CIE 327 Probability Project

StatViz.py

SalahDin Ahmed Salh Rezk 202201079
Marwan Bassem Ragaa 202200776

December 22, 2024

Abstract

This report describes StatViz.py, a Python application that offers a graphical user interface for analyzing random variables. The tool calculates and visualizes probability distributions, moments, moment generating functions (MGFs), covariance, and correlation for single and joint random variables. It supports both single and joint random variable analysis, including derived variables. The system uses matplotlib for visualizations and numpy for numerical computation and sympy for symbolic mathematics when dealing with functions of random variables. Test results and sample data are presented to demonstrate the system's functionality as an aid in statistical education and project work.

Contents

List of Figures

1 Introduction

StatViz.py is a Python-based tool developed to offer a user-friendly graphical interface to analyze random variables and their statistical properties. The application allows users to visualize and compute key statistical measures for single and joint random variables, including probability distributions, moments, moment generating functions (MGFs), covariance, and correlation.

It supports both single random variable analysis and joint random variable analysis, providing plots such as CDFs, marginal distributions, and visualizations of derived variables. The tool is designed to help students and practitioners deepen their understanding of probability theory through interactive and dynamic visualizations. Additionally, it might proof helpful for teams working on simple statistical projects. Test cases and example files are provided to demonstrate various random variable distributions, as the tool is a valuable resource for learning and experimentation.

2 Code Overview

The system is organized into multiple Python modules that handle different aspects of statistical analysis, visualization, and data generation. The project structure is as follows:

- **ui/**: Contains the user interface components, including the main window, dialog boxes, and plot widgets.
- **src/**: The main script that launches the graphical user interface (GUI) and handles user interactions.
  - **requirements.txt**: Contains the required Python packages for the project.
  - **poetry.lock**: Contains the dependencies and their versions.
  - **pyproject.toml**: Contains the project's metadata and dependencies.
  - **statviz/**: Contains the project's metadata and dependencies.
    - **___main___.py**: The main script that launches the GUI.
    - **analysis/**: Contains functions for analyzing random variables, including computing moments, MGFs, covariance, and correlation.
      - **single_random_variable.py**
      - **joint_random_variables.py**
      - **functions_of_random_variables.py**
      - **test_generator.py**
      - **utils.py**
      - **helpers.py**
    - **gui/**: Contains the programming logic for the GUI.
- **samples/**: Contains sample files for testing the tool, including single and joint random variables, this directory is git ignored as it contains large files.

2.1 Analysis Module

All the statistical analysis functions are implemented in the **analysis** module. The module is divided into three submodules: **single random variable**, **joint random variables**, and **functions of random variables**. Each submodule contains functions for analyzing random variables, computing moments, MGFs, covariance, and correlation.

Figures are generated using the **matplotlib** library, which provides a wide range of plotting functions. Most of them are histograms, bar plots, and line plots, which are used to visualize the probability distribution, CDF, and moments of random variables. Bin width is a critical parameter in histogram construction and probability density estimation. It represents the size of the intervals into which the data range is divided when creating histograms or empirical probability distributions. The choice of bin width significantly impacts the visualization and interpretation of the data.

The system implements Scott's Rule for automatic bin width determination:

$$h = \frac{3.5 \times \text{std}(x)}{n^{1/3}} \tag{1}$$

where h is the bin width, std(x) is the standard deviation of the data, and n is the number of samples. This rule is widely used in statistics and data visualization to determine the optimal bin width for histograms. It provides a good balance between capturing the underlying data distribution and avoiding excessive noise or oversmoothing. The bin width affects the visual appearance of the plot and the interpretation of the data. Additionally, it can impact the interpretation of the data distribution and the detection of patterns or trends.

The codebase implements bin width calculation in several key areas:

- **Histograms**: The bin width is used to determine the size of the intervals in the histogram, which represent the frequency of data points in each range.

```
_, bins = np.histogram(X, bins="scott", density=True)
```

- **Probability Density Estimation**: The bin width is used to estimate the probability density function of the data, which provides a continuous representation of the data distribution.

```
      H, xedges, yedges = np.histogram2d(X, Y, density=True, bins=(bins_X,
      ↪  bins_Y))
```

Moreover, all analysis modules are designed to take output I/O stream as an argument to allow for easy integration with the GUI. The default output stream is `sys.stdout`. However, it can be set to a file, a variable, or a custom stream for logging or debugging purposes.

```
def main(stream: TextIO = stdout):
    # Code here
    stream.write("\n=== Results ===\n")
    stream.write("\nStatistical Measures:\n")
    stream.write(f"Mean = {mean_X:.4f}\n")
    stream.write(f"Variance = {var_X:.4f}\n")
    stream.write(f"Third Moment = {third_moment:.4f}\n")
    # Remaining code
```

This design choice allows for flexible output handling and enables users to redirect the output to different destinations based on their requirements. For example, users can display the results in the console, save them to a file, or store them in a variable for further processing.

The module ensures numerical stability and precision through consistent use of float64 data types. The visualization components are designed to be both informative and aesthetically pleasing, with careful attention to color choices, transparency, and grid lines to enhance readability.

2.2 Helpers Module

The helpers module serves as a foundational component of the analysis system, providing essential utility functions for data manipulation, statistical calculations, and file operations. This module emphasizes numerical precision and efficient computation through consistent use of NumPy's float64 data type and vectorized operations. A key feature of the module is its robust file handling capabilities. The `read_file_joint` function is designed to safely load joint random variable data from MATLAB .mat files:

```
def read_file_joint(filename: str) -> tuple[np.ndarray, np.ndarray]:
"""
Reads sample pairs (X, Y) from a MATLAB .mat file.
Returns arrays with float64 dtype.
"""
try:
data = loadmat(filename).get("XY")
X = data[0, :].astype(np.float64)
```

```
Y = data[1, :].astype(np.float64)
if X.size == 0 or Y.size == 0:
raise ValueError("The .mat file must contain 'X' and 'Y' variables.")
if len(X) != len(Y):
raise ValueError("The number of samples in 'X' and 'Y' do not match.")
return X, Y
except Exception as e:
print(f"Error: {e}")
exit(1)
```

The module implements several critical statistical computations. The `calc_joint_prob` function efficiently calculates joint probability distributions using NumPy's advanced indexing capabilities:

```
def calc_joint_prob(X: np.ndarray, Y: np.ndarray) -> np.ndarray:
X = X.astype(np.float64)
Y = Y.astype(np.float64)
Xuq, X_inv = np.unique(X, return_inverse=True)
Yuq, Y_inv = np.unique(Y, return_inverse=True)
Nxy = np.zeros((len(Xuq), len(Yuq)), dtype=np.float64)
np.add.at(Nxy, (X_inv, Y_inv), 1)
Pxy = Nxy / len(X)
return Pxy
```

For statistical analysis, the module provides functions to compute correlation and covariance:

- `calc_cov`: Computes the covariance between two random variables using their joint probability distribution
- `calc_cor`: Calculates the correlation coefficient, utilizing the previously computed covariance
- `calc_stats`: Determines basic statistical measures including mean, variance, and third moment
- `calc_marg_prob`: Computes marginal probability distributions from joint distributions

The module also implements Scott's Rule for optimal bin width calculation in histogram generation:

```
def calc_bin_width(Xuq: np.ndarray) -> float:
"""
Calculate bin width using Scott's Rule.
Xuq: unique values
Returns: bin width
"""
n = len(Xuq)
std = np.std(Xuq)
bin_width = 3.5 * std / (n ** (1 / 3))
return bin_width
```

All functions in the module maintain numerical precision through consistent use of float64 data types and employ vectorized operations where possible to ensure computational efficiency. Error handling is implemented throughout to provide meaningful feedback when processing invalid inputs or encountering computational issues.

2.3 Single Random Variable

The **single random variable** submodule contains functions for analyzing single random variables, including computing moments, MGFs, and visualizing probability distributions. The module provides a set of functions for generating random variables from various distributions, such as the Poisson, Binomial, and Normal distributions. These functions are used to create sample data for testing and demonstration purposes.

The module also contains functions for computing the moments of random variables, including the mean, variance, and skewness. These moments provide key insights into the shape and properties of the data distribution. The module uses the **numpy** library for numerical computations and statistical analysis.

```python
def calc_mgf_deriv(
    X: np.ndarray, P: np.ndarray, t_max: float
) -> tuple[np.ndarray, np.ndarray, np.ndarray]:
    """Calculates the Moment Generating Function (MGF) and its derivatives."""
    Xuq = np.unique(X)
    t_values = np.linspace(0, t_max, N)
    MGF = np.array([np.sum(np.exp(t * Xuq) * P) for t in t_values])
    MGF_prime = np.array([np.sum(Xuq * np.exp(t * Xuq) * P) for t in t_values])
    MGF_double_prime = np.array([np.sum(Xuq**2 * np.exp(t * Xuq) * P) for t in
    ↪  t_values])
    return MGF, MGF_prime, MGF_double_prime
```

This script has been designed to streamline the analysis of random variables stored in MATLAB .mat files. These files are expected to contain at least one key variable, X, which represents the sample space of the random variable. The script reads the data from the .mat file and computes the statistical measures of the random variable, mentioned above. The script also generates plots of the probability distribution, CDF, and moments of the random variable.

```python
def read_file_single(filename: str) -> np.ndarray:
    """
    Reads a MATLAB .mat file and returns sample space X and probabilities P.
    File format:
        The .mat file should contain two variables:
            - 'X': array of sample space values
            - 'P': array of corresponding probabilities
    """
    try:
        data = loadmat(filename)
        X = np.array(
            data.get("X", []), dtype=np.float64
        )  # Get 'X', default to empty array if not found

        if X.size == 0:
            raise ValueError("The .mat file must contain 'X' variable.")
        return X
    except Exception as e:
        print(f"Error reading the file: {e}")
        exit(1)
```

