.equation_input.text()
132         lower_limit = float(self.lower_limit_input.text())
133         upper_limit = float(self.upper_limit_input.text())
134         n = int(self.subintervals_input.text())
135
136         # Define the function to integrate
137         def f(x):
138             return eval(equation)
139
140         # Handle 'log' and 'exp' functions
141         equation = equation.replace('log', 'math.log')
142         equation = equation.replace('exp', 'math.exp')
143
144         result = simpsons_3_8_rule(f, lower_limit, upper_limit, n)
145         self.result_display.setText(str(result))
146
147     except Exception as e:
148         self.result_display.setText("Error: " + str(e))
149
150 if __name__ == "__main__":
151     app = QApplication(sys.argv)
152     window = IntegrationApp()
153     window.show()
154     sys.exit(app.exec_())

```

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

Result –

Numerical Integration using Sim...

Enter the equation (use 'x' as variable):
 $x^2 + 2$

Lower limit:
5

Upper limit:
6

Number of subintervals:
5

7 8 9 +
4 5 6 -
1 2 3 *
0 . x /
 x^2 () log

Calculate
Clear
Backspace
Edit Limits

Result:
31.44300000000001

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 X
C:\Users\HP> HP> Downloads> Desertation> Code> Euler's method.py > ...
Click here to ask Blackbox to help you code faster
1 import numpy as np
2 import matplotlib.pyplot as plt
3 def euler_method(f, x0, y0, xmax, h):
4     x_values = np.arange(x0, xmax+h, h)
5     y_values = [y0]
6     for i in range(1, len(x_values)):
7         y_next = y_values[-1] + h * f(x_values[i-1], y_values[-1])
8         y_values.append(y_next)
9     return x_values, y_values
10 # Example usage:
11 def f(x, y):
12     return x + y
13 x0 = 0 # Initial x
14 y0 = 1 # Initial y
15 xmax = 2 # Maximum value of x
16 h = 0.1 # Step size
17 x_values, y_values = euler_method(f, x0, y0, xmax, h)
18 # Print values
19 print("x values:", x_values)
20 print("y values:", y_values)
21 # Plotting
22 plt.plot(x_values, y_values, label="Euler's method")
23 plt.scatter(x_values, y_values, color='red', label="Computed Points")
24 plt.xlabel('x')
25 plt.ylabel('y')
26 plt.title('Solution of dy/dx = x + y using Euler\'s method')
27 plt.legend()
28 plt.grid(True)
29 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, 4.076856753442001, 4.6045424287
86201, 5.1949667166482, 5.854496338831303, 6.589345972714433, 7.408940569985877, 8.319834626984465, 9.331818089682912, 10.454999898651202]

```

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

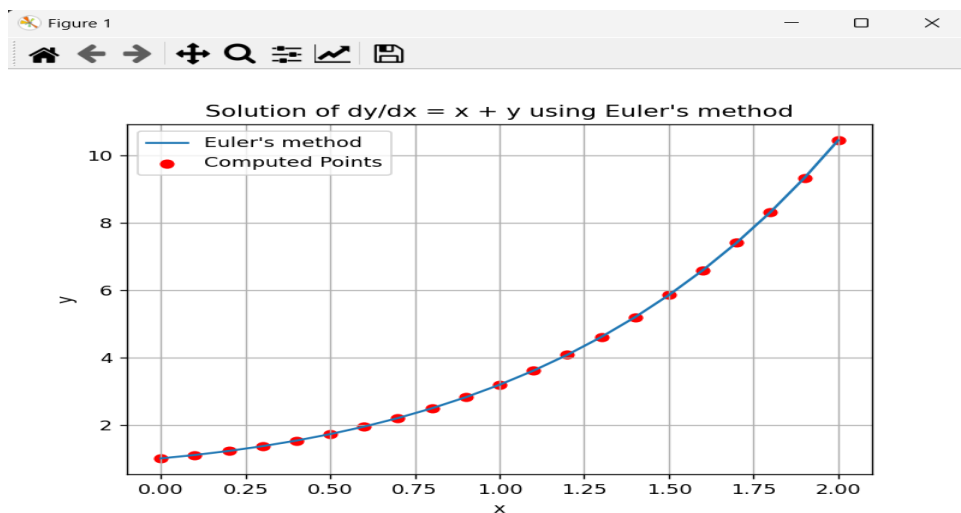
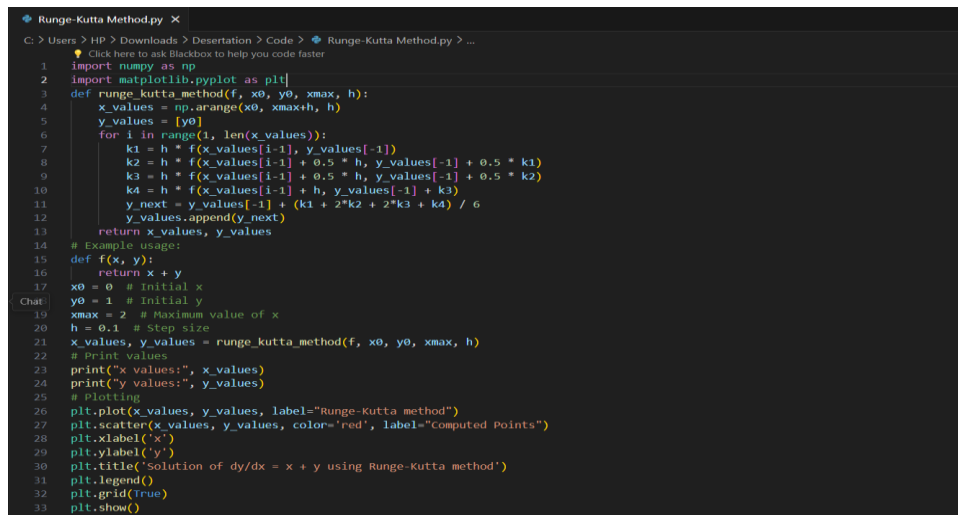


Fig 18(b): Graph for the basic code of Euler's method

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.

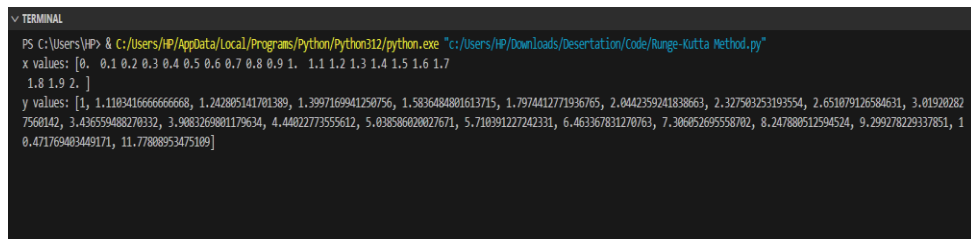
5.1.2.1. Code:



```
Runge-Kutta Method.py X
C:\Users\HP> HP > Downloads > Desertation > Code > Runge-Kutta Method.py > ...
Click here to ask Blackbox to help you code faster
1 import numpy as np
2 import matplotlib.pyplot as plt
3 def runge_kutta_method(f, x0, y0, xmax, h):
4     x_values = np.arange(x0, xmax+h, h)
5     y_values = [y0]
6     for i in range(1, len(x_values)):
7         k1 = h * f(x_values[i-1], y_values[-1])
8         k2 = h * f(x_values[i-1] + 0.5 * h, y_values[-1] + 0.5 * k1)
9         k3 = h * f(x_values[i-1] + 0.5 * h, y_values[-1] + 0.5 * k2)
10        k4 = h * f(x_values[i-1] + h, y_values[-1] + k3)
11        y_next = y_values[-1] + (k1 + 2*k2 + 2*k3 + k4) / 6
12        y_values.append(y_next)
13    return x_values, y_values
14 # Example usage:
15 def f(x, y):
16     return x + y
17 x0 = 0 # Initial x
18 y0 = 1 # Initial y
19 xmax = 2 # Maximum value of x
20 h = 0.1 # step size
21 x_values, y_values = runge_kutta_method(f, x0, y0, xmax, h)
22 # Print values
23 print("x values:", x_values)
24 print("y values:", y_values)
25 # Plotting
26 plt.plot(x_values, y_values, label="Runge-Kutta method")
27 plt.scatter(x_values, y_values, color='red', label="computed Points")
28 plt.xlabel('x')
29 plt.ylabel('y')
30 plt.title('Solution of dy/dx = x + y using Runge-Kutta method')
31 plt.legend()
32 plt.grid(True)
33 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. 1.1183416666666668, 1.242805141781389, 1.3997169941250756, 1.5836484801613715, 1.7974412771936765, 2.0442359241838663, 2.327583253193554, 2.651079126584631, 3.01920282
7568142, 3.436559488270332, 3.9083269801179634, 4.44022773555612, 5.038586020027671, 5.710391227242331, 6.463367831270763, 7.306052695558702, 8.247880512594524, 9.299278229337851, 1
0.47176948349171, 11.77808953475109]
```

