TABLE OF CONTENTS

Abstract

This report presents the development and implementation of a 2D graph plotter using Python, a versatile programming language renowned for its data analysis and visualization capabilities. The primary goal of this project is to create a tool that enables users to generate and customize various types of 2D graphs, including line plots, scatter plots, and bar charts, to facilitate data interpretation and analysis. By leveraging Python libraries such as Matplotlib and Seaborn, the plotter provides an intuitive interface for visualizing relationships between two variables, adjusting graph parameters, and exporting visual outputs.

The report covers the design and architecture of the plotter, detailing its functional and non-functional requirements, and offers a comprehensive overview of the implementation process. It includes code snippets, explanations of key functionalities, and descriptions of the user interface. Testing and validation results are presented to demonstrate the tool's performance, reliability, and accuracy. Additionally, the report explores real-world applications through case studies and user feedback, highlighting the plotter’s impact on various fields such as research and business analytics.

Future work and potential enhancements are discussed, outlining opportunities for expanding the tool's capabilities and integrating advanced features. Overall, this report provides a detailed account of the 2D graph plotter project, showcasing its effectiveness as a data visualization tool and its contributions to simplifying data analysis tasks.

CHAPTER 1

INTRODUCTION

1.1 Background

In the realm of data analysis, effective visualization is paramount for interpreting and communicating insights derived from complex datasets. Graphs and charts transform raw data into visual representations, making it easier to identify patterns, trends, and anomalies. Among the various types of visualizations, 2D graphs are fundamental tools that represent data across two dimensions, typically plotting variables against each other to reveal relationships and insights.

2D graph plotting is employed extensively across numerous fields, including science, engineering, economics, and business. For instance, scientists use 2D plots to illustrate experimental results, while businesses utilize them to analyze market trends and performance metrics. Common types of 2D graphs include line plots, scatter plots, and bar charts, each serving distinct purposes:

- Line Plots: Ideal for showing trends over time or continuous data. They connect data points with a line, highlighting changes and patterns.
- Scatter Plots: Useful for examining the relationship between two numerical variables. They display individual data points, allowing for the identification of correlations or clusters.
- Bar Charts: Effective for comparing discrete categories or groups. They use rectangular bars to represent the frequency or magnitude of different categories.

1.2 Objective

The objective of developing a 2D graph plotter is to create a user-friendly tool that enables efficient and customizable visualization of data. The plotter aims to:

1. Facilitate Data Visualization: Simplify the process of creating various types of 2D graphs, allowing users to quickly translate data into meaningful visual representations.
2. Provide Customization Options: Offer extensive customization features for adjusting graph parameters such as axis labels, titles, colors, and legends, ensuring that users can tailor the visualizations to their specific needs.
3. Support Multiple Data Formats: Handle different data inputs, including lists, arrays,

and data frames, to accommodate diverse data sources and structures.

4. Enhance Accessibility: Enable users with varying levels of technical expertise to create and interpret 2D graphs without requiring advanced programming skills.

By achieving these objectives, the plotter enhances the capability of users to analyze and present data effectively, making it a valuable tool for data scientists, researchers, and business analysts alike.

1.3 Scope

The scope of the 2D graph plotter encompasses several key aspects:

- Features: The plotter supports various graph types, including line plots, scatter plots, and bar charts. It provides customization options such as adjusting graph size, colors, and labels, as well as saving and exporting graphs in different formats (e.g., PNG, PDF).
- Limitations: While the plotter offers essential graphing functionalities, it does not cover advanced visualization techniques such as 3D plots or interactive graphs. Performance may vary depending on the complexity and size of the data being processed.
- Applications: The tool is designed for a broad range of applications, including academic research, data analysis, and business reporting. It is intended to streamline the process of data visualization, making it accessible to both novice users and experienced analysts.

1.4 Motivation and Significance

The development of a 2D graph plotter addresses the growing need for accessible and versatile data visualization tools. As data continues to grow in volume and complexity, the ability to quickly and effectively visualize data is crucial. A well-designed plotter not only aids in data interpretation but also enhances communication by presenting information in a clear and engaging manner.

The significance of this project lies in its potential to simplify data analysis and decision-making processes. By providing a straightforward tool for generating 2D graphs, the plotter empowers users to uncover insights, make informed decisions, and convey findings effectively.

1.5 Overview of Report

This report details the development and implementation of the 2D graph plotter, covering its design, functionality, and performance. It begins with a literature review of data visualization techniques and Python libraries, followed by a discussion of the system's design and architecture. The implementation section provides an in-depth look at the code and functionality, while testing and performance evaluations assess the tool's effectiveness. Case studies demonstrate real-world applications, and future work outlines potential enhancements. The report concludes with a summary of key findings and contributions.

1.1 Modules Used

- Matplotlib
- NumPy
- Pandas
- Bokeh
- Seaborn

i. Matplotlib

Matplotlib is one of the most widely used libraries for 2D plotting in Python. It provides a comprehensive set of tools for creating static, animated, and interactive plots.

Key Features:

- **Plot Types**: Supports line plots, scatter plots, bar charts, histograms, pie charts, and more.
- **Customization**: Offers extensive options for customizing plots, including adjusting axis labels, titles, legends, colors, and styles.
- **Integration**: Works well with other libraries like NumPy and Pandas for data manipulation and analysis.

ii. NumPy

NumPy is a fundamental library for numerical computations in Python. While it is not specifically a plotting library, it is often used to generate or manipulate data for plotting.