By leveraging `numpy`'s efficient array processing capabilities (as it is mainly written in C/C++ instead of Python), the function determines the frequencies of unique elements in the input array X. These frequencies are then normalized to calculate probabilities, ensuring the resulting array aligns with statistical conventions where probabilities sum to one. This normalization step is critical for ensuring the correctness of subsequent statistical analyses and visualizations.

```python
def calc_prob(X: np.ndarray) -> np.ndarray:
    """
    Calculates and normalizes probabilities of unique elements in the input array
    ↪  X.

    Parameters:
        X (np.ndarray): Input array.
```

```
    Returns:
        np.ndarray: Normalized probabilities corresponding to unique elements in X.
    """
    if len(X) == 0:
        return np.array([])  # Return an empty array for empty input

    _, counts = np.unique(X, return_counts=True)
    P = np.array(
        counts / counts.sum(), dtype=np.float64
    )  # Explicit normalization step
    return P
```

The `plot_prob_cdf` function provides a dual representation of the data in the form of a histogram (probability distribution) and a stepwise CDF plot. The inclusion of grid lines, labels, and well-structured layouts enhances the clarity and interpretability of the plots. Similarly, the `plot_mgf_deriv` function visualizes the MGF and its first two derivatives across a specified range of t values. The three plots generated—one each for the MGF, its first derivative, and its second derivative—offer valuable insights into the moments of the distribution, including the mean (derivable from the first derivative) and variance (derivable from the second derivative). These plots are meticulously crafted with distinct titles, axes labels, and gridlines, making them highly informative for users aiming to analyze the data's behavior across varying t.

```
def plot_prob_cdf(X: np.ndarray, P: np.ndarray) -> None:
    """
    Plots the probability distribution (as a histogram) and cumulative distribution
    ↪  function (CDF)
    with automatic bin width calculation.
    """
    Xuq = np.unique(X)
    CDF = np.cumsum(P)

    # Calculate automatic bin width using Scott's Rule
    _, bins = np.histogram(X, bins="scott", density=True)

    plt.figure(figsize=(10, 6))

    # Plot the histogram
    plt.subplot(2, 1, 1)
    plt.hist(Xuq, bins=bins, weights=P, edgecolor="black", align="mid", rwidth=0.9)
    plt.title("Probability Distribution")
    plt.xlabel("Sample Space")
    plt.ylabel("Probability")
    plt.grid(True)

    # Plot the CDF
    plt.subplot(2, 1, 2)
    plt.step(Xuq, CDF, where="post", linewidth=2)
    plt.title("Cumulative Distribution Function (CDF)")
    plt.xlabel("Sample Space")
    plt.ylabel("Cumulative Probability")
    plt.grid(True)

    plt.tight_layout()
    plt.show(block=False)


```

```python
def plot_mgf_deriv(
    MGF: np.ndarray,
    MGF_prime: np.ndarray,
    MGF_double_prime: np.ndarray,
    t_max: float,
) -> None:
    """Plots the Moment Generating Function (MGF) and its derivatives."""
    t_values = np.linspace(0, t_max, N)
    plt.figure(figsize=(10, 8))

    plt.subplot(3, 1, 1)
    plt.plot(t_values, MGF, linewidth=2)
    plt.title("Moment Generating Function (MGF)")
    plt.xlabel("t")
    plt.ylabel("M(t)")
    plt.grid(True)

    plt.subplot(3, 1, 2)
    plt.plot(t_values, MGF_prime, linewidth=2)
    plt.title("First Derivative of MGF (M'(t))")
    plt.xlabel("t")
    plt.ylabel("M'(t)")
    plt.grid(True)

    plt.subplot(3, 1, 3)
    plt.plot(t_values, MGF_double_prime, linewidth=2)
    plt.title("Second Derivative of MGF (M''(t))")
    plt.xlabel("t")
    plt.ylabel("M''(t)")
    plt.grid(True)

    plt.tight_layout()
    plt.show(block=False)
```

2.4 Joint Random Variables

The core visualization capabilities are implemented through two main plotting functions. The first, `plot_marg_prob`, creates side-by-side histograms of the marginal distributions. Matplotlib's `hist` function is used to generate the histograms, with the bin edges calculated using Scott's Rule. The histograms are plotted with distinct colors and transparency to enhance readability, and grid lines are included to guide the viewer's eye.

```python
def plot_marg_prob(
    X: np.ndarray, Y: np.ndarray, Px: np.ndarray, Py: np.ndarray
) -> None:
    """
    Plot marginal distributions using float64 arrays.
    """
    X = X.astype(np.float64)
    Y = Y.astype(np.float64)
    Px = Px.astype(np.float64)
    Py = Py.astype(np.float64)

    Xuq = np.unique(X)
    Yuq = np.unique(Y)

```

```
    _, (ax2, ax3) = plt.subplots(1, 2, figsize=(12, 5))

    _, bins_X = np.histogram(X, bins="scott", density=True)
    _, bins_Y = np.histogram(Y, bins="scott", density=True)

    # Plot X marginal distribution
    ax2.hist(
        Xuq,
        bins=bins_X,
        weights=Px,
        edgecolor="black",
        align="mid",
        rwidth=0.9,
        color="blue",
        alpha=0.7,
    )
    ax2.set_title("Marginal Distribution P(X)")
    ax2.set_xlabel("X")
    ax2.set_ylabel("Probability")
    ax2.grid(True)

    # Plot Y marginal distribution
    ax3.hist(
        Yuq,
        bins=bins_Y,
        weights=Py,
        edgecolor="black",
        align="mid",
        rwidth=0.9,
        color="orange",
        alpha=0.7,
    )
    ax3.set_title("Marginal Distribution P(Y)")
    ax3.set_xlabel("Y")
    ax3.set_ylabel("Probability")
    ax3.grid(True)
```

The second key visualization function, `plot_joint_prob`, creates a three-dimensional representation of the joint probability distribution using Matplotlib's 3D capabilities. The function creates bin edges for X and Y using numpy's builtin Scott's Rule, then generates a 2D histogram of the joint distribution using the `histogram2d` function. The calculation of `xpos` and `ypos` creates the grid of positions for the bars, while `dz` contains the heights of the bars.

```
def plot_joint_prob(X: np.ndarray, Y: np.ndarray, Pxy: np.ndarray) -> None:
    """
    Plot joint probability distribution using float64 arrays.
    Bars are sized according to the bin widths for better visualization.
    """
    _, bins_X = np.histogram(X, bins="scott", density=True)
    _, bins_Y = np.histogram(Y, bins="scott", density=True)

    H, xedges, yedges = np.histogram2d(X, Y, density=True, bins=(bins_X, bins_Y))

    xpos, ypos = np.meshgrid(xedges[:-1], yedges[:-1], indexing="ij")
    xpos = xpos.ravel()
    ypos = ypos.ravel()
    zpos = np.zeros_like(xpos)
```

```
    dx = np.diff(xedges)[0]
    dy = np.diff(yedges)[0]
    dz = H.ravel()
```

2.5 Functions of Random Variables

The functions of random variables module extends the analysis capabilities to handle transformations of random variables through arbitrary mathematical functions. This module combines symbolic mathematics using SymPy with numerical computations using NumPy to provide flexible analysis of derived random variables.

A key feature of the module is the `matheval` function, which safely converts string-based mathematical expressions into callable functions:

```
def matheval(expr: str, variables: list) -> Callable:
    """
    Safely evaluate a mathematical expression provided as a string using sympy.
    expr: String representation of the expression.
    valid_symbols: List of valid symbols.
    """
    try:
        locals = {name: symbols(name) for name in variables}
        expr = sympify(expr, locals=locals)
        func = lambdify(variables, expr, modules="numpy")
        return func
    except Exception as e:
        raise ValueError(f"Error evaluating expression '{expr}': {e}")
```

This function provides several important capabilities:

1. Conversion of string expressions into symbolic mathematical expressions
2. Validation of mathematical expressions for safety
3. Creation of efficient NumPy-based callable functions
4. Support for arbitrary variable names and mathematical operations

The module implements visualization functions similar to those in the joint random variables module, but adapted for transformed variables Z and W:

```
def plot_marg_prob(
    Z: np.ndarray, W: np.ndarray, Pz: np.ndarray, Pw: np.ndarray
) -> None:
    """
    Plot marginal distributions using float64 arrays.
    """
    Z = Z.astype(np.float64)
    W = W.astype(np.float64)
    Pz = Pz.astype(np.float64)
    Pw = Pw.astype(np.float64)

    Zuq = np.unique(Z)
    Wuq = np.unique(W)

    _, (ax2, ax3) = plt.subplots(1, 2, figsize=(12, 5))
    # ... plotting implementation
```

The main program flow is controlled through the `main` function, which introduces several additional features compared to the basic joint analysis:

```
def main(stream: TextIO = stdout):
    args = handle_args()
    filename = args.filename
    X, Y = read_file_joint(filename)

    # Define functions for Z and W
    z_func = matheval(args.Z_func, ["x", "y"])
    w_func = matheval(args.W_func, ["x", "y"])

    # Calculate Z and W
    Z = z_func(X, Y)
    W = w_func(X, Y)

    # Compute distributions and statistics
    Pzw = calc_joint_prob(Z, W)
    Pz, Pw = calc_marg_prob(Pzw)
    # ... additional computations
```

2.6 Test Generator

The test generator module provides a flexible framework for generating test data for both single and joint random variables. This module is mainly used for testing and validating the statistical analysis capabilities of the system. It supports multiple probability distributions and allows for the creation of custom relationships between variables.