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

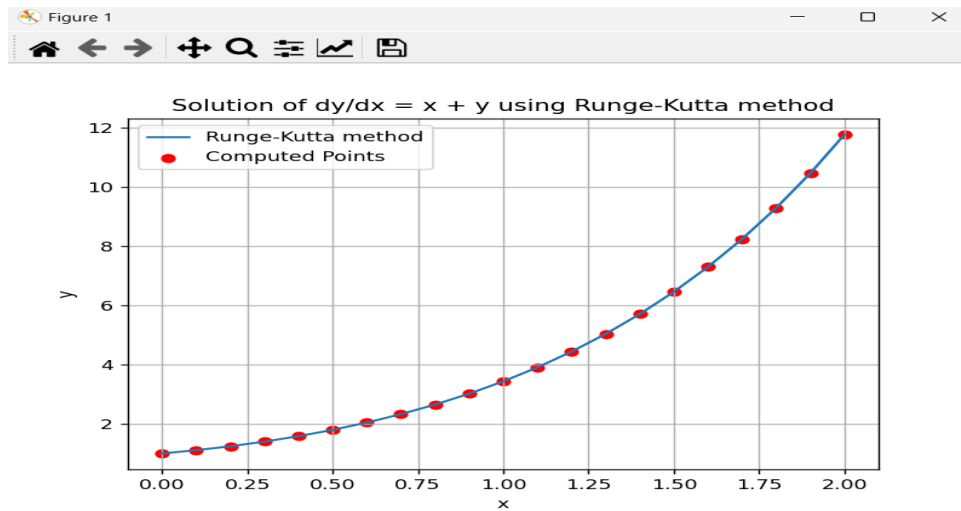


Fig 20(b): Graph for the basic code of Runge-Kutta Method

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

```

596 class ODESolverApp(QDialog):
597     def __init__(self):
598         super().__init__()
599
600     def initUI(self):
601
602         # Order selection
603         self.order_label = QLabel('Select the order of the ODE', self)
604         self.order_combobox = QComboBox(self)
605         self.order_combobox.addItems(['1', '2'])
606         self.order_combobox.setStyleSheet("background-color: #34495E; color: #ECF0F1;")
607
608         # Initial condition input
609         self.initial_condition_label = QLabel('Enter the initial condition', self)
610         self.initial_condition_edit = QLineEdit(self)
611         self.initial_condition_edit.setStyleSheet("background-color: #34495E; color: #ECF0F1;")
612
613         # Coefficients input
614         self.coefficient_labels = ['a(y\'\'\'', 'b(y\'\'', 'c(y)']
615         self.coefficient_lineedit1 = QLineEdit(self)
616         self.coefficient_lineedit2 = QLineEdit(self)
617         self.coefficient_lineedit3 = QLineEdit(self)
618         self.coefficient_lineedit1.setStyleSheet("background-color: #34495E; color: #ECF0F1;")
619         self.coefficient_lineedit2.setStyleSheet("background-color: #34495E; color: #ECF0F1;")
620         self.coefficient_lineedit3.setStyleSheet("background-color: #34495E; color: #ECF0F1;")
621
622         # Solve button
623         self.solve_button = QPushButton('Solve ODE', self)
624         self.solve_button.setStyleSheet("background-color: #34495E; color: #ECF0F1;")
625         self.solve_button.clicked.connect(self.solveODE)
626
627         # Cool icon for the button
628         cool_icon = QIcon("C:/Users/HP/Downloads/thesartationdownload.png")
629         self.solve_button.setIcon(cool_icon)
630         self.solve_button.setIconSize(QSize(24, 24))
631
632         # Result labels
633         self.result_label = QLabel(self)
634         self.result_label.setTextInteractionFlags(Qt.TextSelectableByMouse)
635         self.result_label.setStyleSheet("background-color: #34495E; color: #ECF0F1;")
636
637         # Plot area
638         self.figure, self.ax = plt.subplots()
639         self.canvas = FigureCanvas(self.figure)
640
641         # Layout
642         input_layout = QHBoxLayout()
643         input_layout.addWidget(self.order_label)
644         input_layout.addWidget(self.order_combobox)
645         input_layout.addWidget(self.initial_condition_label)
646         input_layout.addWidget(self.initial_condition_edit)
647         for label in self.coefficient_labels:
648             input_layout.addWidget(QLabel(f'Enter the coefficient \'{label}\''))
649             input_layout.addWidget(self.coefficient_lineedit1)
650
651         button_layout = QHBoxLayout()
652         button_layout.addWidget(self.solve_button)
653
654         self.setLayout(input_layout)
655         self.setLayout(button_layout)
656
657         self.ax.set_xlabel('time')
658         self.ax.set_ylabel('y(t)')
659
660         # Display roots and general solution
661         characteristic_roots = np.roots([coefficients['a(y\'\'\'', coefficients['b(y\'\'', coefficients['c(y)']])
662         roots_str = ', '.join([f'round(roots.real, 2) \'+\' if roots.imag == 0 else \'+\' + str(abs(round(roots.imag, 2)))\' for root in characteristic_roots])
663
664         if np.all(np.isreal(characteristic_roots)):
665             root_type = "Real Roots"
666         else:
667             root_type = "Complex Roots"
668
669         general_solution = f"({root_type})roots: {roots_str})u"
670         general_solution = f"General solution: y(t) = e^{(characteristic_roots.real)*t} (C1 * cos(abs(characteristic_roots.imag)*t) + C2 * sin(abs(characteristic_roots.imag)*t))"
671
672         if order == 2:
673             general_solution += f"General solution: y(t) = C1 * e^{(characteristic_roots[0]) * t} + C2 * e^{(characteristic_roots[1]) * t}"
674         else:
675             general_solution += f"First-order ODE solution: y(t) = C1 * e^{(-(coefficients['b(y\'\'') / coefficients['a(y\'\'')]) * t)"
676
677         self.result_label.setText(general_solution)
678
679         # Draw the new plot
680         self.ax.legend(loc='best')
681         self.ax.set_title("Solution of (coefficients['a(y\'\'')])y\'\'\' + (coefficients['b(y\'\'')])y\' + (coefficients['c(y)'])y = 0")
682         self.ax.grid()
683         self.canvas.draw()

```

Fig 21: GUI of calculator which calculates Ordinary Differential Equation

Result-

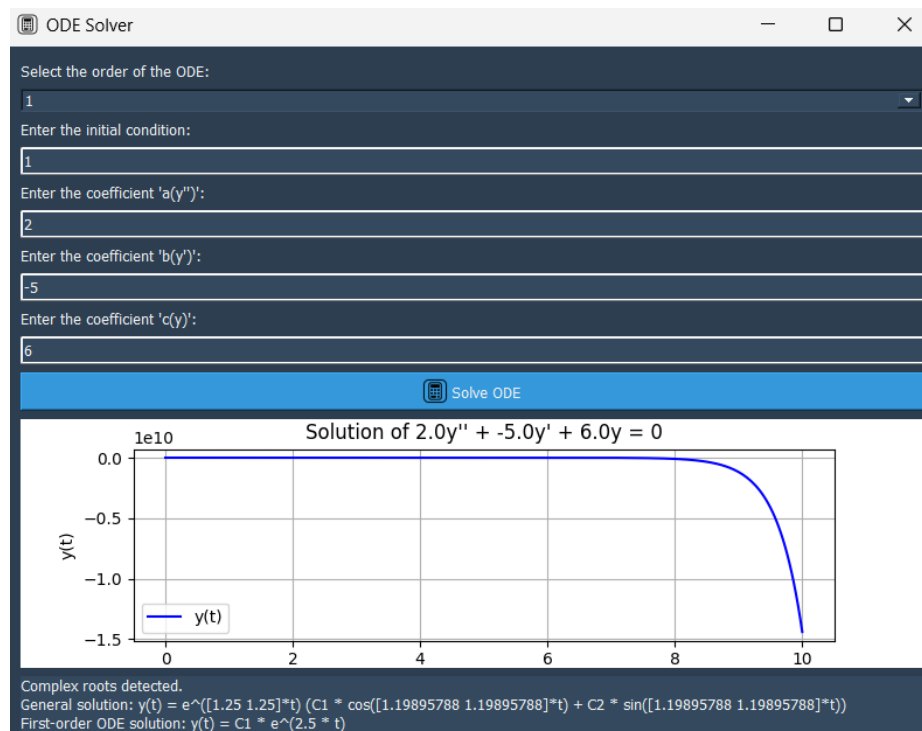


Fig 22: Ordinary Differential Equation's GUI Result

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:

```

1 # Click here to ask Question to help you code faster
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 # Parameters
6 L = 1.0 # Length of the rod
7 T = 0.1 # Total time
8 Nx = 100 # Number of spatial grid points
9 Nt = 1000 # Number of time steps
10 alpha = 0.01 # Thermal diffusivity
11 # Grid spacing
12 dx = L / (Nx - 1)
13 dt = T / Nt
14
15 # Initial condition
16 def initial_condition(x):
17     return np.sin(np.pi * x)
18
19 # Boundary conditions
20 def boundary_condition_left(t):
21     return 0.0
22
23 def boundary_condition_right(t):
24     return 0.0
25
26 # Finite Difference Method
27 def solve_heat_equation(Nx, Nt, alpha, dx, dt):
28     # Initialize solution matrix
29     u = np.zeros((Nt+1, Nx))
30
31     # Initial condition
32     u[0, :] = initial_condition(np.linspace(0, L, Nx))
33
34     # Boundary conditions
35     u[:, 0] = boundary_condition_left(np.linspace(0, T, Nt+1))
36     u[:, -1] = boundary_condition_right(np.linspace(0, T, Nt+1))
37
38     # Finite Difference iteration
39     for n in range(Nt):
40         for i in range(1, Nx-1):
41             u[n+1, i] = u[n, i] + alpha * dt / dx**2 * (u[n, i+1] - 2*u[n, i] + u[n, i-1])
42
43     return u
44
45 # Solve the heat equation
46 u = solve_heat_equation(Nx, Nt, alpha, dx, dt)
47
48 # Plotting
49 x_values = np.linspace(0, L, Nx)
50 t_values = np.linspace(0, T, Nt+1)
51 fig = plt.figure(figsize=(10, 6))
52 ax = fig.add_subplot(111, projection='3d')
53 ax.plot_surface(t_values, x_values, u, cmap='viridis')
54 ax.set_xlabel('Position')
55 ax.set_ylabel('Time')
56 ax.set_zlabel('Temperature')
57 ax.set_title('Solution of 1D Heat Equation using Finite Difference Method')
58 plt.show()