Key Features:

- Array Operations: Efficient operations on large arrays and matrices.
- Mathematical Functions: Provides functions for mathematical operations, including statistical calculations.

iii. Pandas

Pandas is a powerful data manipulation library that also includes basic plotting capabilities. It is particularly useful for handling and preparing data before visualization.

Key Features:

- Data Handling: Provides DataFrame and Series objects for efficient data manipulation.
- Plotting: Simple and quick plotting using DataFrame.plot() and Series.plot() methods.
- Integration: Works well with Matplotlib for more advanced customization.

iv. Bokeh

Bokeh is another interactive visualization library that enables the creation of web-ready plots and dashboards. It focuses on providing interactive plots that can be embedded into web applications.

Key Features:

- Interactivity: Features such as zooming, panning, and hover tools.
- Web Integration: Designed for easy integration into web applications and dashboards.

v. Seaborn

Seaborn is a statistical data visualization library based on Matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics.

Key Features:

- **Enhanced Visualization**: Built-in themes and color palettes for improved aesthetics.
- **Statistical Plots**: Simplifies the creation of complex plots like regression plots, pair plots, and heatmaps.
- **Integration**: Works seamlessly with Pandas DataFrames.

CHAPTER 2

2. Literature Survey

2.1 Evolution of Data Visualization

Early Techniques: The history of data visualization dates back to the 18th century, with pioneers like William Playfair, who is credited with creating some of the first statistical graphics. His innovations included the line chart, bar chart, and pie chart, which laid the foundation for modern data visualization techniques.

20th Century Developments: In the 20th century, advancements in computing technology led to the development of more sophisticated graphing tools. Early graphical user interfaces (GUIs) and software like MATLAB (introduced in 1984) provided researchers with enhanced capabilities for plotting and analyzing data.

Modern Era: The rise of open-source software and programming languages like Python in the 21st century democratized data visualization, allowing for greater accessibility and customization. This era saw the emergence of numerous libraries and tools designed to simplify and enhance the process of creating 2D graphs.

2.2 Data Visualization Techniques

Graph Types and Their Uses:

- Line Graphs: Used to represent data trends over time or continuous variables. Line graphs are essential for time series analysis and showing temporal changes.
- Scatter Plots: Display individual data points and are ideal for examining the relationship between two variables. They help in identifying correlations, outliers, and clusters.
- Bar Charts: Effective for comparing discrete categories. Bar charts can represent frequencies or amounts, making them useful for categorical data analysis.
- Histograms: Provide insights into the distribution of a dataset by grouping data into bins. They are useful for understanding the frequency distribution of continuous data.

Visualization Principles: Edward Tufte's work on data visualization emphasizes the principles of graphical excellence, clarity, and integrity. His books, such as "The Visual Display of Quantitative Information," highlight the importance of minimizing chartjunk and presenting data in a clear and truthful manner.

2.3 Python Libraries for 2D Graph Plotting

i. Matplotlib:

- Overview: Developed by John D. Hunter in 2003, Matplotlib is a foundational library for 2D plotting in Python. It provides a flexible framework for creating static, animated, and interactive plots.
- Features: Includes support for various plot types, customization options, and integration with other Python libraries.
- Usage: Widely used in academic research, data analysis, and scientific computing due to its comprehensive feature set and flexibility.

ii. Seaborn:

- Overview: Built on top of Matplotlib and created by Michael Waskom, Seaborn enhances statistical graphics and offers improved aesthetics and functionality.
- Features: Provides built-in themes and color palettes, as well as tools for complex visualizations like regression plots and pair plots.
- Usage: Commonly used in data science and machine learning for its ease of creating attractive and informative statistical visualizations.

iii. Pandas:

- Overview: Primarily a data manipulation library, Pandas includes basic plotting capabilities integrated with its DataFrame and Series objects.
- Features: Simplifies plotting directly from data structures, facilitating quick and efficient visualization of data.
- Usage: Often used in data analysis workflows for preliminary plotting and data exploration.

iv. Plotly:

- Overview: A modern library for creating interactive plots and dashboards, Plotly supports a wide range of visualization types, including 3D plots and geographic maps.
- Features: Interactive features such as zooming, panning, and hover effects, as well as integration with web applications.
- Usage: Ideal for web-based applications and interactive data exploration.

v. Bokeh:

- Overview: Bokeh is designed for creating interactive and web-ready visualizations,

focusing on providing rich interactivity and integration with web technologies.

- Features: Includes tools for creating dashboards and embedding plots in web applications.
- Usage: Used for building interactive data visualizations and web-based analytics tools.

2.4 Comparative Analysis of Libraries

Ease of Use:

- Matplotlib: Offers extensive functionality but may require more effort for customization.
- Seaborn: Provides a higher-level interface and attractive defaults, making it easier to produce complex statistical plots.
- Pandas: Simplifies the plotting process for users already working with Pandas DataFrames.

Interactivity:

- Plotly and Bokeh: Both libraries excel in creating interactive plots, with Plotly offering more advanced features and integration capabilities.

Integration:

- Matplotlib and Seaborn: Strong integration with Python's data analysis ecosystem (e.g., NumPy, Pandas).
- Plotly and Bokeh: Better suited for web applications and real-time data interaction.

2.5 Current Trends and Future Directions

Interactive Visualizations: The growing importance of interactive data visualization reflects the need for dynamic and user-engaging graphics. Interactive plots allow users to explore data more deeply and gain insights through interaction.

Integration with Big Data Technologies: As data sizes continue to grow, integrating visualization tools with big data technologies (e.g., Spark, Hadoop) becomes increasingly important. Libraries are evolving to handle larger datasets and real-time data processing.