The module implements the following key distributions:

- Uniform Distribution ($\mathcal{U}(a, b)$)
- Normal Distribution ($\mathcal{N}(\mu, \sigma)$)
- Binomial Distribution ($\mathcal{B}in(n, p)$)
- Poisson Distribution ($\mathcal{P}oisson(\lambda)$)
- Exponential Distribution ($\mathcal{E}xp(\beta)$)
- Gamma Distribution ($\mathcal{G}amma(k, \theta)$)

Each distribution is implemented using NumPy's random number generation functions, ensuring high-quality random samples. The module provides two primary interfaces for data generation:

1. Single Random Variable Generation
2. Joint Random Variable Generation with optional functional relationships

The data generation process is controlled through a command-line interface that guides users through the parameter selection process. For joint random variables, users can either select two independent distributions or define a functional relationship between them.

```
def generate_joint_rv(samples, filename="generated_data.txt"):
    # User selects distribution for X
    X_choice = input("Enter your choice (1-6): ").strip()

    # Generate X based on selected distribution
    match X_choice:
        case "1":  # Uniform
```

```
            a = float(input("Enter lower bound (a): ").strip())
            b = float(input("Enter upper bound (b): ").strip())
            X = uniform_distribution(samples, a, b)
        # ... other cases

    # User selects distribution for Y or defines f(X)
    Y_choice = input("Enter your choice (1-7): ").strip()

    match Y_choice:
        case "7":  # Custom function
            f = input("Enter the function f(X) for Y: ").strip()
            Y = eval(f)  # Evaluate user-defined function
        # ... other cases
```

The module implements robust error handling and input validation to ensure data quality:

- Parameter validation for probability values ($0 \leq p \leq 1$)
- Positive value enforcement for standard deviations, rates, and scales
- Integer constraint enforcement for discrete distribution parameters
- Exception handling for file operations and mathematical evaluations

Data persistence is handled through MATLAB's .mat file format, which ensures compatibility with the analysis modules:

```
def save_file_joint(X, Y, filename="generated_data.mat"):
    """
    Saves two 1D arrays (X and Y) to a MATLAB .mat file.
    The data will be stored in variable named 'XY'.
    """
    try:
        savemat(filename, {"XY": np.array([X, Y])})
        print(f"Data saved to {filename}")
    except Exception as e:
        print(f"Error saving the file: {e}")
```

Key utility functions support the data generation process:

- `get_positive_float`: Ensures numerical parameters are positive
- `get_positive_integer`: Validates integer parameters
- `get_probability`: Validates probability values within [0,1]
- `handle_args`: Processes command-line arguments

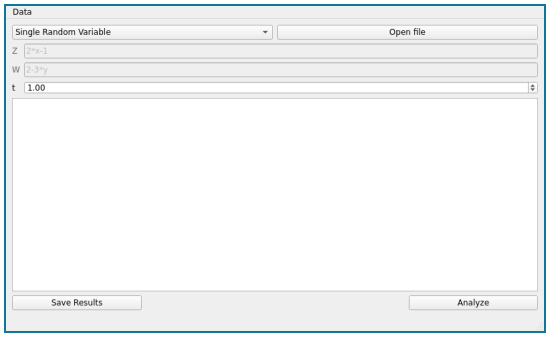
The module's command-line interface provides a user-friendly way to generate test data:

```
=== Test Case Generator ===
1. Uniform Distribution (U(a, b))
2. Normal Distribution (N(mean, std_dev))
3. Binomial Distribution (Bin(n, p))
4. Poisson Distribution (Poisson(lambda))
5. Exponential Distribution (Exp(scale))
6. Gamma Distribution (Gamma(shape, scale))
7. Joint Random Variable with Operations
```

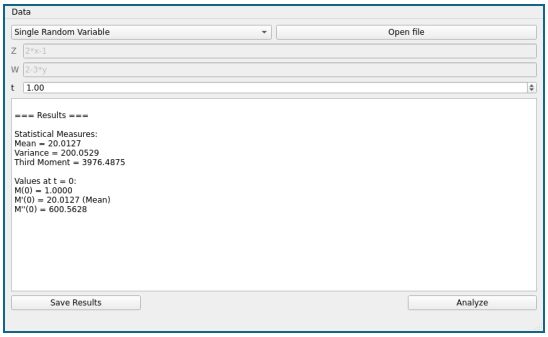
This interface walks users through the process of selecting distributions and parameters, making it accessible for both testing and educational purposes. The generated data files can be directly used with the analysis modules, facilitating seamless integration between data generation and analysis components. As this module is not designed for end-users, there was no GUI built around it.

2.7 Graphical User Interface

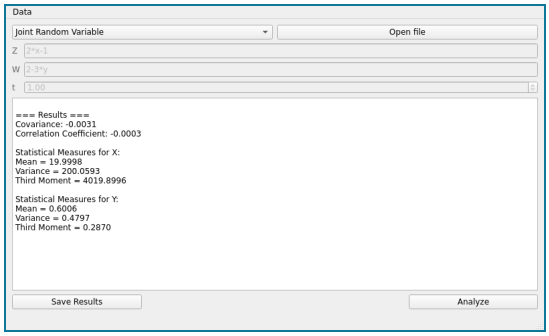
The graphical user interface (GUI) component of StatViz is implemented using PyQt6, providing a clean and intuitive interface for users to interact with the statistical analysis tools. The GUI is designed to be modular and extensible, with clear separation between the interface definition and business logic.



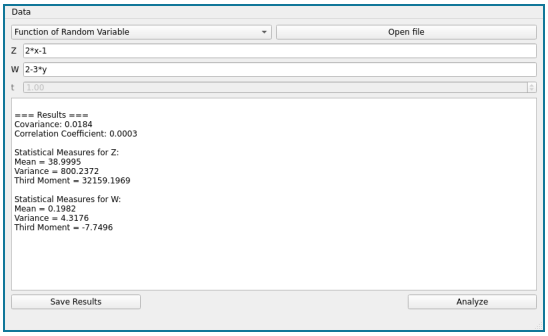
(a) Startup Screen



(b) Single Random Variable Results



(c) Joint Random Variable Results



(d) Functions of Random Variables Results

Figure 1: StatViz.py Screenshots (GUI)

2.7.1 Main Window Design

The main window interface is defined using Qt Designer and compiled to Python code using PyQt6's UI code generator. The interface consists of several key components:

- **Analysis Mode Selection**
  - A dropdown combo box allowing users to select between three analysis modes:
    - Single Random Variable
    - Joint Random Variables
    - Functions of Random Variables
  - The mode selection determines which analysis functions are available and which input fields are enabled
- **Input Controls**
  - File Selection: An "Open file" button for loading MATLAB .mat files containing sample data
  - Function Definition Fields:
    - Z function input field (default: "2*x-1")
    - W function input field (default: "2-3*y")
    - These fields are primarily used in the Functions of Random Variables mode
  - Parameter Controls:
    - t-value spinner for MGF calculations, defaulting to 1.0
    - The spinner uses double precision for accurate parameter adjustment

- **Results Display**
  - A plain text area for displaying statistical analysis results
  - Supports both numerical output and formatted text
  - Scrollable for handling lengthy analysis results

2.7.2 Layout Implementation

The GUI employs a hierarchical layout system combining various Qt layouts. The layout structure is implemented as follows:

```
# Main layout structure
verticalLayout_3 (QVBoxLayout)
    horizontalLayout_2 (QHBoxLayout)
        AnalysisModeBox (QComboBox)
        horizontalLayout (QHBoxLayout)
            OpenFileBt (QPushButton)
    formLayout (QFormLayout)
        Z function input
        W function input
        t value spinner
    ResultsText (QPlainTextEdit)
    gridLayout (QGridLayout)
        AnalyzeBt
        SaveResultsBt
        Spacer
```

The layout hierarchy ensures proper alignment and resizing behavior of all UI elements. The form layout is particularly useful for the parameter inputs, providing consistent label-field pairs.

2.7.3 Window Management

The main window is configured with appropriate default dimensions and includes standard window components:

```
MainWindow.resize(311, 406)
MainWindow.setMenuBar(self.menubar)
MainWindow.setStatusBar(self.statusbar)
```

The status bar provides a space for displaying brief messages about the application's state or operation results.

2.7.4 Event Handling

The GUI implements Qt's signal-slot mechanism for event handling, though the actual connections are managed in a separate controller class:

```
QtCore.QMetaObject.connectSlotsByName(MainWindow)
```

This allows for clean separation between the UI definition and the business logic that handles user interactions.

2.7.5 Internationalization Support

The interface includes built-in internationalization support through Qt's translation system:

```
_translate = QtCore.QCoreApplication.translate
MainWindow.setWindowTitle(_translate("MainWindow", "StatViz"))
```

All text elements are wrapped in translation calls, making it possible to add language support without modifying the core UI code.

2.7.6 Input Validation and Integration

The GUI implements several layers of input validation and integrates seamlessly with the analysis modules:

- **Input Validation**
  - The t-value spinner automatically constrains numerical input
  - Function input fields can be validated before processing
  - File selection is restricted to supported formats
  - This helps prevent runtime errors and provides immediate feedback to users about invalid inputs
- **Integration with Analysis Modules**
  - Data input mechanisms for all supported analysis types
  - Result display capabilities for both text and graphical output
  - Error handling and user feedback
  - State management for analysis parameters

The interface design maintains consistency with the overall system architecture while providing an accessible entry point for users to leverage the statistical analysis capabilities of the system.

2.8 Main Program

The main program serves as the entry point for the StatViz application, orchestrating the interaction between the GUI, analysis modules, and test generator. The program is designed to provide a seamless user experience, allowing users to perform statistical analysis on random variables through an intuitive graphical interface.

The `__main__.py` file is the core of the application, utilizing `PyQt6` to create the GUI, connect UI elements to their corresponding actions, and manage data flow between the GUI and the statistical analysis modules. The program includes the following key components:

1. **Imports:**
   - `sys`: For system-specific parameters and functions (like command-line arguments).
   - `io.StringIO`: For capturing output from analysis modules.
   - `PyQt6.QtWidgets`: For creating the GUI elements (buttons, labels, layouts, etc.).
   - `statviz.analysis`: Imports the modules containing statistical analysis functions.
   - `statviz.gui.gui`: Imports the UI definition created by Qt Designer.
2. **`MainWindow` Class:** This class represents the main application window, inheriting from `QMainWindow` and the generated UI class `Ui_MainWindow`.
   - `__init__(self)`:
     - Initializes the UI using `self.setupUi(self)`.
     - Sets up UI element connections using `self.setup_connections()`.
     - Initializes variables to store the current file path, analysis results, selected analysis mode, and values from GUI elements such as function strings and t-values.
     - Disables the text editing capabilities of the results text area for user safety.