```

Fig 23: Basic code for solving PDEs using FDM

Result –

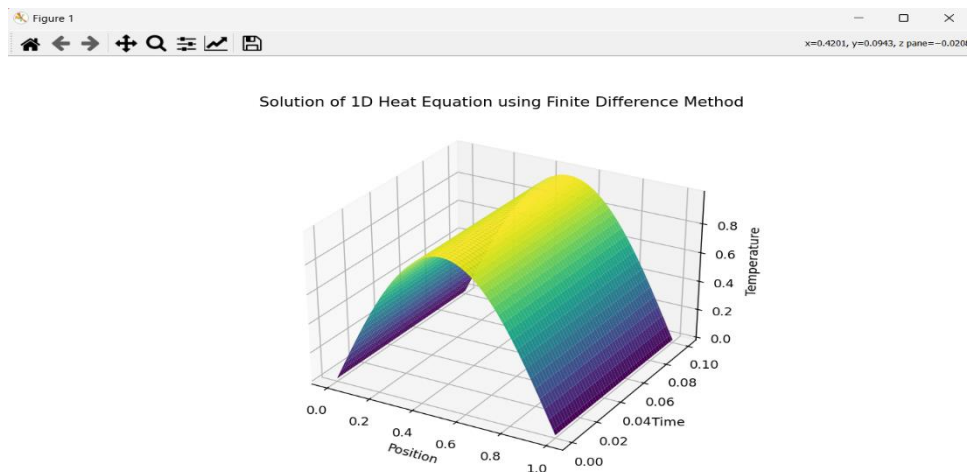


Fig 24: Result for basic code of FDM

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:

```
FEMpy
C:\Users\HP>Downloads>Desertation>Code>FEMpy>...
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from scipy.sparse import csr_matrix, linalg
4
5 # Parameters
6 L = 1.0 # Length of the rod
7 T = 0.1 # Total time
8 Nx = 100 # Number of spatial grid points
9 Nt = 100 # Number of time steps
10 alpha = 0.01 # Thermal diffusivity
11
12 # Grid spacing
13 dx = L / (Nx - 1)
14 dt = T / Nt
15
16 # Generate mesh
17 x_values = np.linspace(0, L, Nx)
18
19 # Assemble stiffness matrix
20 A = np.zeros((Nx, Nx))
21 for i in range(1, Nx - 1):
22     A[i, i] = 2 / dx
23     A[i, i - 1] = -1 / dx
24     A[i, i + 1] = -1 / dx
25
26 # Apply boundary conditions
27 A[0, 0] = 1 / dx
28 A[-1, -1] = 1 / dx
29
30 # Time integration
31 u = np.sin(np.pi * x_values) # Initial condition
32 for _ in range(Nt):
33     u = u + dt * alpha * np.dot(A, u)
34
35 # Plotting
36 plt.plot(x_values, u)
37 plt.xlabel('Position')
38 plt.ylabel('Temperature')
39 plt.title('Solution of 1D Heat Equation using Finite Element Method')
40 plt.grid(True)
41 plt.show()
```

Fig 25: Basic code for solving ODEs using FEM

Result-

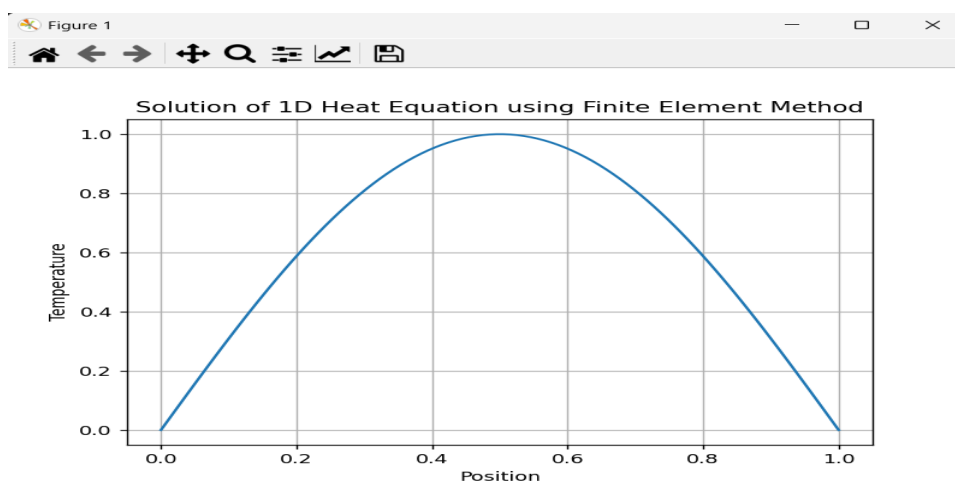


Fig 26: Result for basic code of FEM

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

```

1 import sys
2 import numpy as np
3 import matplotlib.pyplot as plt
4 from PyQt5.QtWidgets import *
5 from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as FigureCanvas
6
7 class FEMSolver:
8     def __init__(self, equation_type, solver_type):
9         self.equation_type = equation_type
10        self.solver_type = solver_type
11
12    def solve(self, equation, **kwargs):
13        # Implement Finite Element Method solver based on the equation and solver type
14        # Placeholder implementation
15        x = np.linspace(0, 1, 100)
16        y = np.sin(np.pi * x)
17        return x, y
18
19 class MainWindow(QMainWindow):
20     def __init__(self):
21         super().__init__()
22         self.setWindowTitle("FEM Solver")
23         self.setGeometry(100, 100, 800, 600)
24
25         self.central_widget = QWidget()
26         self.setCentralWidget(self.central_widget)
27
28         self.layout = QHBoxLayout()
29         self.central_widget.setLayout(self.layout)
30
31         self.equation_label = QLabel("Equation Type:")
32         self.layout.addWidget(self.equation_label)
33         self.equation_combo = QComboBox()
34         self.equation_combo.addItems(["Custom Equation", "Heat Equation", "Wave Equation"])
35         self.layout.addWidget(self.equation_combo)
36         self.equation_combo.currentIndexChanged.connect(self.on_equation_combo_changed)
37         self.layout.addWidget(self.equation_combo)
38
39         self.equation_text_edit = QTextEdit()
40         self.layout.addWidget(self.equation_text_edit)
41
42         self.solver_label = QLabel("Solver Type:")
43         self.layout.addWidget(self.solver_label)
44         self.solver_combo = QComboBox()
45         self.solver_combo.addItems(["FEM"]) # Add more solvers as needed
46         self.layout.addWidget(self.solver_combo)
47
48         self.params_label = QLabel("Parameters:")
49         self.layout.addWidget(self.params_label)
50         self.params_edit = QLineEdit()
51         self.layout.addWidget(self.params_edit)
52
53         self.solve_button = QPushButton("Solve")
54         self.solve_button.clicked.connect(self.solve)
55         self.layout.addWidget(self.solve_button)
56
57         self.clear_button = QPushButton("Clear")
58         self.clear_button.clicked.connect(self.clear)
59         self.layout.addWidget(self.clear_button)
60
61         self.canvas = FigureCanvas(plt.figure())
62         self.layout.addWidget(self.canvas)
63
64     def on_equation_combo_changed(self, index):
65         if index == 0:
66             self.equation_text_edit.setEnabled(True)
67         else:
68             self.equation_text_edit.setEnabled(False)
69
70     def solve(self):
71         equation_type = self.equation_combo.currentText()
72         solver_type = self.solver_combo.currentText()
73         if equation_type == "Custom Equation":
74             equation = self.equation_text_edit.toPlainText()
75         else:
76             equation = None
77         params = self.params_edit.text()
78         fem_solver = FEMSolver(equation_type, solver_type)
79         x, y = fem_solver.solve(equation, parameters=params)
80         self.plot_solution(x, y)
81
82     def plot_solution(self, x, y):
83         ax = self.canvas.figure.add_subplot(111)
84         ax.clear()
85         ax.plot(x, y)
86         ax.set_xlabel("x")
87         ax.set_ylabel("y")
88         ax.set_title("Finite Element Method Solution")
89
90         # Add annotations (example: mark max value)
91         max_index = np.argmax(y)
92         max_x = x[max_index]
93         max_y = y[max_index]
94         ax.annotate("Max Value: (%s, %s)" % (max_x, max_y), xy=(max_x, max_y), xytext=(max_x, max_y + 0.5),
95                    arrowprops=dict(facecolor='black', shrink=0.05),
96                    )
97
98         # Show values on specific points
99         for i in range(0, len(x), len(x)//10): # Show values at 10 equally spaced points
100             ax.text(x[i], y[i], f'({x[i]:.2f}, {y[i]:.2f})', ha='center', va='bottom')
101
102         self.canvas.draw()
103
104     def clear(self):
105         self.canvas.figure.clear()
106         self.canvas.draw()
107
108 if __name__ == "__main__":
109     app = QApplication(sys.argv)
110     window = MainWindow()
111     window.show()
112     sys.exit(app.exec_())