Machine Learning Integration: Combining data visualization with machine learning can enhance the ability to interpret complex models and their outputs. Visualizing model performance, feature importance, and predictions can provide valuable insights.

Customization and Aesthetics: There is an ongoing trend towards improving the aesthetics and customization options of plots. Libraries are continuously adding features to enhance the visual appeal and clarity of graphs.

CHAPTER 3

REQUIREMENT SPECIFICSTION

In the development of a 2D graph plotter, it's crucial to define clear and comprehensive requirements to ensure that the tool meets user needs and performs effectively. The following sections detail both functional and non-functional requirements for the 2D graph plotter.

3.1 Functional Requirements

Functional requirements describe what the system should do. For a 2D graph plotter, these requirements outline the core functionalities and features necessary to create, customize, and manage 2D plots.

3.1.1 Graph Types

- **Line Plots**: The plotter must support the creation of line plots to visualize trends over time or continuous data.
- **Scatter Plots**: It should enable the creation of scatter plots to display the relationship between two numerical variables.
- **Bar Charts**: The plotter must allow for the generation of bar charts to compare discrete categories or groups.
- **Histograms**: Support for histograms to show the distribution of data across intervals.
- **Pie Charts**: Optional support for pie charts to illustrate the proportions of a whole.

3.1.2 Customization Options

- **Axis Labels:** Users should be able to set and customize labels for the x-axis and y-axis.
- **Titles:** The plotter should allow users to add and modify the title of the graph.
- **Legends:** The tool must provide options to include and customize legends to describe the different elements of the graph.
- **Colors and Styles:** Users should be able to choose colors, line styles, and marker styles to differentiate between data series and enhance visual appeal.
- **Gridlines and Ticks:** The plotter must offer options to include or exclude gridlines and customize tick marks on the axes.

3.1.3 Data Input and Handling

- **Supported Formats:** The plotter should support various data formats such as lists, arrays, and Pandas Data Frames.
- **Data Validation:** The tool must validate data inputs to ensure they are suitable for plotting (e.g., checking for non-numeric values in numeric plots).

3.1.4 Export and Saving

- **File Formats:** Users should be able to export plots in common formats such as PNG, JPEG, PDF, and SVG.
- **Resolution and Size:** The plotter must provide options to adjust the resolution and size of the exported images.

3.1.5 User Interface

- **Ease of Use:** The interface should be intuitive, allowing users to create and customize plots with minimal effort.
- **Interactive Features:** If applicable, the tool should support interactive features such as zooming, panning, and tooltips (for web-based plotters).
- **Help and Documentation:** The plotter should include help options and documentation to assist users in utilizing its features effectively.

3.2 Non-Functional Requirements

Non-functional requirements define how the system performs its functions. These include performance, usability, and reliability criteria.

3.2.1 Performance

- **Speed:** The plotter should generate graphs quickly, even with large datasets, to ensure a smooth user experience.
- **Efficiency:** It should efficiently handle memory and processing resources to avoid crashes or slowdowns, particularly when dealing with complex or large data.

3.2.2 Usability

- **User Experience:** The interface should be user-friendly, with clear navigation and accessible features for both novice and advanced users.
- **Documentation:** Comprehensive user manuals and online help should be provided, including tutorials and examples to guide users in creating and customizing plots.

3.2.3 Reliability

- **Error Handling:** The plotter must handle errors gracefully, providing informative messages when invalid data is entered or when operations fail.
- **Stability:** The tool should be stable and reliable, with minimal bugs or issues that could impact functionality.

3.2.4 Compatibility

- **Operating Systems:** The plotter should be compatible with major operating systems, including Windows, macOS, and Linux.
- **Python Versions:** It must be compatible with commonly used Python versions and environments, ensuring it integrates well with other data analysis tools.

3.2.5 Security

- **Data Privacy:** For online or cloud-based plotters, user data must be protected and handled according to relevant privacy standards and regulations.
- **Code Security:** The software should be free from vulnerabilities that could compromise its integrity or the security of user data.

3.2.6 Scalability

- **Extensibility:** The tool should be designed to accommodate future enhancements or additional features without requiring a complete redesign.
- **Performance with Growth:** It should maintain performance standards as the complexity of data or the number of users increases.

CHAPTER 4

SYSTEM DESIGN

The system design for a 2D graph plotter outlines the architecture, components, and interactions necessary to build a functional and efficient tool for creating and customizing 2D plots. This design covers both the high-level architecture and detailed components, including data handling, user interface, and plotting functionalities.

4.1 High-Level Architecture

The high-level architecture of the 2D graph plotter consists of several key components that work together to deliver the required functionality. The main components include:

1. **User Interface (UI)**
2. **Plotting Engine**
3. **Data Management Module**
4. **Customization Module**
5. **Export/Save Module**

i. User Interface (UI)

The User Interface is the front-end component where users interact with the plotter. It includes:

- **Input Forms**: For data entry, file uploads, or selecting data sources.
- **Plot Configuration**: Tools for choosing plot types, customizing appearance, and setting parameters.
- **Preview Area**: A real-time preview of the plot as users make changes.
- **Controls**: Buttons and menus for saving, exporting, and interacting with the plot.

ii. Plotting Engine

The Plotting Engine is responsible for generating the visual representation of data. It includes:

- **Plotting Libraries**: Utilizes libraries like Matplotlib, Seaborn, or Plotly to create plots based on user specifications.
- **Rendering Engine**: Handles the actual drawing of graphs and managing visual elements.

iii. Data Management Module

The Data Management Module deals with data input, validation, and processing. It includes:

- **Data Import**: Capabilities for importing data from various formats (e.g., CSV, Excel, JSON).
- **Data Validation**: Ensures the data is suitable for plotting (e.g., correct types, non-null values).
- **Data Transformation**: Prepares data for visualization, including sorting, filtering, and aggregation.

iv. Customization Module

The Customization Module provides options for users to modify the appearance and attributes of the plots. It includes:

- **Graph Types**: Allows selection of different types of plots (e.g., line, scatter, bar).
- **Style Options**: Enables changes to colors, markers, lines, and other visual elements.
- **Axis Configuration**: Customizes axis labels, titles, scales, and ticks.
- **Legend and Gridlines**: Configures the display of legends, gridlines, and annotations.

v. Export/Save Module

The Export/Save Module manages the output and storage of plots. It includes:

- **File Formats**: Supports saving plots in various formats such as PNG, JPEG, PDF, and SVG.
- **Resolution Settings**: Allows users to adjust the resolution and size of exported files.
- **File Management**: Handles file naming, saving locations, and user access.

4.2 Detailed Design

i. User Interface (UI)

Design Considerations:

- **User Experience (UX)**: The UI should be intuitive and responsive, providing clear instructions and feedback.
- **Consistency**: Maintain a consistent look and feel across all components.
- **Accessibility**: Ensure the UI is accessible to users with disabilities, including support for screen readers and keyboard navigation.

Components:

- **Main Window**: Contains the primary workspace where plots are displayed.
- **Toolbars**: Provide quick access to plot types, customization options, and export features.
- **Dialogs**: Used for data input, configuration settings, and error messages.

ii. Plotting Engine

Design Considerations:

- **Library Selection**: Choose appropriate libraries based on functionality and ease of integration.
- **Performance**: Optimize rendering to handle large datasets and complex plots efficiently.

Components:

- **Plot Generator**: Interfaces with plotting libraries to create graphs based on user input.
- **Renderer**: Manages the display of plots and updates the view in real-time as changes are made.

iii. Data Management Module

Design Considerations:

- **Data Integrity**: Ensure data is accurately processed and validated.
- **Scalability**: Handle large datasets without performance degradation.

Components:

- **Data Importer**: Reads data from files or other sources and converts it into a format suitable for plotting.
- **Data Validator**: Checks for errors and inconsistencies in the data.
- **Data Processor**: Performs necessary transformations to prepare the data for visualization.

iv. Customization Module

Design Considerations:

- **Flexibility**: Provide extensive options for customization to meet diverse user needs.
- **User Control**: Allow users to easily adjust and preview changes.

Components:

- **Customization Panel**: Interface elements for modifying plot attributes, including colors, styles, and labels.
- **Preview Function**: Displays a real-time update of the plot as customization options are

adjusted.

v. Export/Save Module

Design Considerations:

- **File Format Support**: Ensure compatibility with commonly used file formats.
- **User Preferences**: Allow users to define default settings for export.

Components:

- **Export Dialog**: Provides options for selecting file format, resolution, and save location.
- **File Handler**: Manages file creation, naming, and saving processes.

4.3 System Flow

➢ **Workflow Overview**

1. **Data Input**: Users import data through the UI, which is processed by the Data Management Module.
2. **Plot Selection**: Users choose the type of plot they want to create from the Plotting Engine.
3. **Customization**: Users configure plot attributes and preview changes in real-time.
4. **Rendering**: The Plotting Engine generates the plot based on the user's specifications.
5. **Export/Save**: Users save or export the plot in their desired format.

➢ **Interaction Diagram**

1. **User → UI**: Interacts with the UI to input data and configure plots.
2. **UI → Data Management Module**: Sends data for validation and processing.
3. **Data Management Module → Plotting Engine**: Provides processed data for plotting.
4. **Plotting Engine → UI**: Updates the plot preview based on customization.
5. **User → UI**: Finalizes the plot and requests export.
6. **UI → Export/Save Module**: Handles the export and saving of the plot.

4.4 Deployment Considerations

i. Platform Compatibility

- **Cross-Platform**: Ensure the tool works on Windows, macOS, and Linux.
- **Dependencies**: Manage and include necessary libraries and dependencies for smooth installation and operation.

ii. Performance Optimization

- **Efficiency**: Optimize the plotting process to handle large datasets efficiently.
- **Resource Management**: Minimize memory and CPU usage to ensure smooth operation.

iii. Security

- **Data Privacy**: For web-based applications, ensure user data is protected.
- **Code Security**: Implement best practices to avoid vulnerabilities and ensure code integrity.

CHAPTER 5

SYSTEM TESTING

System testing is a critical phase in the development of a 2D graph plotter. It ensures that the tool meets its specified requirements, performs correctly under various conditions, and delivers a reliable and user-friendly experience. This section covers the types of testing to be conducted, testing strategies, and specific test cases relevant to a 2D graph plotter.

5.1 Types of Testing

i. Functional Testing

Objective: Verify that all functionalities of the 2D graph plotter work as intended and meet the specified requirements.

Test Cases:

- **Plot Generation**: Test the creation of different types of plots (line, scatter, bar, histogram, pie) to ensure they are generated correctly.
- **Data Handling**: Verify that the plotter correctly imports and processes data from various formats (e.g., CSV, Excel).
- **Customization**: Ensure that customization options (colors, styles, labels) are applied correctly and reflected in the generated plots.
- **Export/Save**: Test the export functionality to ensure plots are saved in various formats (PNG, JPEG, PDF, SVG) with the correct resolution and size.

ii. Usability Testing

Objective: Evaluate the user interface and experience to ensure the tool is intuitive and user-friendly.