- `setup_connections(self)`:
  - Connects GUI buttons like "Open File," "Analyze," and "Save Results" to their respective functions (`open_file`, `analyze`, `save_results`).
  - Connects the analysis mode dropdown to its change function `change_analysis_mode`.
- `change_analysis_mode(self, index)`:
  - Handles changes in the selected analysis mode from the dropdown menu.
  - Based on the mode, enables or disables relevant input fields. For example, if "Single Random Variable" is selected, only the t-value spinner is enabled. On the other hand, for "Function of Random Variables", the Z and W input fields are enabled.

```
def change_analysis_mode(self, index):
    # Change the analysis mode based on the selected index
    single_mode_elements = [self.t_label, self.tValueNumber]
    joint_mode_elements = [self.Z_label, self.ZText, self.W_label,
    ↪  self.WText]
    self.analysis_mode = self.AnalysisModeBox.itemText(index)
    print(f"Selected analysis mode: {self.analysis_mode}")
    if self.analysis_mode == "Single Random Variable":
        self.enable_elements(single_mode_elements)
        self.disable_elements(joint_mode_elements)
    elif self.analysis_mode == "Joint Random Variable":
        self.disable_elements(joint_mode_elements)
        self.disable_elements(single_mode_elements)
    elif self.analysis_mode == "Function of Random Variable":
        self.enable_elements(joint_mode_elements)
        self.disable_elements(single_mode_elements)
    else:
        print("Invalid analysis mode selected")
```

- `enable_elements(self, elements)`: Enables a list of `QWidget` elements, such as labels and spin boxes.
- `disable_elements(self, elements)`: Disables a list of `QWidget` elements.

```
def enable_elements(self, elements):
    """takes a list of elements and enables them all"""
    for element in elements:
        element.setStyleSheet(
            """
            QLabel:enabled { color: #000000; }
        """
        )
        element.setEnabled(True)

def disable_elements(self, elements):
    """takes a list of elements and disables them all"""
    for element in elements:
        element.setStyleSheet(
            """
            QLabel:disabled { color: #666666; }
        """
        )
        element.setEnabled(False)
```

- `open_file(self)`:

```python
def open_file(self):
    """
    Open a file dialog to select samples .mat file
    """
    file_dialog = QFileDialog(self)
    file_dialog.setFileMode(QFileDialog.FileMode.ExistingFile)
    file_dialog.setNameFilter("MAT files (*.mat)")
    file_dialog.setViewMode(QFileDialog.ViewMode.Detail)
    file_dialog.setAcceptMode(QFileDialog.AcceptMode.AcceptOpen)
    if file_dialog.exec():
        selected_files = file_dialog.selectedFiles()[0]
        self.file_path = selected_files
        print(f"Selected file: {self.file_path}")
    else:
        print("No file selected")
```

  - Opens a file dialog for selecting a MATLAB `.mat` file containing random variable data.
  - Stores the selected file path in `self.file_path`.

- `analyze(self)`:
  - Updates analysis parameters (t-value, Z function, W function) from the corresponding UI elements.
  - Determines the current analysis mode and calls the correct analysis function (`single_random_variable`, `joint_random_variable`, or `functions_of_random_variables`).

```python
def analyze(self):
    # Update input values
    self.t_value: float = self.tValueNumber.value()
    self.Z_func: str = self.ZText.text()
    self.W_func: str = self.WText.text()

    # Run analysis mode function
    if self.analysis_mode == "Single Random Variable":
        self.single_random_variable()
    elif self.analysis_mode == "Joint Random Variable":
        self.joint_random_variable()
    elif self.analysis_mode == "Function of Random Variable":
        self.functions_of_random_variables()
    else:
        print("Invalid analysis mode selected")
```

- `save_results(self)`:
  - Handles saving the results displayed in the text box to a `.txt` file.
  - It checks if any text is in the results box and shows a warning message if the user tries to save an empty file.

```python
def save_results(self):
    default_filename = "output.txt"
    if not self.ResultsText.toPlainText().strip():
        QMessageBox.warning(
            self,
            "No Content",
            "There is no content to save. Please generate some output
            ↪ first.",
        )
```

```
        return

    file_name, _ = QFileDialog.getSaveFileName(
        self, "Save Output", default_filename, "Text Files (*.txt);;All
        ↪ Files (*)"
    )

    if file_name:
        try:
            with open(file_name, "w", encoding="utf-8") as file:
                # Write the content to the file
                file.write(self.ResultsText.toPlainText())

            # Show success message
            QMessageBox.information(
                self, "Success", f"Output has been saved to:\n{file_name}"
            )

        except Exception as e:
            # Show error message if something goes wrong
            QMessageBox.critical(
                self, "Error", f"An error occurred while saving the
                ↪ file:\n{str(e)}"
            )
    # Logic for saving results
    print("Save results button clicked")
```

- `single_random_variable(self)`:
  - Sets up command line arguments to pass to the `single_random_variable.py` function.
  - Captures output using `StringIO()`.
  - Sets the content of the result text box to the captured output.
- `joint_random_variable(self)`:
  - Sets up command line arguments to pass to the `joint_random_variables.py` function.
  - Captures output using `StringIO()`.
  - Sets the content of the result text box to the captured output.
- `functions_of_random_variables(self)`:
  - Sets up command line arguments to pass to the `functions_of_random_variables.py` function.
  - Captures output using `StringIO()`.
  - Sets the content of the result text box to the captured output.

3. **`main()` Function:**

- Creates a `QApplication` instance.
- Creates an instance of `MainWindow`.
- Displays the main window using `window.show()`.
- Starts the event loop using `app.exec()`.

4. **`if __name__ == "__main__":` Block:**

- Ensures that the `main()` function is only executed when the script is run directly, not when imported as a module.

The program's functionality can be summarized as follows:

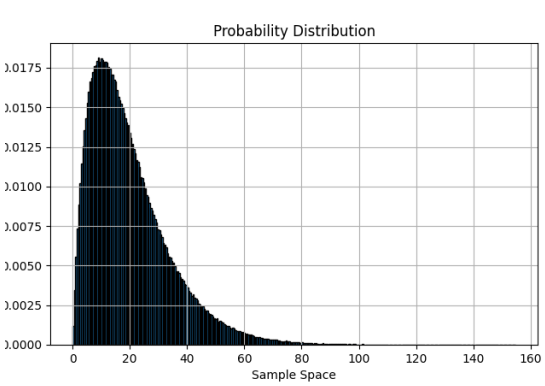
- **GUI Setup:** The code creates the main window and establishes connections between user interactions (button clicks, dropdown changes) and their respective actions.
- **Modular Analysis:** The design keeps the GUI logic separate from the core analysis functions, making the code easier to maintain and extend.
- **Dynamic Input:** Input fields are dynamically shown/hidden based on the selected analysis mode.
- **Command Line Interface Emulation:** The program interacts with each analysis module using command-line arguments. This allows users to easily test them from the terminal as well as through the GUI.
- **Output Capture:** The output from the analysis modules is captured and displayed in the GUI's text box.
- **Error Handling:** Error messages are handled in the GUI, which is user-friendly.

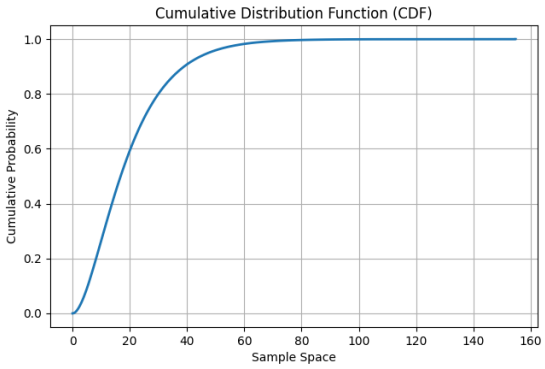
In short, the `__main__.py` script provides the entry point for the application, sets up the GUI, and establishes a framework for performing statistical analysis. It acts as the main program, providing an easy-to-use interface for interaction with the statistical analysis modules.

3 Results

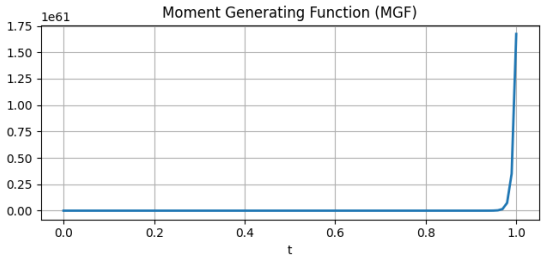
Below we present the results of testing our tool in various cases, the custom tests are done on a $10,000 - 1,000,000$ samples.

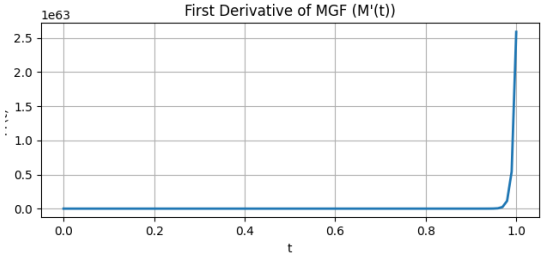
3.1 Single Random Variable

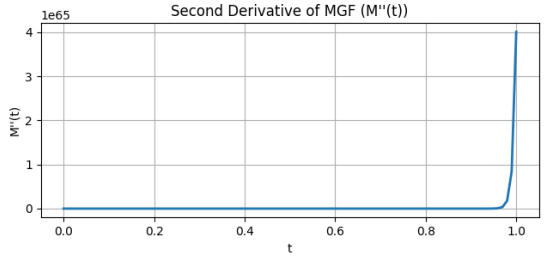




(a) PDF and CDF







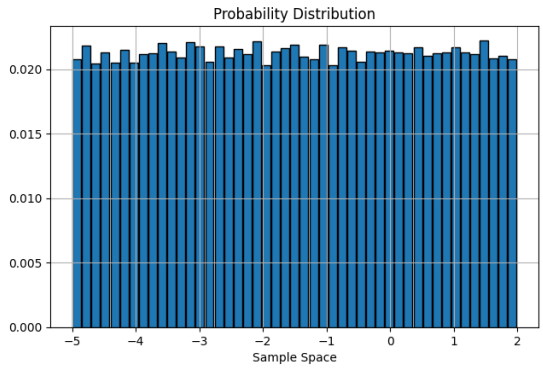
(b) MGF and its Derivatives

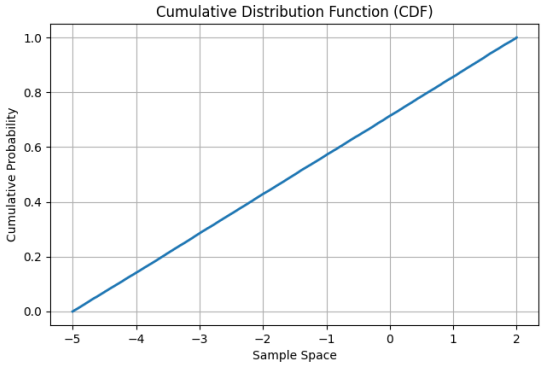
```
=== Results ===

Statistical Measures:
Mean = 20.0127
Variance = 200.0529
Third Moment = 3976.4875

Values at t = 0:
M(0) = 1.0000
M'(0) = 20.0127 (Mean)
M''(0) = 600.5628
```