```

Fig 27: GUI of calculator which calculates partial differential equation

Result-

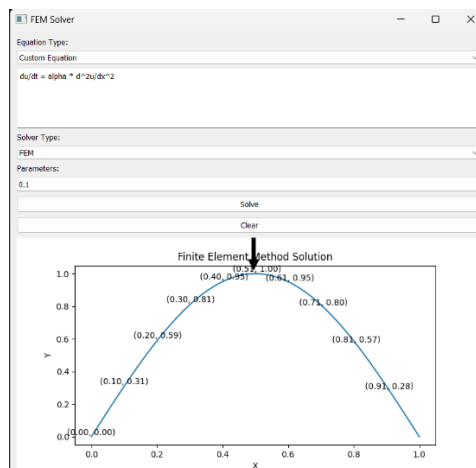


Fig 28: Partial Differential Equation's GUI Result

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

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

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

```
539 class TranscendentalEquationSolver(QWidget):
540     def __init__(self):
541         super().__init__()
542
543         self.initUI()
544
545     def initUI(self):
546         # Widgets
547         self.equation_label = QLabel("Enter the transcendental equation (use 'x' as the variable):")
548         self.equation_input = QLineEdit(self)
549
550         self.guess_label = QLabel("Enter an initial guess for the root:")
551         self.guess_input = QDoubleSpinBox(self)
552         self.guess_input.setRange(-1000, 1000)
553         self.guess_input.setSingleStep(0.1)
554
555         self.result_text = QTextEdit(self)
556         self.result_text.setReadOnly(True)
557
558         self.solve_button = QPushButton("Solve", self)
559         self.solve_button.clicked.connect(self.solve)
560
561         self.clear_button = QPushButton("Clear", self)
562         self.clear_button.clicked.connect(self.clear)
563
564         # Layout
565         layout = QVBoxLayout(self)
566         layout.addWidget(self.equation_label)
567         layout.addWidget(self.equation_input)
568         layout.addWidget(self.guess_label)
569         layout.addWidget(self.guess_input)
570         layout.addWidget(self.solve_button)
571         layout.addWidget(self.clear_button)
572         layout.addWidget(self.result_text)
573
574         self.setLayout(layout)
575
576         self.setGeometry(300, 300, 400, 300)
577         self.setWindowTitle("Transcendental Equation Solver")
578
579         # Apply qdarkstyle globally
580         self.setStyleSheet(qdarkstyle.load_stylesheet_pyqt5())
581
582     def solve(self):
583         equation_str = self.equation_input.text()
584         x = sp.symbols('x')
585         equation = sp.sympify(equation_str)
586         derivative = sp.diff(equation, x)
587         x0 = self.guess_input.value()
588
589         max_iterations = 20
590         tolerance = 1e-6
591
592         for i in range(max_iterations):
593             f_x = equation.subs(x, x0)
594             f_prime_x = derivative.subs(x, x0)
595             x0 = x0 - f_x.evalf() / f_prime_x.evalf()
596
597             self.result_text.append(f"Iteration {i + 1}: x = {x0}")
598
599             if sp.N(f_x.evalf()).is_zero:
600                 self.result_text.append(f"Converged to root: {x0}")
601                 break
602             else:
603                 self.result_text.append("Maximum number of iterations reached. Solution may not have converged.")
604
605     def clear(self):
606         # Clear the input fields and result text
607         self.equation_input.clear()
608         self.guess_input.clear()
609         self.result_text.clear()
```

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

Result-

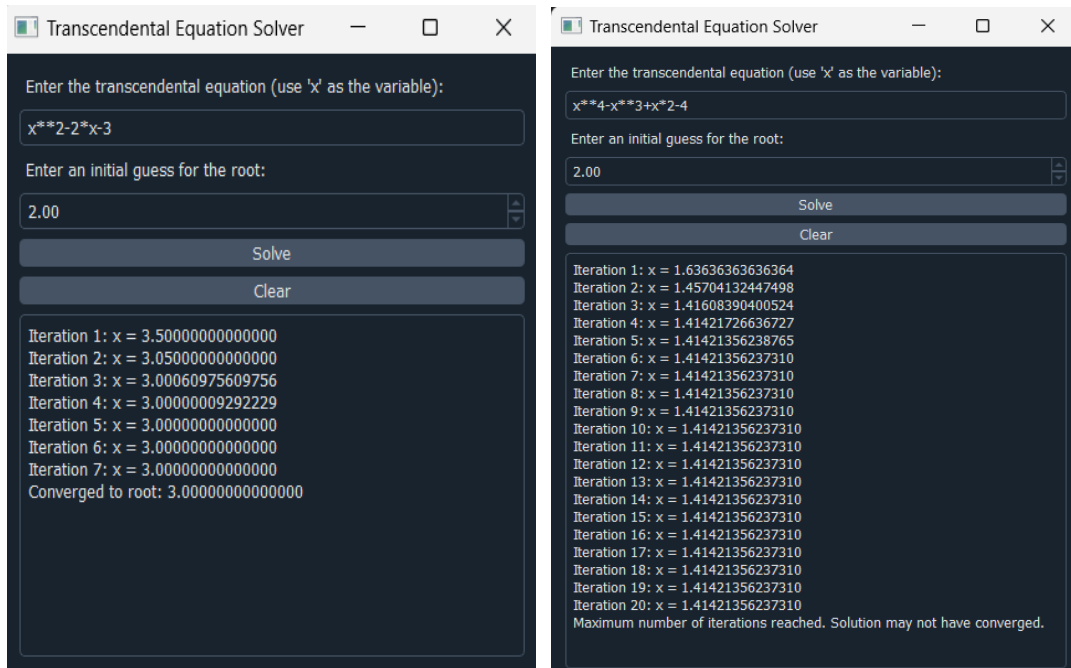


Fig 30: Numerical Method's GUI Result

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:

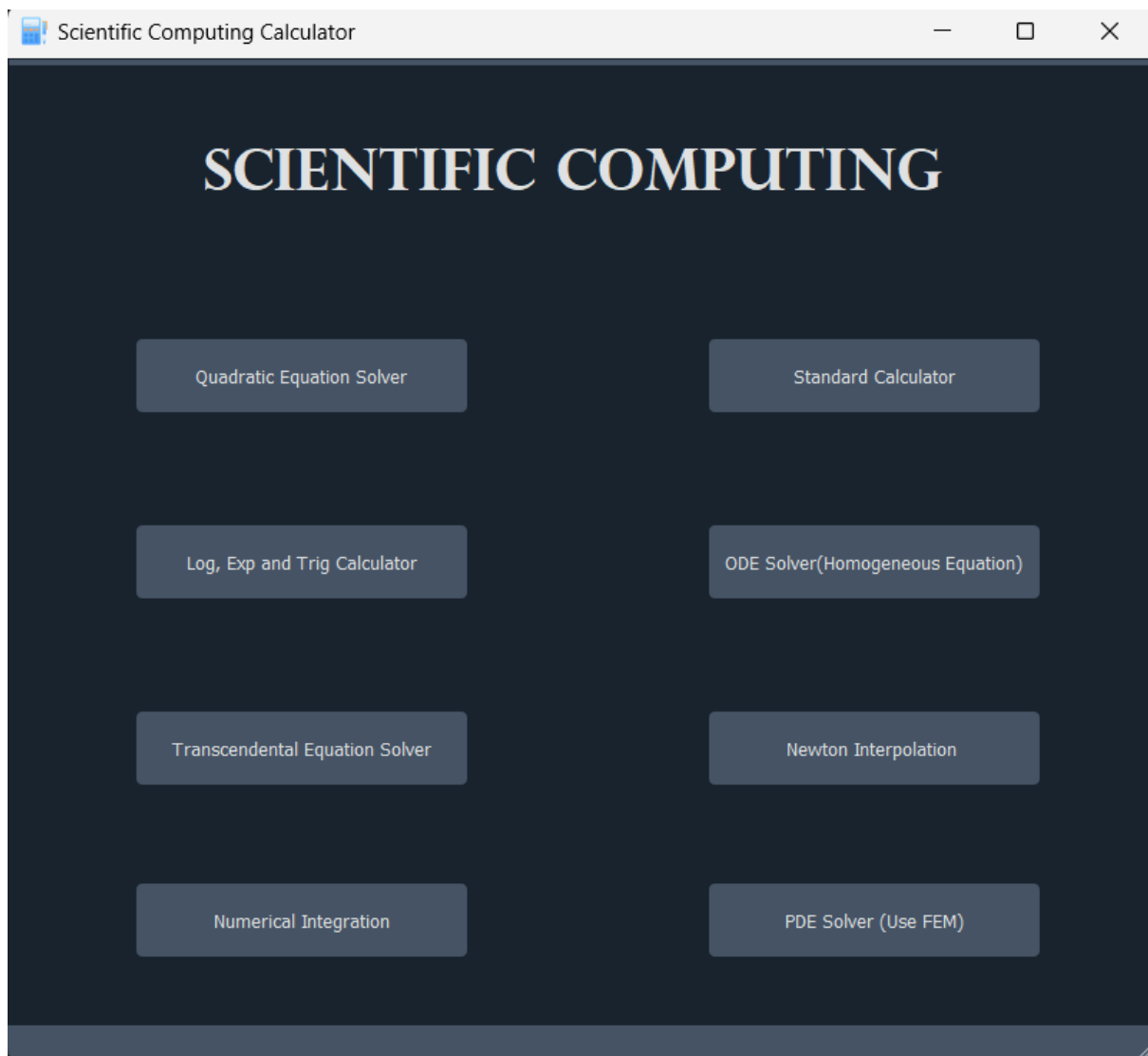


Fig 31: The Final Prototype

The code for this whole GUI is given below:

```

1 import sys
2 from math import log, exp, sin, cos, tan, radians, degrees, log10, atan
3 from PyQt5 import QtCore, QtGui, QtWidgets
4 from PyQt5.QtWidgets import *
5 from PyQt5.QtGui import *
6 from PyQt5.QtCore import *
7 from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as FigureCanvas
8 from scipy.integrate import odeint
9 import numpy as np
10 import numpy as np
11 import matplotlib.pyplot as plt
12 import sympy as sp
13
14 class Calculator(QtWidgets.QWidget):
15     def __init__(self):
16         super().__init__()
17
18         self.setWindowTitle("Log, exp & trig Calculator")
19         self.setGeometry(100, 100, 400, 400)
20
21         self.expression_lineedit = QLineEdit(self)
22         self.expression_lineedit.setPlaceholderText("Enter expression")
23
24         self.result_lineedit = QLineEdit(self)
25         self.result_lineedit.setReadOnly(True)
26
27         self.create_buttons()
28         self.create_layout()
29
30     def create_buttons(self):
31         buttons_layout = QGridLayout()
32
33         buttons = [
34             ('7', 0, 0),
35             ('8', 0, 1),
36             ('9', 0, 2),
37             ('4', 1, 0),
38             ('5', 1, 1),
39             ('6', 1, 2),
40             ('1', 2, 0),
41             ('2', 2, 1),
42             ('3', 2, 2),
43             ('0', 3, 0),
44             ('+', 3, 1),
45             ('-', 3, 2),
46             ('*', 4, 0),
47             ('/', 4, 1),
48             ('^', 4, 2),
49             ('ln', 5, 0),
50             ('exp', 5, 1),
51             ('sin', 5, 2),
52             ('cos', 6, 0),
53             ('tan', 6, 1),
54             ('C', 6, 2),
55             ('(', 7, 0),
56             (')', 7, 1),
57             ('%', 7, 2),
58             ('sqrt', 8, 0),
59             ('AC', 8, 1),
60         ]
61
62         for btn_text, row, col in buttons:
63             button = QPushButton(btn_text, self)
64             button.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Expanding)
65             button.clicked.connect(lambda: self.button_pressed(btn_text))
66             buttons_layout.addWidget(button, row, col)
67
68         self.buttons_layout = buttons_layout
69
70     def create_layout(self):
71         main_layout = QVBoxLayout()
72
73         main_layout.addWidget(self.expression_lineedit)
74         main_layout.addWidget(self.result_lineedit)
75         main_layout.addLayout(self.buttons_layout)
76
77         self.setLayout(main_layout)
78
79     # Set data points
80     self.setStyleSheet("""
81         QWidget {
82             background-color: #282828;
83             color: #f8f8f2;
84         }
85         QLineEdit {
86             background-color: #3c3b36;
87             color: #f8f8f2;
88             border: none;
89         }
90         QPushButton {
91             background-color: #3c3b36;
92             color: #f8f8f2;
93             border: none;
94         }
95         QPushButton:hover {
96             background-color: #45433b;
97         }
98     """)
99
100     def button_pressed(self, text):
101         current_text = self.expression_lineedit.text()
102
103         if text == '=':
104             try:
105                 result = eval(current_text)
106                 self.result_lineedit.setText(str(result))
107             except Exception as e:
108                 self.result_lineedit.setText("Error")
109         elif text == 'C':
110             new_text = current_text[:-1]
111             self.expression_lineedit.setText(new_text)
112         elif text == 'AC':
113             self.expression_lineedit.clear()
114             self.result_lineedit.clear()
115         else:
116             new_text = current_text + text
117             self.expression_lineedit.setText(new_text)
118
119 class NewtonInterpolationCalculator(QtWidgets.QWidget):
120     def __init__(self):
121         super().__init__()
122         self.setWindowTitle("Newton Interpolation Calculator")
123         self.setGeometry(250, 350, 500, 300)
124         self.initUI()
125
126     def initUI(self):
127         layout = QVBoxLayout()
128
129         self.data_points_layout = QVBoxLayout()
130         self.add_data_point_button = QPushButton("Add Data Point")
131         self.add_data_point_button.clicked.connect(self.add_data_point)
132         layout.addLayout(self.data_points_layout)
133         layout.addWidget(self.add_data_point_button)
134
135         self.x_interp_edit = QLineEdit()
136         self.x_interp_edit.setPlaceholderText("Enter x for interpolation")
137         self.interpolated_value_label = QLabel("Interpolated Value:")
138         layout.addWidget(self.x_interp_edit)
139         layout.addWidget(self.interpolated_value_label)
140
141         self.calculate_button = QPushButton("Calculate")
142         self.calculate_button.clicked.connect(self.calculate_interpolation)
143         layout.addWidget(self.calculate_button)
144
145         self.setLayout(layout)
146
147         self.data_points = []

```

```

149 def add_data_point(self):
150     data_point_layout = QHBoxLayout()
151     x_edit = QLineEdit()
152     y_edit = QLineEdit()
153     data_point_layout.addWidget(QLabel("x:"))
154     data_point_layout.addWidget(x_edit)
155     data_point_layout.addWidget(QLabel("y:"))
156     data_point_layout.addWidget(y_edit)
157     self.data_points_layout.addWidget(data_point_layout)
158     self.data_points.append((x_edit, y_edit))
159
160 def calculate_interpolation(self):
161     x_values = []
162     y_values = []
163     for x_edit, y_edit in self.data_points:
164         x_value = float(x_edit.text())
165         y_value = float(y_edit.text())
166         x_values.append(x_value)
167         y_values.append(y_value)
168
169     x_interp = float(self.x_interp_edit.text())
170
171     x_values = np.array(x_values)
172     y_values = np.array(y_values)
173
174     interpolated_value = self.newton_interpolation(x_values, y_values, x_interp)
175     self.interpolated_value_label.setText(f"Interpolated Value: {interpolated_value}")
176
177 def newton_interpolation(self, x, y, x_interp):
178     coef = self.divided_difference(x, y)
179     n = len(x)
180     result = coef[-1]
181     for i in range(n - 2, -1, -1):
182         result = result * (x_interp - x[i]) + coef[i]
183     return result
184
185 def divided_difference(self, x, y):
186     n = len(y)
187     coef = np.copy(y)
188     for j in range(1, n):
189         coef[j:] = (coef[j:] - coef[j - 1:-1]) / (x[j:] - x[:j-1])
190     return coef
191
192 class QuadraticEquationSolver(QWidget):
193     def __init__(self):
194         super().__init__()
195
196         self.setWindowTitle("Quadratic Equation Solver")
197         self.setGeometry(100, 100, 400, 250)
198         self.setStyleSheet("background-color: #232323; color: white;")
199
200         # Header label with FontAwesome icon
201         header_label = QLabel(self)
202         header_label.setText("<font size='5' color='RGBA(255, 255, 255, 0.8)'>i class='fab fa-font-awesome'</i><font size='5'>Quadratic Equation Solver</font>")
203         header_label.setAlignment(Qt.AlignmentFlag.AlignCenter)
204
205         # Labels for coefficients
206         label_a = QLabel("Coefficient a(x^2):", self)
207         label_b = QLabel("Coefficient b(x):", self)
208         label_c = QLabel("Coefficient c:", self)
209
210         # Input fields for coefficients
211         self.a_input = QLineEdit(self)
212         self.b_input = QLineEdit(self)
213         self.c_input = QLineEdit(self)
214
215         # Label to display the result
216         self.result_label = QLabel(self)
217         self.result_label.setAlignment(Qt.AlignmentFlag.AlignCenter)
218
219         # Solve button with hover effect
220         solve_button = QPushButton("Solve", self)
221         solve_button.clicked.connect(self.solve_quadratic)
222         solve_button.setStyleSheet("<QPushButton (background-color: #4CAF50; color: white; padding: 8px 16px; border: none; border-radius: 4px;)>
223             <QPushButton:Hover (background-color: #458040)>")
224
225         # Set size policy for each widget to expand to fill available space
226         header_label.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Preferred)
227         label_a.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Preferred)
228         self.a_input.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Preferred)
229         label_b.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Preferred)
230         self.b_input.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Preferred)
231         label_c.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Preferred)
232         self.c_input.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Preferred)
233         solve_button.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Preferred)
234
235         # Create layouts
236         input_layout = QHBoxLayout()
237         input_layout.addWidget(label_a)
238         input_layout.addWidget(self.a_input)
239         input_layout.addWidget(label_b)
240         input_layout.addWidget(self.b_input)
241         input_layout.addWidget(label_c)
242         input_layout.addWidget(self.c_input)
243
244         result_layout = QVBoxLayout()
245         result_layout.addWidget(self.result_label)
246
247         button_layout = QHBoxLayout()
248         button_layout.addWidget(solve_button)
249
250         main_layout = QVBoxLayout(self)
251         main_layout.addWidget(header_label)
252         main_layout.addLayout(input_layout)
253         main_layout.addLayout(result_layout)
254         main_layout.addWidget(button_layout)
255         main_layout.setAlignment(Qt.AlignmentFlag.AlignTop)
256
257     def solve_quadratic(self):
258         try:
259             a = float(self.a_input.text())
260             b = float(self.b_input.text())
261             c = float(self.c_input.text())
262
263             # Calculate the discriminant
264             discriminant = b**2 - 4*a*c
265
266             if discriminant > 0:
267                 root1 = (-b + discriminant**0.5) / (2*a)
268                 root2 = (-b - discriminant**0.5) / (2*a)
269                 result = f"Roots: {root1:.2f}, {root2:.2f}"
270             elif discriminant == 0:
271                 root1 = root2 = -b / (2*a)
272                 result = f"Roots: {root1:.2f}"
273             else:
274                 real_part = -b / (2*a)
275                 imaginary_part = (-discriminant**0.5) / (2*a)
276                 result = f"Roots: {real_part:.2f} + {imaginary_part:.2f}i, {real_part:.2f} - {imaginary_part:.2f}i"
277
278             self.animate_result(result)
279         except ValueError:
280             self.show_error("Invalid Input", "Please enter valid numerical coefficients.")
281
282     def show_error(self, title, message):
283         error_box = QMessageBox()
284         error_box.setIcon(QMessageBox.Critical)
285         error_box.setWindowTitle(title)
286         error_box.setText(message)
287         error_box.exec_()
288
289     @QtCore.pyqtSlot(str)
290     def animate_result(self, result):
291         self.result_label.setText(result)
292         animation = QPropertyAnimation(self.result_label, b"opacity")
293         animation.setStartValue(0)
294         animation.setEndValue(1)