Test Cases:

- **Ease of Use**: Assess whether users can easily navigate the UI and access the plotting features.
- **Help and Documentation**: Check if the help options and documentation are sufficient and accessible.
- **Interactive Features**: For interactive plots, verify that zooming, panning, and tooltips function correctly.

iii. Performance Testing

Objective: Determine how well the plotter performs under various conditions, including different dataset sizes and system configurations.

Test Cases:

- **Speed**: Measure the time taken to generate plots with small, medium, and large datasets.
- **Resource Utilization**: Monitor CPU and memory usage during plot generation to ensure efficient resource management.
- **Scalability**: Test the plotter's performance with increasing complexity and size of data.

iv. Compatibility Testing

Objective: Ensure the plotter functions correctly across different operating systems and Python environments.

Test Cases:

- **Operating Systems**: Verify compatibility with Windows, macOS, and Linux.
- **Python Versions**: Test the tool with different Python versions and ensure it integrates well with libraries and dependencies.

v. Security Testing

Objective: Identify and address potential security vulnerabilities, especially for web-based or cloud-based plotters.

Test Cases:

- **Data Privacy**: For web-based applications, ensure that user data is protected according to privacy standards.
- **Code Security**: Check for common vulnerabilities and ensure secure coding practices are followed.

vi. Regression Testing

Objective: Ensure that new changes or enhancements do not adversely affect existing functionalities.

Test Cases:

- **Feature Testing**: Re-test all functionalities after any code changes or updates to ensure they still work as intended.
- **Bug Fixes**: Verify that previously reported bugs are resolved and have not reappeared.

5.2 Testing Strategies

i. Unit Testing

Objective: Test individual components of the system in isolation to ensure each part functions correctly.

Components to Test:

- **Plotting Functions**: Test individual functions or methods responsible for generating different types of plots.
- **Data Processing**: Validate functions involved in data import, validation, and transformation.

Tools:

- **pytest** or **unittest** for Python-based unit testing.

ii. Integration Testing

Objective: Verify that different components of the system work together as expected.

Components to Test:

- **Data Handling and Plotting**: Ensure that data flows correctly from the Data Management Module to the Plotting Engine.
- **UI and Plotting Engine**: Test interactions between the UI and the Plotting Engine to ensure user inputs are correctly reflected in the plots.

Tools:

- **Integration Testing Frameworks**: Use tools like **pytest** or **Selenium** for end-to-end testing.

iii. System Testing

Objective: Test the entire system as a whole to ensure it meets the specified requirements and performs correctly in real-world scenarios.

Components to Test:

- **End-to-End Functionality**: Test the complete workflow from data import to plot generation and export.
- **User Interaction**: Evaluate overall user experience and interface usability.

Tools:

- **Manual Testing**: Perform hands-on testing of the entire system.
- **Automated Testing**: Use tools like **Selenium** for automated UI testing.

iv. Acceptance Testing

Objective: Validate the system against user requirements and expectations to determine if it is ready for deployment.

Components to Test:

- **Functional Requirements**: Ensure all specified functionalities are working as required.
- **Usability**: Confirm that the tool meets user expectations in terms of ease of use and performance.

Tools:

- **User Feedback**: Collect feedback from actual users to assess satisfaction and usability.
- **Acceptance Test Scripts**: Develop and execute test scripts based on user requirements.

5.3 Test Case Examples

i. Test Case: Line Plot Generation

- **Objective**: Verify that a line plot is generated correctly.
- **Preconditions**: Data is available in a supported format.
- **Steps**:
    1. Import the data into the plotter.
    2. Select "Line Plot" from the plot types.
    3. Configure axis labels and title.
    4. Generate the plot.
- **Expected Result**: A line plot is displayed with the correct data, labels, and title.

ii. Test Case: Customization Options

- **Objective**: Ensure customization options are applied correctly.
- **Preconditions**: A plot is generated.
- **Steps**:
    1. Select a plot to customize.
    2. Change the color and style of the lines or markers.
    3. Adjust axis labels and add a legend.
- **Expected Result**: The plot updates to reflect the changes in color, style, labels, and legend.

iii. Test Case: Export Functionality

- **Objective**: Test the export functionality for different file formats.
- **Preconditions**: A plot is generated and ready for export.
- **Steps**:
    1. Choose the "Export" option.
    2. Select PNG format and adjust resolution settings.
    3. Save the file and verify the output.
- **Expected Result**: The plot is saved as a PNG file with the specified resolution and appears correctly.

CHAPTER 6

CODE

2D Graph plotter.py

```python
import tkinter as tk
from tkinter import ttk
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
import numpy as np

class GraphPlotter(tk.Tk):
    def __init__(self):
        super().__init__()

        self.title("Interactive Graph Plotter")
        self.geometry("800x600")

        self.func_entry = tk.Entry(self, width=50)
        self.func_entry.pack(pady=10)
        self.func_entry.insert(0, "np.sin(x)")

        self.plot_button = tk.Button(self, text="Plot Graph", command=self.plot_graph)
        self.plot_button.pack(pady=10)

        self.figure, self.ax = plt.subplots()
        self.canvas = FigureCanvasTkAgg(self.figure, master=self)
        self.canvas.draw()
        self.canvas.get_tk_widget().pack(fill=tk.BOTH, expand=True)

        self.bind("<MouseWheel>", self.zoom)
        self.canvas.mpl_connect("button_press_event", self.on_press)
        self.canvas.mpl_connect("motion_notify_event", self.on_move)
        self.canvas.mpl_connect("button_release_event", self.on_release)

        self.pan_active = False
        self.press = None
```

```python
        self.pan_active = False
        self.press = None

    def plot_graph(self):
        func_str = self.func_entry.get()
        x = np.linspace(-10, 10, 1000)

        try:
            y = eval(func_str)
            self.ax.clear()
            self.ax.plot(x, y)
            self.canvas.draw()
        except Exception as e:
            print(f"Error in function: {e}")

    def zoom(self, event):
        base_scale = 1.1
        cur_xlim = self.ax.get_xlim()
        cur_ylim = self.ax.get_ylim()
        xdata = event.xdata
        ydata = event.ydata

        if event.delta > 0:
            scale_factor = 1 / base_scale
        else:
            scale_factor = base_scale
```