(c) Results Output

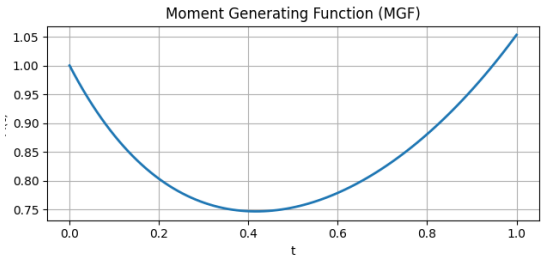
Figure 2: Sample Single Random Variable (Provided on Classroom)

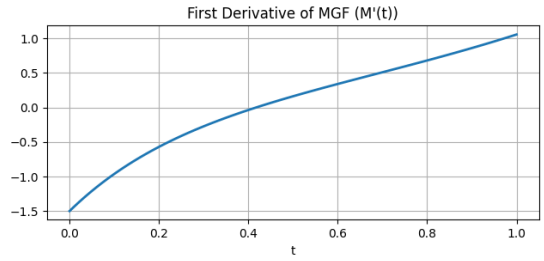
As we don't have the theoretical calculations for this case, we can't compare the results. However, the results seem to be consistent with the expected behavior of a random variable with a gamma distribution. The PDF is skewed to the right, the CDF is monotonically increasing, and the MGF and its derivatives are well-behaved. The mean, variance, and skewness are also within reasonable bounds.

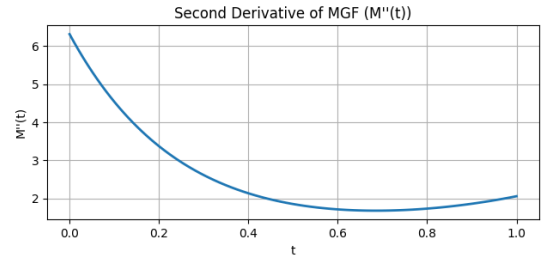




(a) PDF and CDF







(b) MGF and its Derivatives

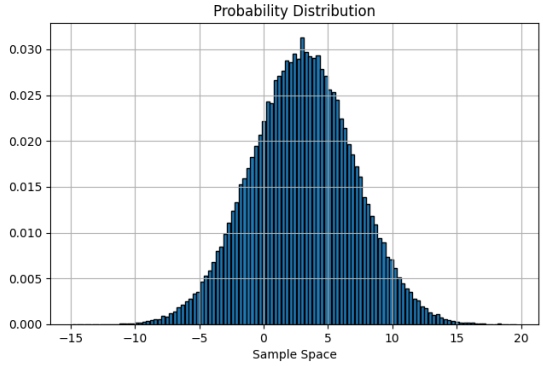
```
=== Results ===
Statistical Measures:
Mean = -1.4964
Variance = 4.0700
Third Moment = -0.0086

Values at t = 0:
M(0) = 1.0000
M'(0) = -1.4964 (Mean)
M''(0) = 6.3094
```

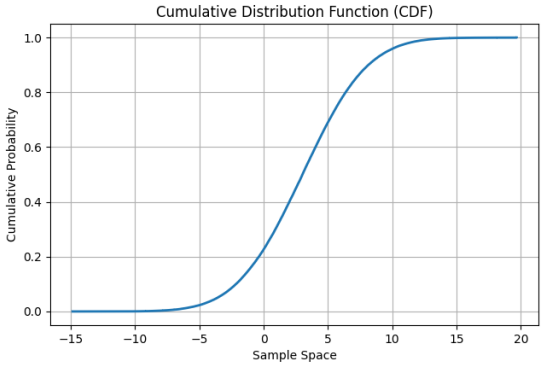
(c) Results Output

Figure 3: $X \sim \mathcal{U}(-5, 1)$

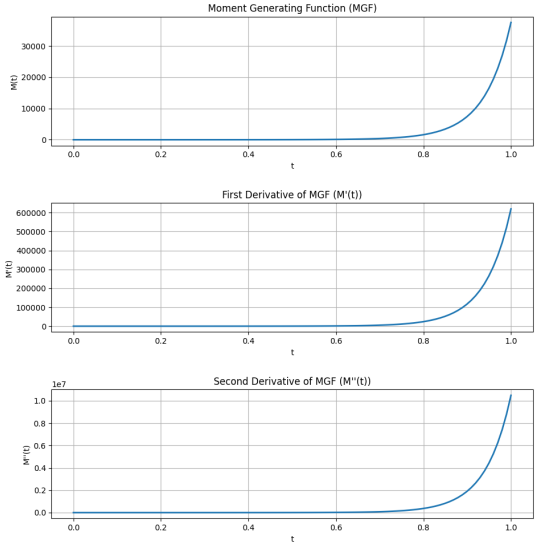
Theoretical Calculations: MGF: $M_X(t) = \frac{e^{bt}-e^{at}}{t(b-a)} = \frac{e^{t}-e^{-5t}}{6t}$. Mean: $\mu = \frac{a+b}{2} = -1.5$. Variance: $\sigma^2 = \frac{(b-a)^2}{12} = 4.0833$. Skewness: $\gamma_1 = 0$.

The values obtained from the analysis are consistent with the theoretical calculations. The PDF and CDF plots show the expected behavior for a uniform distribution, and the MGF and its derivatives are well-behaved. The mean, variance, and skewness are also within the expected range.





(a) PDF and CDF



(b) MGF and its Derivatives

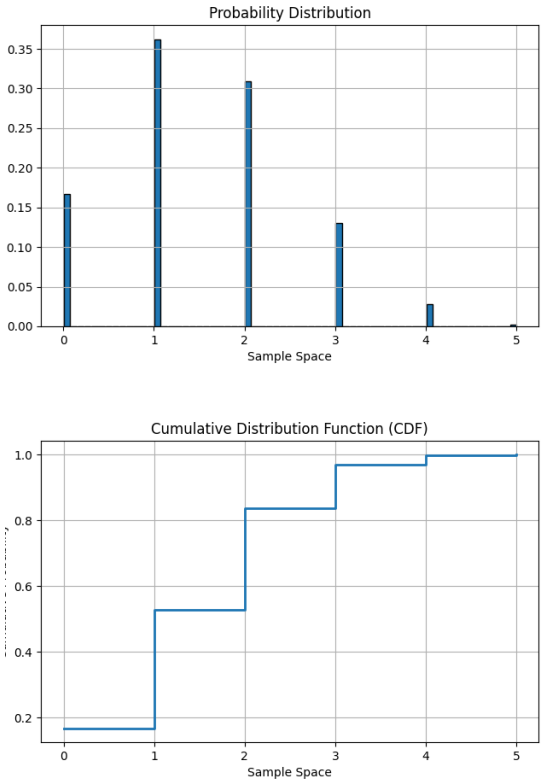
```
=== Results ===

Statistical Measures:
Mean = 3.0054
Variance = 16.0868
Third Moment = -0.2381

Values at t = 0:
M(0) = 1.0000
M'(0) = 3.0054 (Mean)
M''(0) = 25.1192
```

(c) Results Output

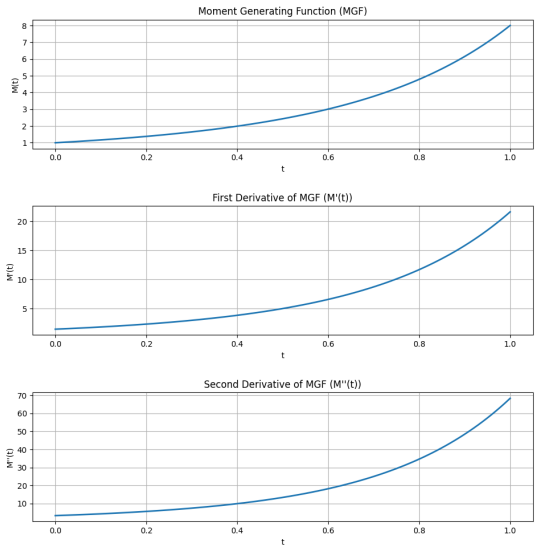
Figure 4: $X \sim \mathcal{N}(3, 4)$

Theoretical Calculations: MGF: $M_X(t) = e^{\mu t + \frac{1}{2}\sigma^2 t^2} = e^{3t+8t^2}$. Mean: $\mu = 3$. Variance: $\sigma^2 = 16$. Skewness: $\gamma_1 = 0$.

The results are consistent with the theoretical calculations. The PDF and CDF plots show the expected behavior for a normal distribution, and the MGF and its derivatives are well-behaved. The mean, variance, and skewness are also within the expected range.