```

```

297         animation.setDuration(1000)
298
299 class BasicCalculator(QWidget):
300     def __init__(self):
301         super().__init__()
302
303         self.setWindowTitle("Basic Calculator")
304         self.setWindowIcon(QIcon("C:/Users/VHP/Desktop/scientific-calculator.png"))
305         self.setGeometry(100, 100, 300, 400)
306
307         # Set dark theme
308         self.setStyleSheet("""
309             QWidget {
310                 background-color: #282828;
311                 color: #F8F8F2;
312             }
313             QPushButton {
314                 background-color: #3c3b36;
315                 color: #F8F8F2;
316                 border: none;
317             }
318             QPushButton:hover {
319                 background-color: #45433b;
320             }
321             QLineEdit {
322                 background-color: #3c3b36;
323                 color: #F8F8F2;
324                 border: none;
325             }
326         """)
327
328         # Create the main layout
329         main_layout = QHBoxLayout()
330
331         # Create the display widget
332         self.result_display = QLineEdit()
333         self.result_display.setReadOnly(True)
334         self.result_display.setStyleSheet(QSizePolicy.Expanding, QSizePolicy.Preferred)
335         main_layout.addWidget(self.result_display)
336
337         # Create the button grid layout
338         button_grid_layout = QVBoxLayout()
339
340         # Row 0
341         row0 = self.create_button_row(['+', '-', '*', '/'])
342         button_grid_layout.addWidget(row0)
343
344         # Row 1
345         row1 = self.create_button_row(['7', '8', '9', '/'])
346         button_grid_layout.addWidget(row1)
347
348         # Row 2
349         row2 = self.create_button_row(['4', '5', '6', '*'])
350         button_grid_layout.addWidget(row2)
351
352         # Row 3
353         row3 = self.create_button_row(['1', '2', '3', '-'])
354         button_grid_layout.addWidget(row3)
355
356         # Row 4
357         row4 = self.create_button_row(['0', '.', '=', '+'])
358         button_grid_layout.addWidget(row4)
359
360         # Add the button grid layout to the main layout
361         main_layout.addWidget(button_grid_layout)
362
363         # Set the main layout for the widget
364         self.setLayout(main_layout)
365
366     def create_button_row(self, button_texts):
367         row_layout = QHBoxLayout()
368
369         for button_text in button_texts:
370             button = QPushButton(button_text)
371             button.clicked.connect(self.on_button_click)
372             button.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Preferred)
373             row_layout.addWidget(button)
374
375         return row_layout
376
377     def on_button_click(self):
378         button = self.sender()
379         current_text = self.result_display.text()
380
381         if button.text() == "+":
382             try:
383                 result = str(eval(current_text))
384                 self.result_display.setText(result)
385             except Exception as e:
386                 self.result_display.setText("Error")
387         elif button.text() == "C":
388             self.result_display.clear()
389         else:
390             self.result_display.setText(current_text + button.text())
391
392     def on_clear_click(self):
393         self.result_display.clear()
394
395 class ODESolverApp(QWidget):
396     def __init__(self):
397         super().__init__()
398
399         self.initUI()
400
401     def initUI(self):
402         self.setStyleSheet("background-color: #2C3E50; color: #ECF8F1;") # Dark theme
403
404         # Order selection
405         self.order_label = QLabel("Select the order of the ODE:", self)
406         self.order_combobox = QComboBox(self)
407         self.order_combobox.addItems(['1', '2'])
408         self.order_combobox.setStyleSheet("background-color: #34495E; color: #ECF8F1;")
409
410         # Initial condition input
411         self.initial_condition_label = QLabel("Enter the initial condition:", self)
412         self.initial_condition_edit = QLineEdit(self)
413         self.initial_condition_edit.setStyleSheet("background-color: #34495E; color: #ECF8F1;")
414
415         # Coefficients input
416         self.coefficient_labels = ['a(y\\\'\\\'', 'b(y\\\'\\\'', 'c(y)']
417         self.coefficient_lineedit = [Label(QLineEdit(self) for label in self.coefficient_labels)
418                                     for line_edit in self.coefficient_lineedit.values()]
419         line_edit.setStyleSheet("background-color: #34495E; color: #ECF8F1;")
420
421         # Solve button
422         solve_button = QPushButton("Solve ODE", self)
423         solve_button.setStyleSheet("background-color: #34495E; color: #ECF8F1;")
424         solve_button.clicked.connect(self.solveODE)
425
426         # Cool icon for the button
427         cool_icon = QIcon("C:/Users/VHP/Downloads/Desertation/download.png")
428         solve_button.setIcon(cool_icon)
429         solve_button.setIconSize(QSize(24, 24))
430
431         # Result labels
432         self.result_label = QLabel(self)
433         self.result_label.setTextInteractionFlags(Qt.TextSelectableByMouse)
434         self.result_label.setStyleSheet("background-color: #34495E; color: #ECF8F1;")
435
436         # Plot area
437         self.figure, self.ax = plt.subplots()
438         self.canvas = FigureCanvas(self.figure)
439
440         # Layout
441         input_layout = QVBoxLayout()
442         input_layout.addWidget(self.order_label)
443         input_layout.addWidget(self.order_combobox)
444         input_layout.addWidget(self.initial_condition_label)
445         input_layout.addWidget(self.initial_condition_edit)

```

```

447         for label in self.coefficient_labels:
448             input_layout.addWidget(QLabel(f"Enter the coefficient '{label}':"))
449             input_layout.addWidget(self.coefficient_lineedit[label])
450
451         button_layout = QHBoxLayout()
452         button_layout.addWidget(solve_button)
453
454         main_layout = QVBoxLayout()
455         main_layout.addWidget(input_layout)
456         main_layout.addWidget(button_layout)
457         main_layout.addWidget(self.canvas)
458         main_layout.addWidget(self.result_label)
459
460         self.setLayout(main_layout)
461
462         self.setWindowTitle('ODE Solver')
463         self.setWindowIcon(QIcon("C:\\Users\\HP\\Downloads\\Dersertation\\download.png"))
464         self.setGeometry(100, 100, 800, 600)
465         self.show()
466
467     def rightSideODE(self, y, t, order):
468         a = float(self.coefficient_lineedit['a(y\\\'')'].text())
469         b = float(self.coefficient_lineedit['b(y\\\'')'].text())
470         c = float(self.coefficient_lineedit['c(y\\\'')'].text())
471
472         if order == 1:
473             dydt = (-b * y - c) / a
474         elif order == 2:
475             dydt = [y[1], (-b * y[1] - c * y[0]) / a]
476         else:
477             raise ValueError("Unsupported order of ODE")
478
479         return dydt
480
481     def solveODE(self):
482         order = int(self.order_combobox.currentText())
483
484         # Get coefficients from user input
485         coefficients = {label: float(self.coefficient_lineedit[label].text()) for label in self.coefficient_labels}
486
487         # Set initial conditions
488         initial_condition_text = self.initial_condition_edit.text()
489         initial_conditions = [float(val) for val in initial_condition_text.split(',')]
490
491         if order == 1:
492             y0 = initial_conditions[0]
493         elif order == 2:
494             y0 = initial_conditions
495
496         # Define the discretization points
497         timePoints = np.linspace(0, 10, 100)
498
499         # Solve the ODE using the modified rightSideODE method
500         solutionOde = odeint(self.rightSideODE, y0, timePoints, args=(order,))
501
502         # Clear previous plot
503         self.ax.clear()
504
505         # Plot the solution
506         if order == 1:
507             self.ax.plot(timePoints, solutionOde, 'b', label='y(t)')
508         elif order == 2:
509             self.ax.plot(timePoints, solutionOde[:, 0], 'b', label='y(t)')
510
511         self.ax.set_xlabel('time')
512         self.ax.set_ylabel('y(t)')
513
514         # Display roots and general solution
515         characteristic_roots = np.roots([coefficients['a(y\\\'')'], coefficients['b(y\\\'')'], coefficients['c(y\\\'')']])
516         roots_str = ', '.join([f'round(roots.real, 2)]' if root.imag == 0 else f'abs(round(roots.imag, 2))]' for root in characteristic_roots])
517
518         if np.all(np.isreal(characteristic_roots)):
519             root_type = "Real Roots"
520         else:
521             root_type = "Complex Roots"
522
523         general_solution = f"({root_type})\nroots: {roots_str}\n"
524         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))"
525
526         if order == 2:
527             general_solution += f"\nGeneral solution: y(t) = C1 * e^{((characteristic_roots[0]) * t)} + C2 * e^{((characteristic_roots[1]) * t)}"
528         else:
529             general_solution += f"\nFirst-order ODE solution: y(t) = C1 * e^{((-coefficients['b(y\\\'')'] / coefficients['a(y\\\'')']) * t)}"
530
531         self.result_label.setText(general_solution)
532
533         # Draw the new plot
534         self.ax.legend(loc='best')
535         self.ax.set_title(f"Solution of (coefficients['a(y\\\'')'])y\\\'' + (coefficients['b(y\\\'')'])y\\\'' + (coefficients['c(y\\\'')'])y = 0")
536         self.ax.grid()
537         self.canvas.draw()
538
539     class TranscendentalEquationSolver(QWidget):
540     def __init__(self):
541         super().__init__()
542
543         self.initUI()
544
545     def initUI(self):
546         # Widgets
547         self.equation_label = QLabel("Enter the transcendental equation (use 'x' as the variable):")
548         self.equation_input = QLineEdit(self)
549
550         self.guess_label = QLabel("Enter an initial guess for the root:")
551         self.guess_input = QLineEdit(self)
552         self.guess_input.setRange(-1000, 1000)
553         self.guess_input.setSingleStep(0.1)
554
555         self.result_text = QTextEdit(self)
556         self.result_text.setReadOnly(True)
557
558         self.solve_button = QPushButton("Solve", self)
559         self.solve_button.clicked.connect(self.solve)
560
561         self.clear_button = QPushButton("Clear", self)
562         self.clear_button.clicked.connect(self.clear)
563
564         # Layout
565         layout = QVBoxLayout(self)
566         layout.addWidget(self.equation_label)
567         layout.addWidget(self.equation_input)
568         layout.addWidget(self.guess_label)
569         layout.addWidget(self.guess_input)
570         layout.addWidget(self.solve_button)
571         layout.addWidget(self.clear_button)
572         layout.addWidget(self.result_text)
573
574         self.setLayout(layout)
575
576         self.setGeometry(200, 300, 400, 300)
577         self.setWindowTitle('Transcendental Equation Solver')
578
579         # Apply qdarkstyle globally
580         self.setStyleSheet(qdarkstyle.load_stylesheet_pyqt5())
581
582     def solve(self):
583         equation_str = self.equation_input.text()
584         x = sp.symbols('x')
585         equation = sp.simplify(equation_str)
586         derivative = sp.diff(equation, x)
587         x0 = self.guess_input.value()
588
589         max_iterations = 20
590         tolerance = 1e-6
591
592         for i in range(max_iterations):
593             f_x = equation.subs(x, x0)
594             f_prime_x = derivative.subs(x, x0)