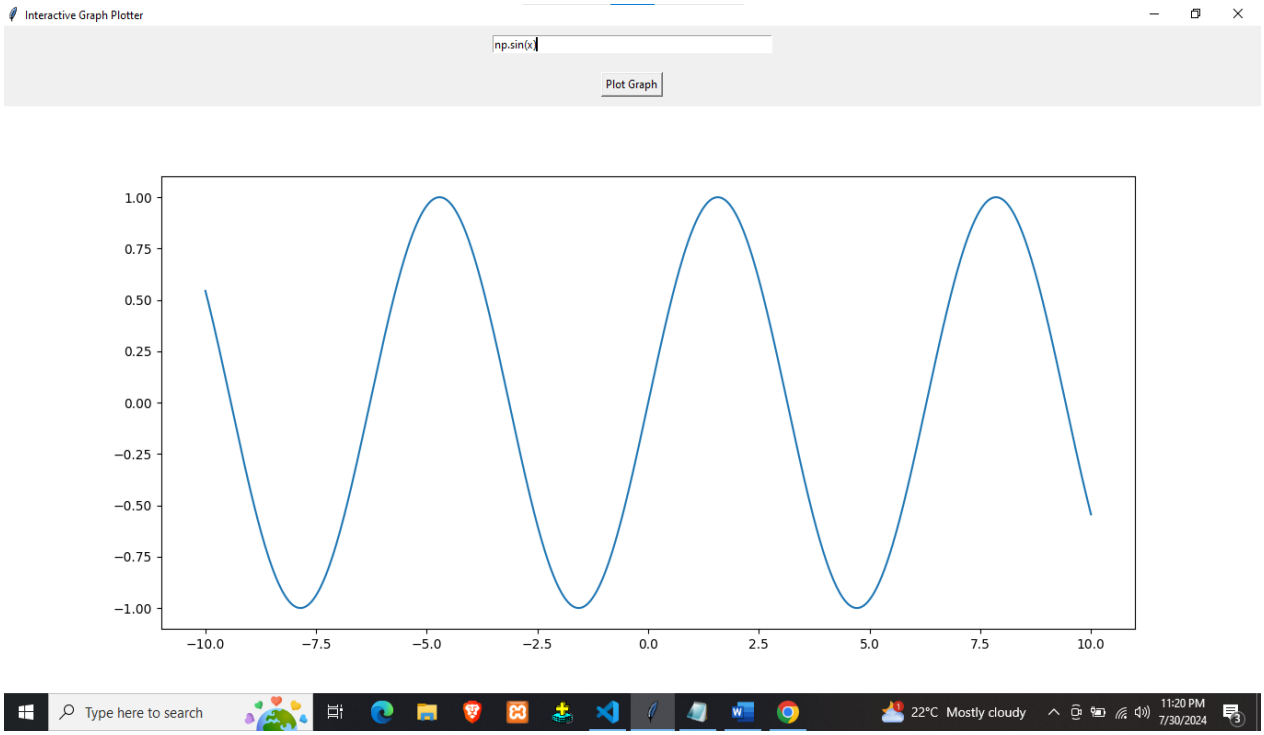
```
            scale_factor = base_scale

        new_width = (cur_xlim[1] - cur_xlim[0]) * scale_factor
        new_height = (cur_ylim[1] - cur_ylim[0]) * scale_factor

        relx = (cur_xlim[1] - xdata) / (cur_xlim[1] - cur_xlim[0])
        rely = (cur_ylim[1] - ydata) / (cur_ylim[1] - cur_ylim[0])

        self.ax.set_xlim([xdata - new_width * (1 - relx), xdata + new_width * relx])
        self.ax.set_ylim([ydata - new_height * (1 - rely), ydata + new_height * rely])

        self.canvas.draw()

    def on_press(self, event):
        if event.button == 1:
            self.pan_active = True
            self.press = event.xdata, event.ydata

    def on_move(self, event):
        if self.pan_active:
            dx = event.xdata - self.press[0]
            dy = event.ydata - self.press[1]
            self.press = event.xdata, event.ydata

            cur_xlim = self.ax.get_xlim()
            cur_ylim = self.ax.get_ylim()

            self.ax.set_xlim(cur_xlim[0] - dx, cur_xlim[1] - dx)
            self.ax.set_ylim(cur_ylim[0] - dy, cur_ylim[1] - dy)
            self.canvas.draw()
```

```
            self.ax.set_ylim(cur_ylim[0] - dy, cur_ylim[1] - dy)
            self.canvas.draw()

    def on_release(self, event):
        self.pan_active = False
        self.press = None

if __name__ == "__main__":
    app = GraphPlotter()
    app.mainloop()
```