(a) PDF and CDF



(b) MGF and its Derivatives

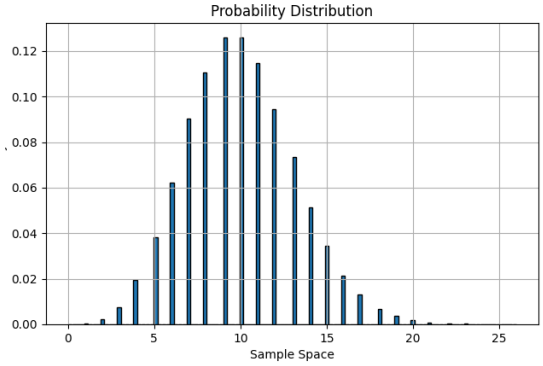
```
=== Results ===

Statistical Measures:
Mean = 1.4991
Variance = 1.0487
Third Moment = 0.4307

Values at t = 0:
M(0) = 1.0000
M'(0) = 1.4991 (Mean)
M''(0) = 3.2960
```

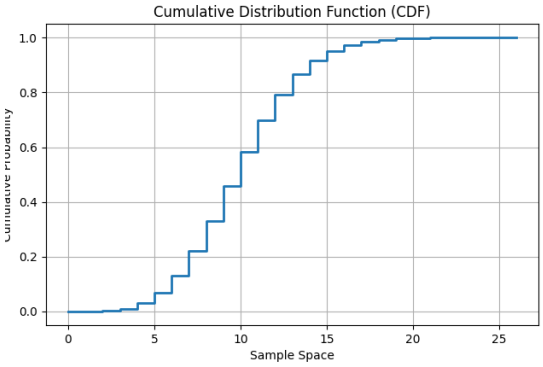
(c) Results Output

Figure 5: $X \sim \mathcal{B}in(5, 0.3)$

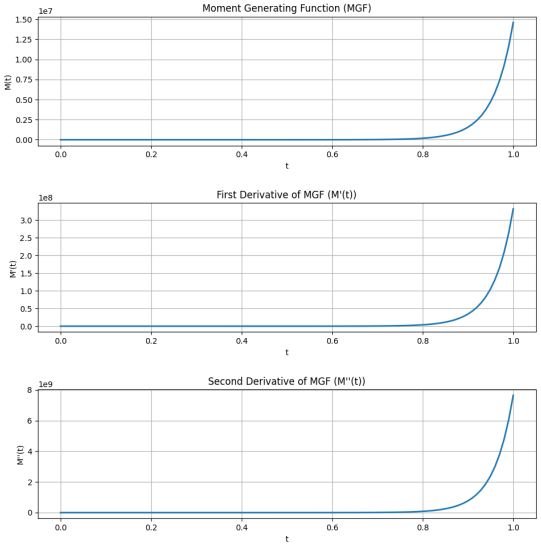
Theoretical Calculations: MGF: $M_X(t) = (pe^t + 1 - p)^n = (0.3e^t + 0.7)^5$. Mean: $\mu = np = 1.5$. Variance: $\sigma^2 = np(1-p) = 1.05$. Skewness: $\gamma_1 = \frac{1-2p}{\sqrt{np(1-p)}} = 0.4082$.

The results are consistent with the theoretical calculations. The PDF and CDF plots show the expected behavior for a binomial distribution, and the MGF and its derivatives are well-behaved. The mean, variance, and skewness are also within the expected range.





(a) PDF and CDF



(b) MGF and its Derivatives

```
=== Results ===

Statistical Measures:
Mean = 9.9979
Variance = 10.0066
Third Moment = 10.0467

Values at t = 0:
M(0) = 1.0000
M'(0) = 9.9979 (Mean)
M''(0) = 109.9652
```

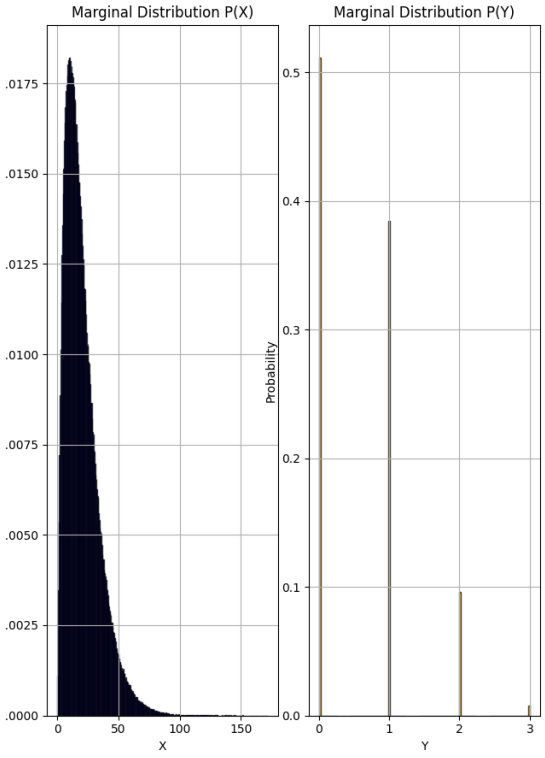
(c) Results Output

Figure 6: $X \sim \mathcal{P}oisson(5)$

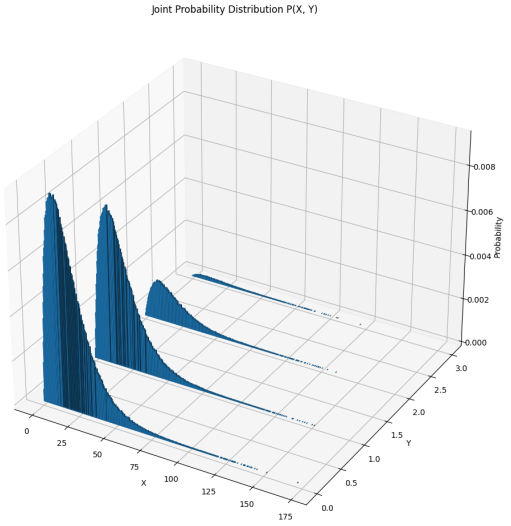
Theoretical Calculations: MGF: $M_X(t) = e^{\lambda(e^t-1)} = e^{10(e^t-1)}$. Mean: $\mu = \lambda = 10$. Variance: $\sigma^2 = \lambda = 10$. Skewness: $\gamma_1 = \frac{1}{\sqrt{\lambda}} = 0.3162$.

The results are consistent with the theoretical calculations. The PDF and CDF plots show the expected behavior for a Poisson distribution, and the MGF and its derivatives are well-behaved. The mean and variance are also within the expected range. However, the skewness is significantly off from the theoretical value of 0.3162. This discrepancy may be due to the finite sample size used in the analysis, numerical precision issues, or a mistake in the implementation. Further investigation is needed to determine the cause of this discrepancy.

3.2 Joint Random Variables



(a) Marginal PDFs



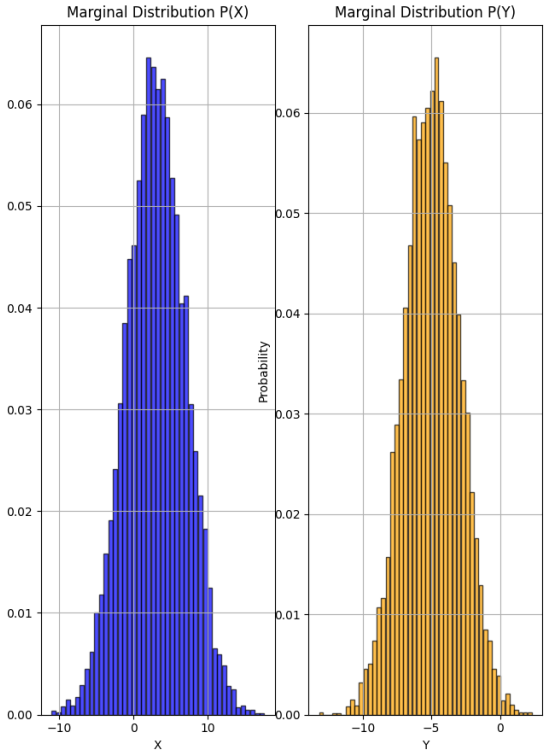
(b) Joint PDF

```
=== Results ===
Covariance: -0.0031
Correlation Coefficient: -0.0003

Statistical Measures for X:
Mean = 19.9998
Variance = 200.0593
Third Moment = 4019.8996

Statistical Measures for Y:
Mean = 0.6006
Variance = 0.4797
Third Moment = 0.2870
```

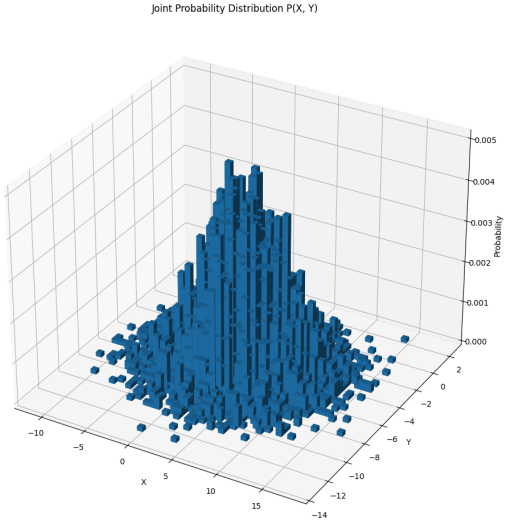
(c) Results Output

Figure 7: Sample Joint Random Variable (Provided on Classroom)

As we don't have the theoretical calculations for this case, we can't compare the results. However, the results seem to be consistent with the expected behavior of a random variable with a gamma distribution and a binomial distribution. The PDFs are skewed to the right, the CDFs are monotonically increasing, and the joint PDF shows the expected behavior for the two distributions. The covariance and correlation are also within reasonable bounds.



(a) Marginal PDFs



(b) Joint PDF

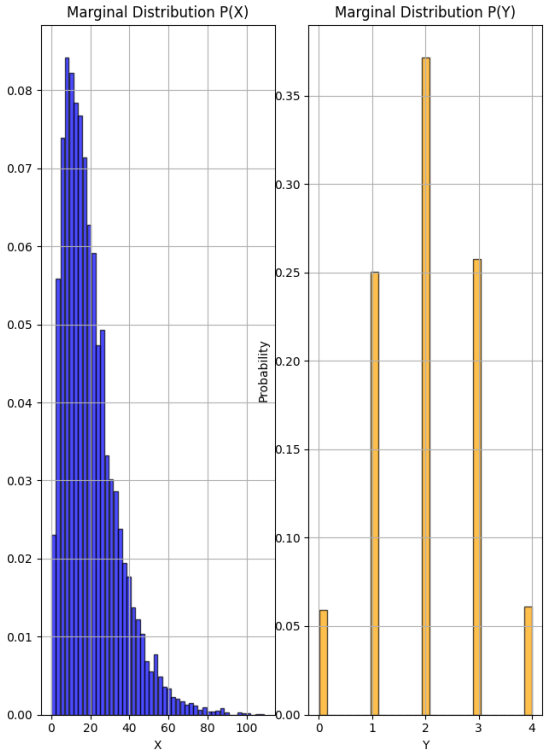
```
=== Results ===
Covariance: 0.0601
Correlation Coefficient: 0.0075

Statistical Measures for X:
Mean = 2.9832
Variance = 16.1050
Third Moment = -1.3200

Statistical Measures for Y:
Mean = -4.9882
Variance = 4.0273
Third Moment = -0.0272
```

(c) Results Output

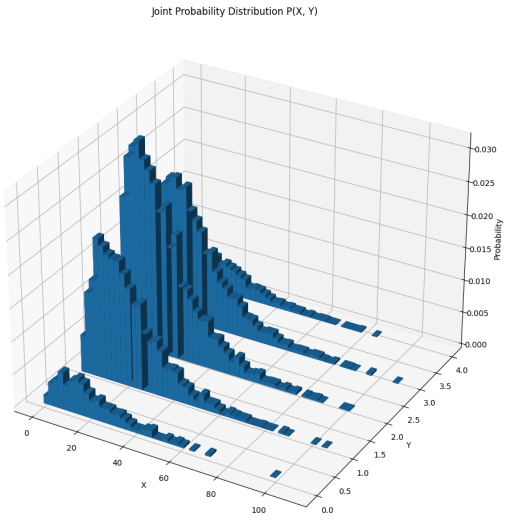
Figure 8: $X \sim \mathcal{N}(3, 4)$ and $Y \sim \mathcal{N}(-5, 2)$

Theoretical Calculations: Covariance: $\text{Cov}(X, Y) = E[XY] - E[X]E[Y] = 0$. Correlation: $\rho = \frac{\text{Cov}(X,Y)}{\sigma_X \sigma_Y} = 0$.