```



```

595         x0 = x0 - f_x.evalf() / f_prime_x.evalf()
596
597         self.result_text.append("Iteration (i + 1): x = (x0)")
598
599         if up.N(f_x.evalf()) is zero:
600             self.result_text.append("Converged to root: (x0)")
601             break
602         else:
603             self.result_text.append("Maximum number of iterations reached. Solution may not have converged.")
604
605     def clear(self):
606         # Clear the input fields and result text
607         self.equation_input.clear()
608         self.result_text.clear()
609
610 class NumericalIntegration(QWidget):
611     def __init__(self):
612         super().__init__()
613         self.setWindowTitle("Numerical Integration using Simpson's 3/8th Rule")
614         self.init_ui()
615
616     def init_ui(self):
617         layout = QVBoxLayout()
618
619         self.equation_label = QLabel("Enter the equation (use 'x' as variable):")
620         self.equation_input = QLineEdit()
621
622         self.lower_limit_label = QLabel("Lower limit:")
623         self.lower_limit_input = QLineEdit()
624         self.lower_limit_input.setReadOnly(True)
625
626         self.upper_limit_label = QLabel("Upper limit:")
627         self.upper_limit_input = QLineEdit()
628         self.upper_limit_input.setReadOnly(True)
629
630         self.subintervals_label = QLabel("Number of subintervals:")
631         self.subintervals_input = QLineEdit()
632         self.subintervals_input.setReadOnly(True)
633
634         self.result_label = QLabel("Result:")
635         self.result_display = QLabel("")
636
637         self.calculate_button = QPushButton("Calculate")
638         self.calculate_button.clicked.connect(self.calculate_integration)
639
640         self.clear_button = QPushButton("Clear")
641         self.clear_button.clicked.connect(self.clear_inputs)
642
643         self.backspace_button = QPushButton("Backspace")
644         self.backspace_button.clicked.connect(self.backspace_input)
645
646         self.edit_limit_button = QPushButton("Edit Limits")
647         self.edit_limit_button.clicked.connect(self.enable_limit_editing)
648
649         self.grid_layout = QGridLayout()
650         buttons = ['+', '-', '*', '/', '^', '1/x', 'sin', 'cos', 'tan', 'exp', 'log', 'sqrt']
651         positions = [(i, j) for i in range(5) for j in range(4)]
652
653         for position, button in zip(positions, buttons):
654             btn = QPushButton(button)
655             if button == 'x^2':
656                 btn.clicked.connect(self.insert_x_squared)
657             else:
658                 btn.clicked.connect(lambda: self.add_to_equation_input(text))
659             self.grid_layout.addWidget(btn, *position)
660
661         grid_widget = QWidget()
662         grid_widget.setLayout(self.grid_layout)
663
664         layout.addWidget(self.equation_label)
665         layout.addWidget(self.equation_input)
666         layout.addWidget(self.lower_limit_label)
667         layout.addWidget(self.lower_limit_input)
668         layout.addWidget(self.upper_limit_label)
669         layout.addWidget(self.upper_limit_input)
670         layout.addWidget(self.subintervals_label)
671         layout.addWidget(self.subintervals_input)
672         layout.addWidget(grid_widget) # Add the grid layout widget
673         layout.addWidget(self.calculate_button)
674         layout.addWidget(self.clear_button)
675         layout.addWidget(self.backspace_button)
676         layout.addWidget(self.edit_limit_button)
677         layout.addWidget(self.result_label)
678         layout.addWidget(self.result_display)
679
680         self.setLayout(layout)
681
682     def simpsons_3_8_rule(f, a, b, n):
683         """
684         Numerical integration using Simpson's 3/8th rule.
685         f: Function to integrate.
686         a: Lower limit of integration.
687         b: Upper limit of integration.
688         n: Number of sub-intervals.
689         """
690         h = (b - a) / n
691         result = f(a) + f(b)
692         for i in range(1, n):
693             if i % 3 == 0:
694                 result += 2 * f(a + i * h)
695             else:
696                 result += 3 * f(a + i * h)
697         result *= 3 * h / 8
698         return result
699
700     def add_to_equation_input(self, text):
701         current_text = self.equation_input.text()
702
703         # Handle arithmetic operations
704         if text in ['+', '-', '*', '/', '^']:
705             # Add space before and after the operator
706             text = ' ' + text + ' '
707
708         self.equation_input.setText(current_text + text)
709
710     def insert_x_squared(self):
711         current_text = self.equation_input.text()
712         x = symbols('x')
713         x_squared = x**2
714         self.equation_input.setText(current_text + str(x_squared))
715
716     def enable_limit_editing(self):
717         self.lower_limit_input.setReadOnly(False)
718         self.upper_limit_input.setReadOnly(False)
719         self.subintervals_input.setReadOnly(False)
720
721     def clear_inputs(self):
722         self.equation_input.clear()
723         self.lower_limit_input.clear()
724         self.upper_limit_input.clear()
725         self.subintervals_input.clear()
726         self.result_display.clear()
727
728     def backspace_input(self):
729         self.equation_input.backspace()
730
731     def calculate_integration(self):
732         try:
733             equation = self.equation_input.text()
734             lower_limit = float(self.lower_limit_input.text())
735             upper_limit = float(self.upper_limit_input.text())
736             n = int(self.subintervals_input.text())
737
738             # Define the function to integrate
739             def f(x):
740                 return eval(equation)
741
742             result = simpsons_3_8_rule(f, lower_limit, upper_limit, n)

```

```

743         # Handle 'log' and 'exp' functions
744         equation = equation.replace('log', 'math.log')
745         equation = equation.replace('exp', 'math.exp')
746
747         result = simpsons_3_8_rule(f, lower_limit, upper_limit, n)
748         self.result_display.setText(str(result))
749
750     except Exception as e:
751         self.result_display.setText("Error: " + str(e))
752
753 class FEMSolver(QWidget):
754     def __init__(self, equation_type, solver_type):
755         self.equation_type = equation_type
756         self.solver_type = solver_type
757
758     def solve(self, equation, **kwargs):
759         x = np.linspace(0, 1, 100)
760         y = np.sin(np.pi * x)
761         return x, y
762
763 class MainWindow(QMainWindow):
764     def __init__(self):
765         super().__init__()
766
767         self.setWindowTitle("FEM Solver")
768         self.setGeometry(100, 100, 800, 600)
769
770         self.central_widget = QWidget()
771         self.setCentralWidget(self.central_widget)
772
773         self.layout = QVBoxLayout()
774         self.central_widget.setLayout(self.layout)
775
776         self.equation_label = QLabel("Equation Type:")
777         self.layout.addWidget(self.equation_label)
778         self.equation_combo = QComboBox()
779         self.equation_combo.addItem("Custom Equation", "Heat Equation", "Wave Equation") # Add more equations as needed
780         self.equation_combo.currentIndexChanged.connect(self.on_equation_combo_changed)
781         self.layout.addWidget(self.equation_combo)
782
783         self.equation_text_edit = QTextEdit()
784         self.layout.addWidget(self.equation_text_edit)
785
786         self.solver_label = QLabel("Solver Type:")
787         self.layout.addWidget(self.solver_label)
788         self.solver_combo = QComboBox()
789         self.solver_combo.addItem("FEM") # Add more solvers as needed
790         self.layout.addWidget(self.solver_combo)
791
792         self.params_label = QLabel("Parameters:")
793         self.layout.addWidget(self.params_label)
794         self.params_edit = QLineEdit()
795         self.layout.addWidget(self.params_edit)
796
797         self.solve_button = QPushButton("Solve")
798         self.solve_button.clicked.connect(self.solve)
799         self.layout.addWidget(self.solve_button)
800
801         self.clear_button = QPushButton("Clear")
802         self.clear_button.clicked.connect(self.clear)
803         self.layout.addWidget(self.clear_button)
804
805         self.canvas = FigureCanvas(plt.figure())
806         self.layout.addWidget(self.canvas)
807
808     def on_equation_combo_changed(self, index):
809         if index == 0:
810             self.equation_text_edit.setEnabled(True)
811         else:
812             self.equation_text_edit.setEnabled(False)
813
814     def solve(self):
815         equation_type = self.equation_combo.currentText()
816         solver_type = self.solver_combo.currentText()
817         if equation_type == "Custom Equation":
818             equation = self.equation_text_edit.toPlainText()
819         else:
820             equation = None
821         params = self.params_edit.text()
822
823         fem_solver = FEMSolver(equation_type, solver_type)
824         x, y = fem_solver.solve(equation, parameters=params)
825
826         self.plot_solution(x, y)
827
828     def plot_solution(self, x, y):
829         ax = self.canvas.figure.add_subplot(111)
830         ax.clear()
831         ax.plot(x, y)
832         ax.set_xlabel("x")
833         ax.set_ylabel("y")
834         ax.set_title("Finite Element Method Solution")
835
836         # Add annotations (example: mark max value)
837         max_index = np.argmax(y)
838         max_x = x[max_index]
839         max_y = y[max_index]
840         ax.annotate('Max Value: (max_x, %f)' % (max_y, 2f), xy=(max_x, max_y), xytext=(max_x, max_y + 0.5),
841                    arrowprops=dict(facecolor='black', shrink=0.05),
842                    )
843
844         # Show values on specific points
845         for i in range(0, len(x), len(x)//10): # Show values at 10 equally spaced points
846             ax.text(x[i], y[i], '%.2f' % (x[i], 2f), y[i], 2f), ha='center', va='bottom')
847
848         self.canvas.draw()
849
850     def clear(self):
851         self.canvas.figure.clear()
852         self.canvas.draw()
853
854 class UI_MainWindow(QObject):
855     def setupUI(self, MainWindow):
856         MainWindow.setObjectName("MainWindow")
857         MainWindow.setStyleSheet("background-color: #f0f0f0;")
858         MainWindow.resize(800, 600)
859         MainWindow.setStyleSheet(qdarkstyle.load_stylesheet_pyqt5())
860         self.centralwidget = QWidget(MainWindow)
861         self.centralwidget.setObjectName("centralwidget")
862         self.label = QLabel(self.centralwidget)
863         self.label.setGeometry(QRect(30, 20, 731, 101))
864         font = QFont()
865         font.setFamily("Futura Titled MT")
866         font.setPointSize(24)
867         font.setBold(True)
868         font.setWeight(75)
869         self.label.setFont(font)
870         self.label.setAlignment(Qt.AlignCenter)
871         self.label.setObjectName("label")
872         self.qs = QPushButton(self.centralwidget)
873         self.qs.setGeometry(QRect(50, 150, 231, 51))
874         self.qs.setObjectName("qs")
875         self.qs.clicked.connect(self.show_quadratic_solver)
876         self.sc = QPushButton(self.centralwidget)
877         self.sc.setGeometry(QRect(450, 150, 231, 51))
878         self.sc.setObjectName("sc")
879         self.sc.clicked.connect(self.show_basic_calculator)
880         self.let = QPushButton(self.centralwidget)
881         self.let.setGeometry(QRect(50, 200, 231, 51))
882         self.let.setObjectName("let")
883         self.let.clicked.connect(self.show_calculator)
884         self.ode = QPushButton(self.centralwidget)
885         self.ode.setGeometry(QRect(450, 200, 231, 51))
886         self.ode.setObjectName("ode")
887         self.ode.clicked.connect(self.show_ode_solver_app)
888         self.ts = QPushButton(self.centralwidget)
889         self.ts.setGeometry(QRect(50, 250, 231, 51))
890         self.ts.setObjectName("ts")