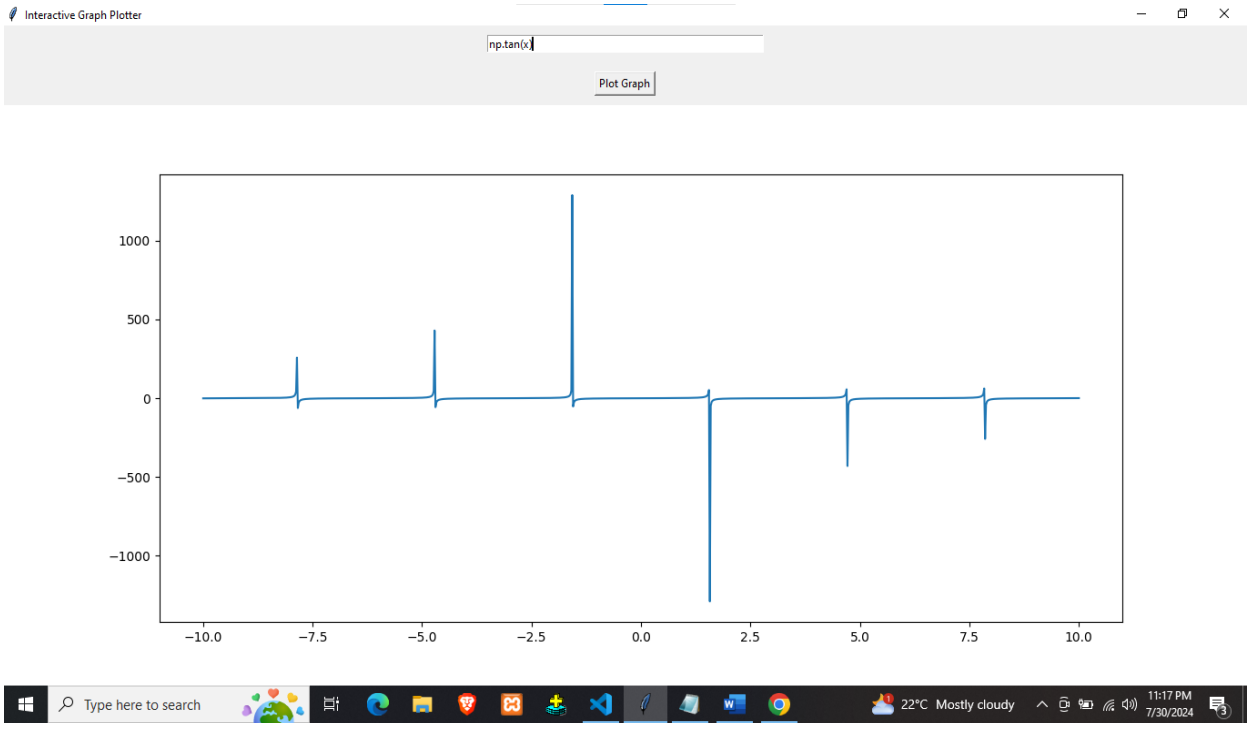
CHAPTER 7

SNAPSHOTS

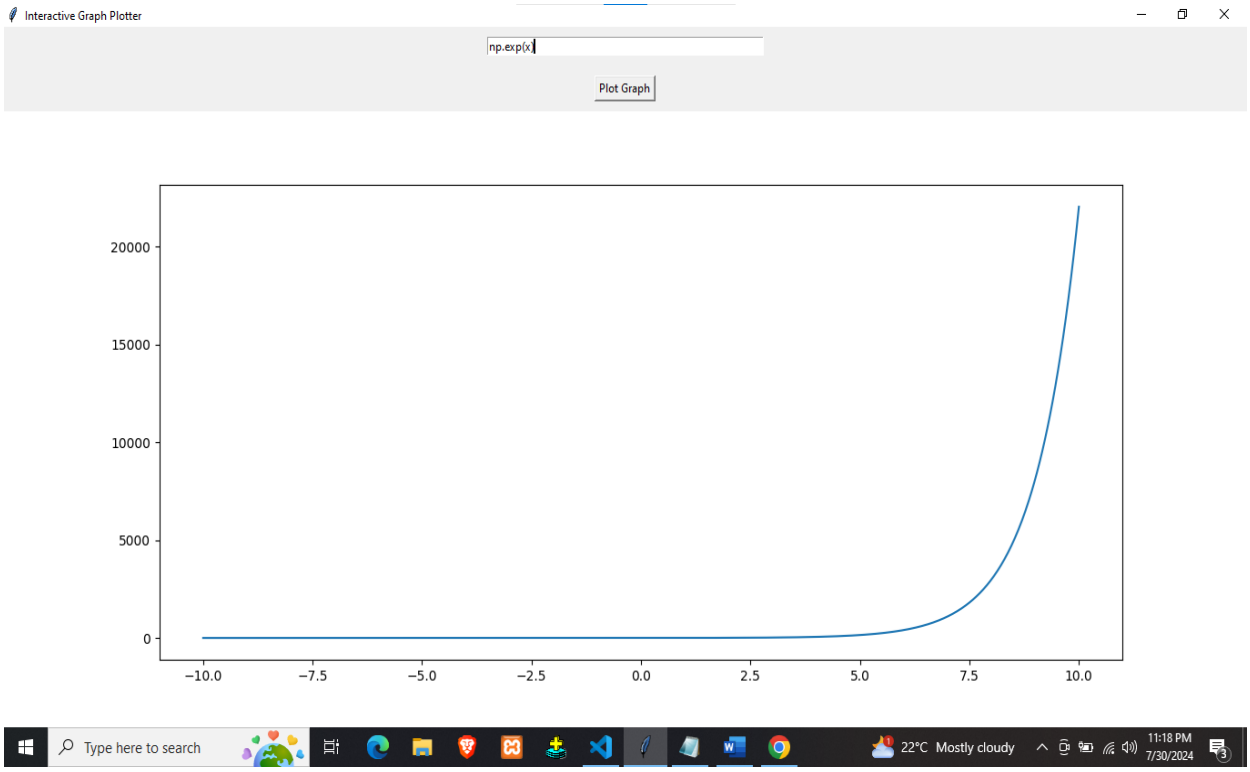
Equation: np.sin(x)