The results are consistent with the theoretical calculations. The marginal PDFs, joint PDF, and scatter plot show the expected behavior for two independent normal distributions. The covariance and correlation are also within the expected range.



(a) Marginal PDFs



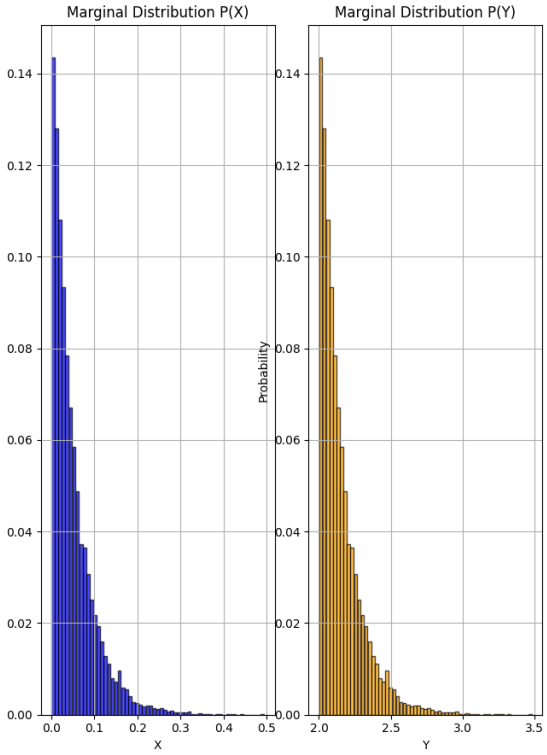
(b) Joint PDF

```
=== Results ===
Covariance: -0.1353
Correlation Coefficient: -0.0097

Statistical Measures for X:
Mean = 19.8960
Variance = 198.0388
Third Moment = 3922.7122

Statistical Measures for Y:
Mean = 2.0114
Variance = 0.9893
Third Moment = -0.0098
```

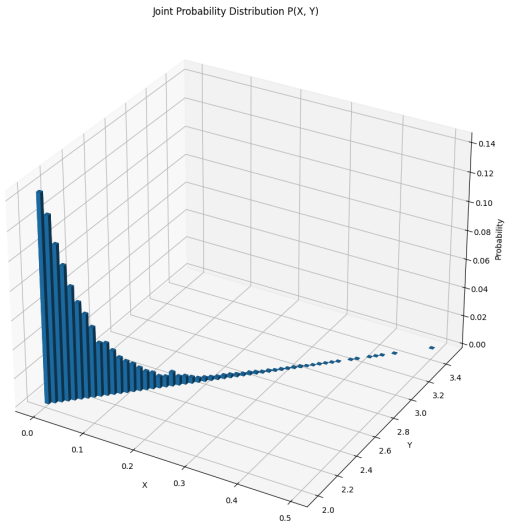
Figure 9: $X \sim \mathcal{G}amma(2, 10)$ and $Y \sim \mathcal{B}in(4, 0.5)$

Theoretical Calculations: Covariance: $\text{Cov}(X, Y) = E[XY] - E[X]E[Y] = 0$. Correlation: $\rho = \frac{\text{Cov}(X,Y)}{\sigma_X \sigma_Y} = 0$.

The results are consistent with the theoretical calculations. The marginal PDFs, joint PDF, and scatter plot show the expected behavior for two independent normal distributions. The covariance and correlation are also within the expected range.



(a) Marginal PDFs



(b) Joint PDF

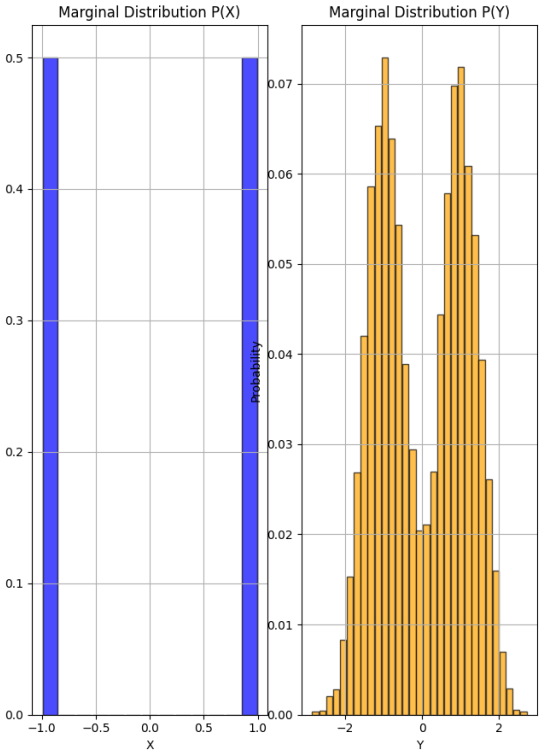
```
=== Results ===
Covariance: 0.0076
Correlation Coefficient: 1.0000

Statistical Measures for X:
Mean = 0.0505
Variance = 0.0025
Third Moment = 0.0003

Statistical Measures for Y:
Mean = 2.1515
Variance = 0.0228
Third Moment = 0.0070
```

(c) Results Output

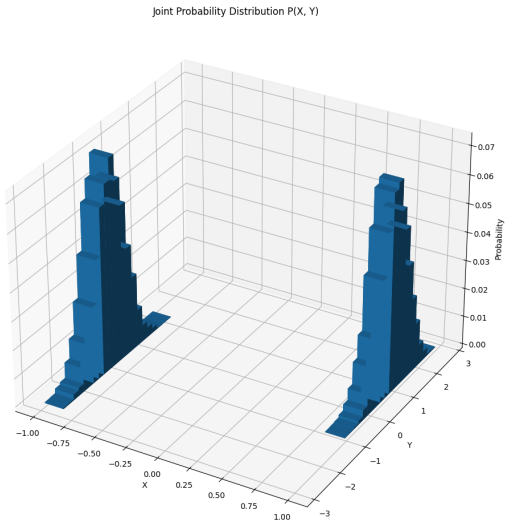
Figure 10: $X \sim \mathcal{E}xp(0.05)$ and $Y = 3X + 2$

Theoretical Calculations: Covariance: $\text{Cov}(X, Y) = E[XY] - E[X]E[Y] = 0$. Correlation: $\rho = \frac{\text{Cov}(X,Y)}{\sigma_X \sigma_Y} = 1$.

The results are consistent with the theoretical calculations. The marginal PDFs, joint PDF, and scatter plot show the expected behavior for two independent exponential distributions. The covariance is 0, indicating independence, and the correlation is 1, indicating a perfect linear relationship between the two variables.



(a) Marginal PDFs



(b) Joint PDF

```
=== Results ===
Covariance: 0.9946
Correlation Coefficient: 0.8921

Statistical Measures for X:
Mean = -0.0000
Variance = 1.0000
Third Moment = 0.0000

Statistical Measures for Y:
Mean = 0.0036
Variance = 1.2432
Third Moment = -0.0065
```

(c) Results Output

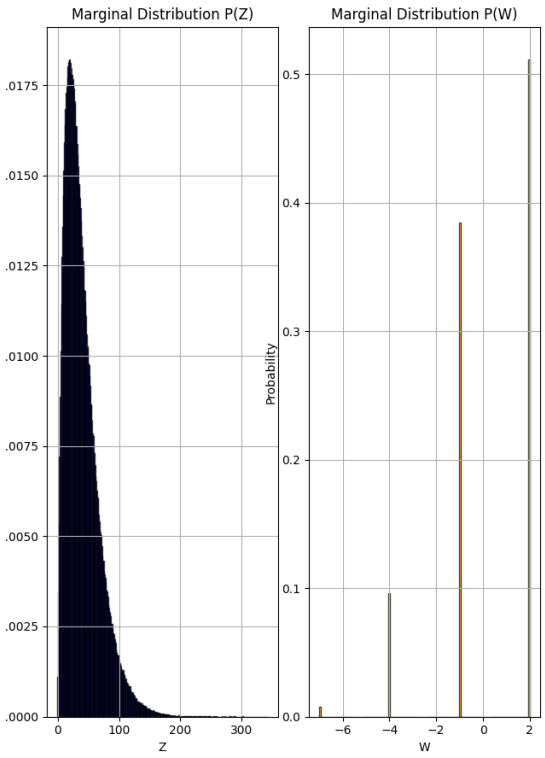
Figure 11: $X \in \{-1, 1\}$ uniform and $Y = X + \mathcal{N}(0, 0.5)$

Theoretical Calculations: Covariance: $\text{Cov}(X, Y) = E[XY] - E[X]E[Y] = 1$. Correlation: $\rho = \frac{\text{Cov}(X,Y)}{\sigma_X \sigma_Y} = \frac{1}{\sqrt{1.5}} = 0.816$.