```

```

991 self.tes.clicked.connect(self.show_TranscendentalEquationSolver)
992 self.nic = QtWidgets.QPushButton(self.centralwidget)
993 self.nic.setGeometry(QtCore.QRect(490, 450, 231, 51))
994 self.nic.setObjectName("nic")
995 self.nic.clicked.connect(self.show_NewtonInterpolationCalculator)
996 self.ni = QtWidgets.QPushButton(self.centralwidget)
997 self.ni.setGeometry(QtCore.QRect(90, 570, 231, 51))
998 self.ni.setObjectName("ni")
999 self.ni.clicked.connect(self.show_NumericalIntegration)
1000 self.pde = QtWidgets.QPushButton(self.centralwidget)
1001 self.pde.setGeometry(QtCore.QRect(490, 570, 231, 51))
1002 self.pde.setObjectName("pde")
1003 self.pde.clicked.connect(self.show_PDESolver)
1004 MainWindow.setCentralWidget(self.centralwidget)
1005 self.menubar = QtWidgets.QMenuBar(MainWindow)
1006 self.menubar.setGeometry(QtCore.QRect(0, 0, 800, 26))
1007 self.menubar.setObjectName("menubar")
1008 MainWindow.setMenuBar(self.menubar)
1009 self.statusbar = QtWidgets.QStatusBar(MainWindow)
1010 self.statusbar.setObjectName("statusbar")
1011 MainWindow.setStatusBar(self.statusbar)
1012
1013 self.retranslateUi(MainWindow)
1014 QtCore.QMetaObject.connectSlotsByName(MainWindow)
1015
1016
1017
1018 def retranslateUi(self, MainWindow):
1019     _translate = QtCore.QCoreApplication.translate
1020     MainWindow.setWindowTitle(_translate("MainWindow", "Scientific Computing Calculator"))
1021     self.label.setText(_translate("MainWindow", "SCIENTIFIC COMPUTING"))
1022     self.qes.setText(_translate("MainWindow", "Quadratic Equation Solver"))
1023     self.ies.setText(_translate("MainWindow", "Log, Exp and Trig Calculator"))
1024     self.ic.setText(_translate("MainWindow", "Inverses Calculator"))
1025     self.ode.setText(_translate("MainWindow", "ODE Solver(Homogeneous Equation))"))
1026     self.tes.setText(_translate("MainWindow", "Transcendental Equation Solver"))
1027     self.nic.setText(_translate("MainWindow", "Newton Interpolation "))
1028     self.ni.setText(_translate("MainWindow", "Numerical Integration"))
1029     self.pde.setText(_translate("MainWindow", "PDE Solver (Use FEM)"))
1030
1031
1032 def show_quadratic_solver(self):
1033     self.quadratic_solver = QuadraticEquationSolver()
1034     self.quadratic_solver.show()
1035
1036 def show_basic_calculator(self):
1037     self.basic_calculator = BasicCalculator()
1038     self.basic_calculator.show()
1039
1040 def show_calculator(self):
1041     self.calculator = Calculator()
1042     self.calculator.show()
1043
1044 def show_ODESolverApp(self):
1045     self.ODESolverApp = ODESolverApp()
1046     self.ODESolverApp.show()
1047
1048 def show_TranscendentalEquationSolver(self):
1049     self.TranscendentalEquationSolver = TranscendentalEquationSolver()
1050     self.TranscendentalEquationSolver.show()
1051
1052 def show_NewtonInterpolationCalculator(self):
1053     self.NewtonInterpolationCalculator = NewtonInterpolationCalculator()
1054     self.NewtonInterpolationCalculator.show()
1055
1056 def show_NumericalIntegration(self):
1057     self.NumericalIntegration = NumericalIntegration()
1058     self.NumericalIntegration.show()
1059
1060 def show_PDESolver(self):
1061     self.PDESolver = PDESolver()
1062     self.PDESolver.show()
1063
1064 if __name__ == "__main__":
1065     app = QtWidgets.QApplication(sys.argv)
1066     MainWindow = QtWidgets.QMainWindow()
1067     ui = Ui_MainWindow()
1068     ui.setupUi(MainWindow)
1069     MainWindow.show()
1070     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:

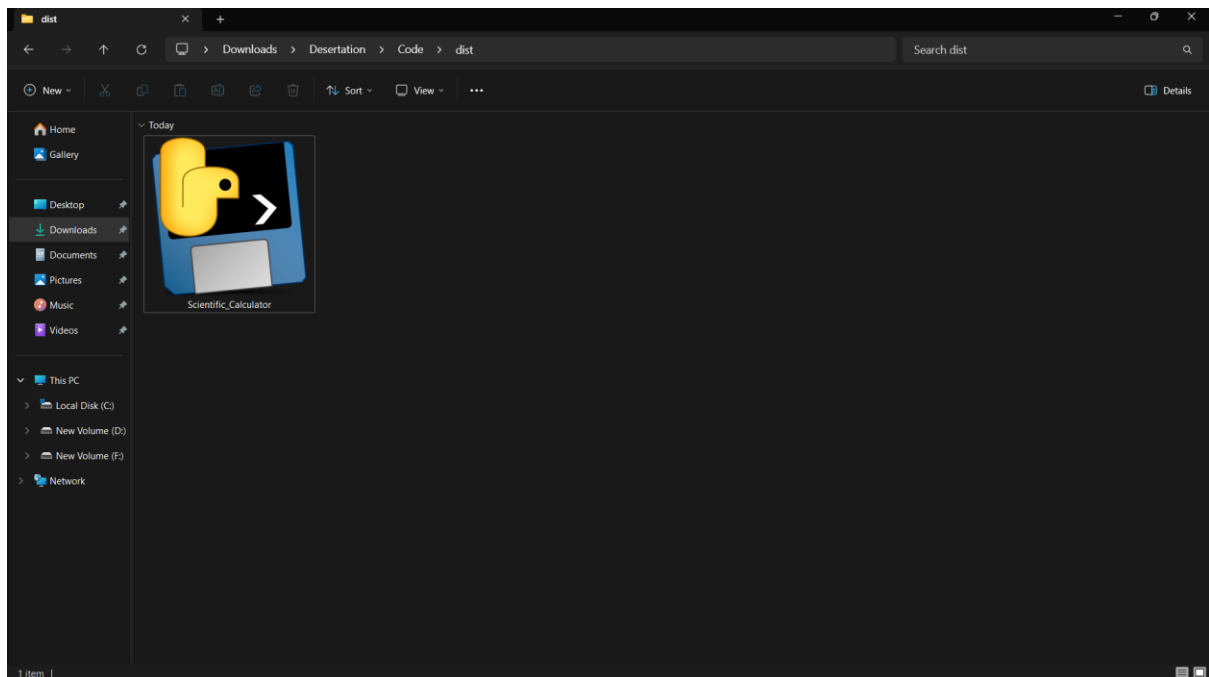


Fig 33: Executable File

References:

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