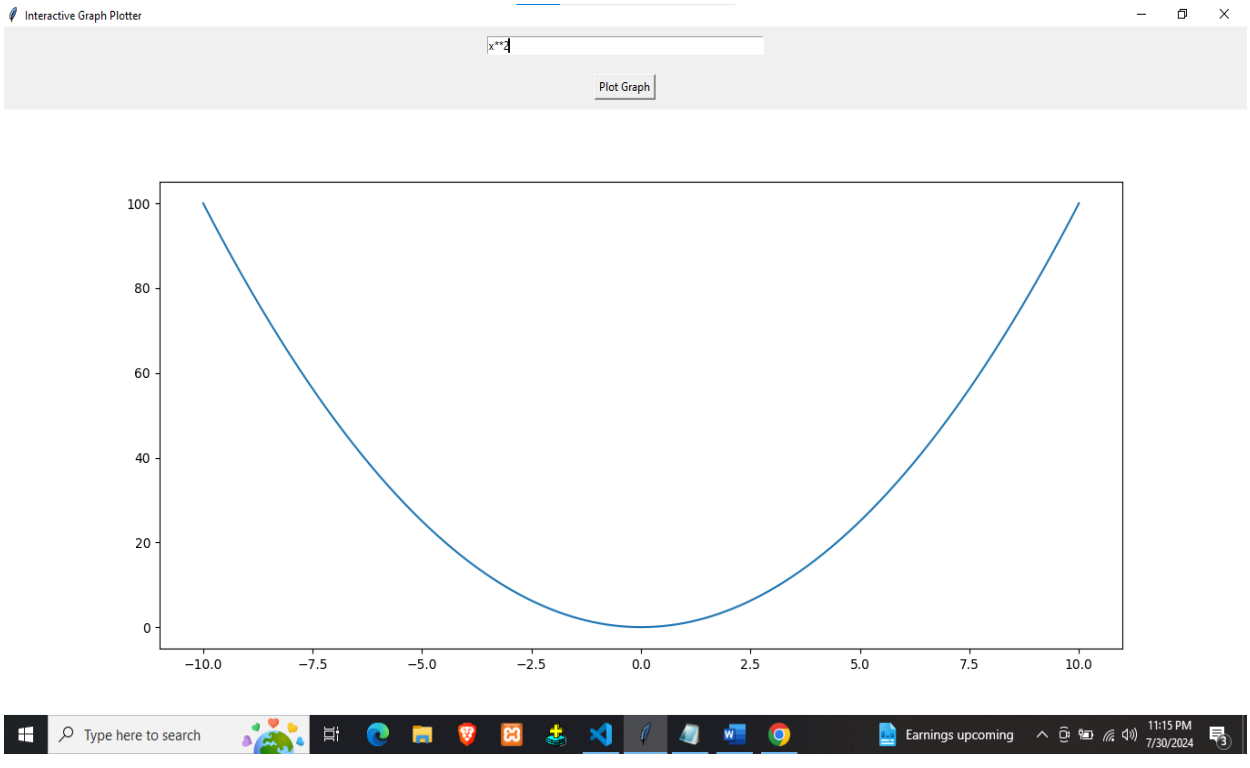
Equation: np.tan(x)



Equation: np.exp(x)



Equation: x**2



CONCLUSION

The color identification system developed through this project effectively demonstrates the integration of image processing, data management, and user interaction functionalities using Python and its powerful libraries: OpenCV, pandas, and NumPy. This project successfully addresses the need for an intuitive tool that allows users to identify colors within images accurately.

Key Achievements

1. **Functional Integration**: The project seamlessly combines image loading, user interaction, and data processing, providing a cohesive tool that performs reliably across different platforms.
2. **User Interaction**: The system offers an interactive interface where users can double-click on any part of an image to retrieve and display the color name and its RGB values. This feature enhances the educational value of the tool.
3. **Accurate Color Matching**: The color matching algorithm ensures precise identification by calculating the minimum distance between the selected pixel's RGB values and the dataset, demonstrating the effectiveness of the implemented algorithm.
4. **Readability and Usability**: The project prioritizes user experience by ensuring that the displayed color information is readable against varying backgrounds, and by making the system user-friendly for individuals with basic computer skills.

REFERENCES

- Introduction to visual studio Microsoft.com Wikipedia.com
- pictures/images Google Image Search
- Python references from www.python.org
- https://www.w3schools.com/html/
- https://www.w3schools.com/css/
- https://itsourcecode.com/free-projects/opencv/color-detection-opencv-python-with-source-code/?source=post_page-----bedfa8165c3b----------
- https://chatgpt.com/c/f3800394-1ab5-4d2d-a670-d216555f1f7a
- "Python Plotting With Matplotlib: A Guide." [Online]. Available: https://realpython.com/python-matplotlib-guide/
- pandas Development Team. (n.d.). pandas: Powerful data structures for data analysis, time series, and statistics. Retrieved from https://pandas.pydata.org/