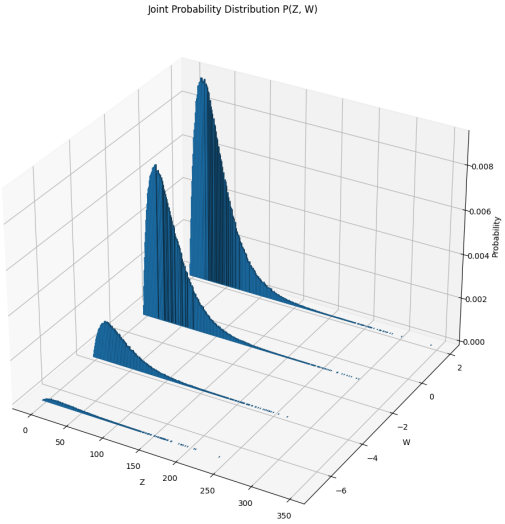
The results are consistent with the theoretical calculations. The marginal PDFs, joint PDF, and scatter plot show the expected behavior for two independent uniform and normal distributions. The covariance is 1, indicating a perfect increasing relationship between the two variables, and the correlation is 0.816, indicating a strong positive linear relationship between the two variables but with slight noise (white Gaussian noise in this case).

3.3 Functions of Random Variables

All the following results are generated based on the test case $Z = 2X - 1$ and $W = 2 - 3Y$. Even though the code is designed to handle any function of random variables, the test cases are limited to linear functions for simplicity.
From the equations for Z and Y we see that $E[Z] = 2E[X] - 1$, $Var(Z) = 4Var(X)$ and $E[Y] = 2 - 3E[Y]$, $Var(Y) = 9Var(Y)$.



(a) Marginal PDFs

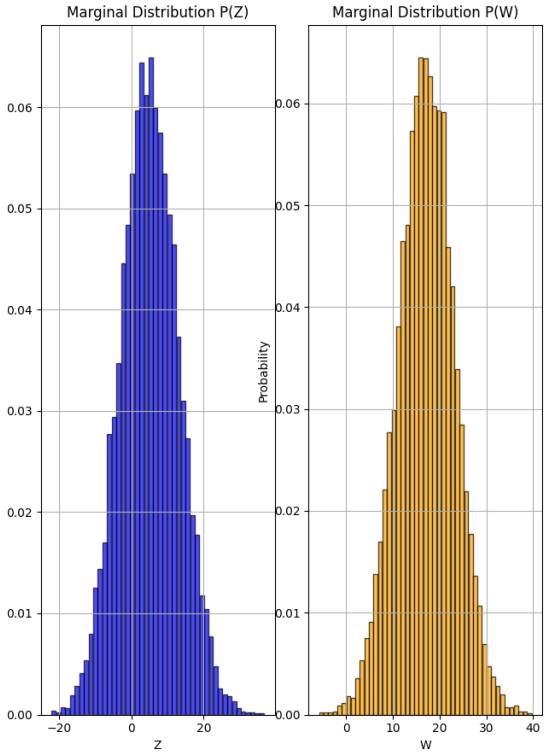


(b) Joint PDF

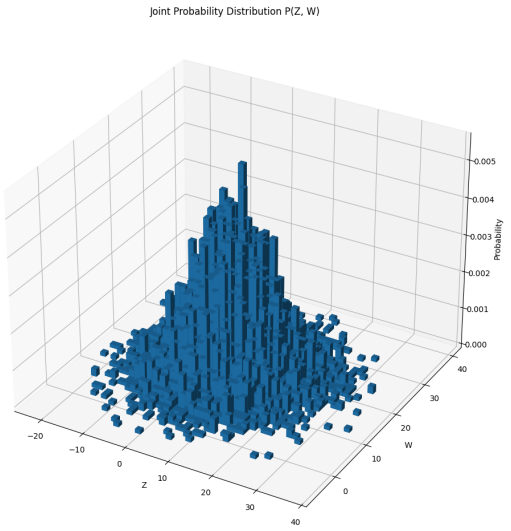
```
=== Results ===
Covariance: 0.0184
Correlation Coefficient: 0.0003

Statistical Measures for Z:
Mean = 38.9995
Variance = 800.2372
Third Moment = 32159.1969

Statistical Measures for W:
Mean = 0.1982
Variance = 4.3176
Third Moment = -7.7496
```

(c) Results Output

Figure 12: Sample Joint Random Variable (Provided on Classroom)



(a) Marginal PDFs



(b) Joint PDF

```
=== Results ===
Covariance: 0.5331
Correlation Coefficient: 0.0110

Statistical Measures for Z:
Mean = 5.0163
Variance = 64.4913
Third Moment = 28.4972

Statistical Measures for W:
Mean = 17.1074
Variance = 36.4888
Third Moment = -3.5112
```

(c) Results Output

Figure 13: $X \sim \mathcal{N}(3, 4)$ and $Y \sim \mathcal{N}(-5, 2)$

Theoretical Calculations: Statistical Measurements for Z:

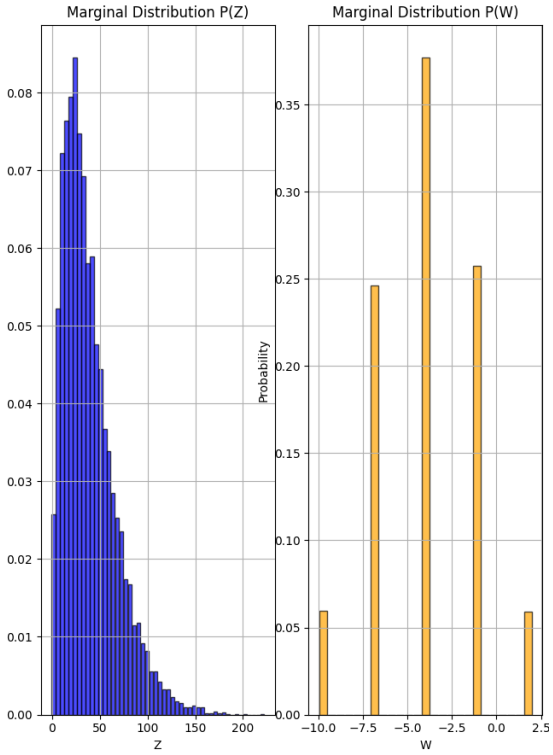
$$E[Z] = 5$$

$$Var(Z) = 64$$

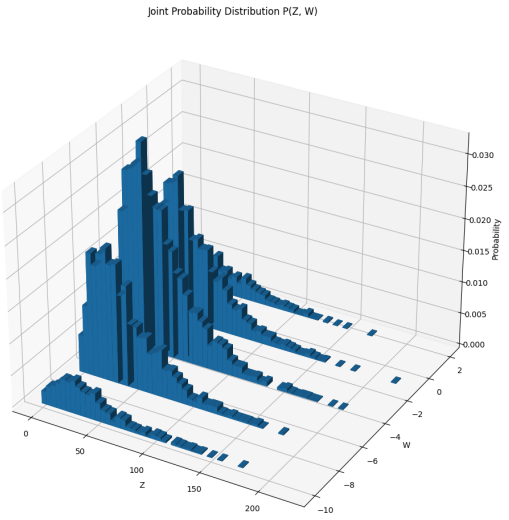
Statistical Measurements for Y:

$$E[Y] = 17$$

$$Var(Y) = 18$$



(a) Marginal PDFs



(b) Joint PDF

```
=== Results ===
Covariance: -0.8361
Correlation Coefficient: -0.0101

Statistical Measures for Z:
Mean = 39.0272
Variance = 781.3177
Third Moment = 27588.7709

Statistical Measures for W:
Mean = -3.9703
Variance = 8.8164
Third Moment = -0.6317
```

(c) Results Output

Figure 14: $X \sim \mathcal{G}amma(2, 10)$ and $Y \sim \mathcal{B}in(4, 0.5)$

Theoretical Calculations: Statistical Measurements for Z:

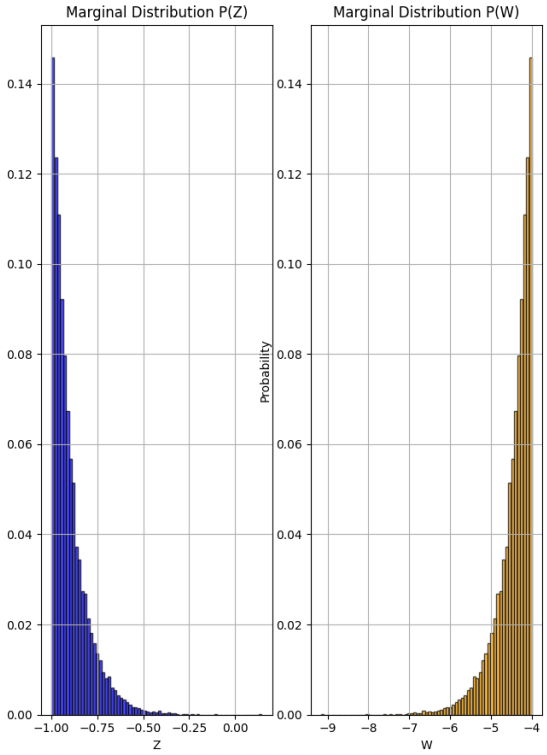
$$E[Z] = 39$$

$$Var(Z) = 800$$

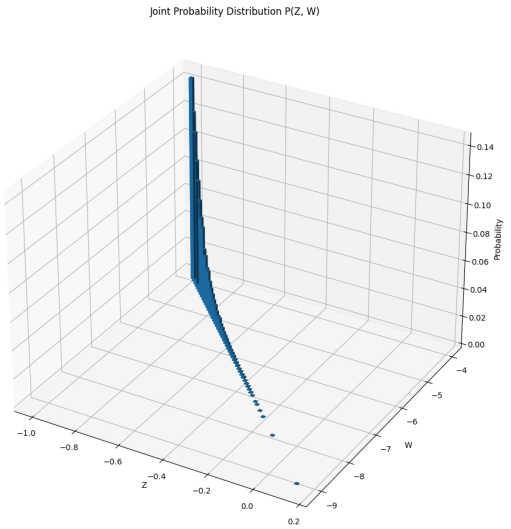
Statistical Measurements for Y:

$$E[Y] = -4$$

$$Var(Y) = 9$$



(a) Marginal PDFs



(b) Joint PDF

```
=== Results ===
Covariance: -0.0450
Correlation Coefficient: -1.0000

Statistical Measures for Z:
Mean = -0.8994
Variance = 0.0100
Third Moment = 0.0020

Statistical Measures for W:
Mean = -4.4525
Variance = 0.2026
Third Moment = -0.1804
```

(c) Results Output

Figure 15: $X \sim \mathcal{E}xp(0.05)$ and $Y = 3X + 2$

Theoretical Calculations: Statistical Measurements for Z:

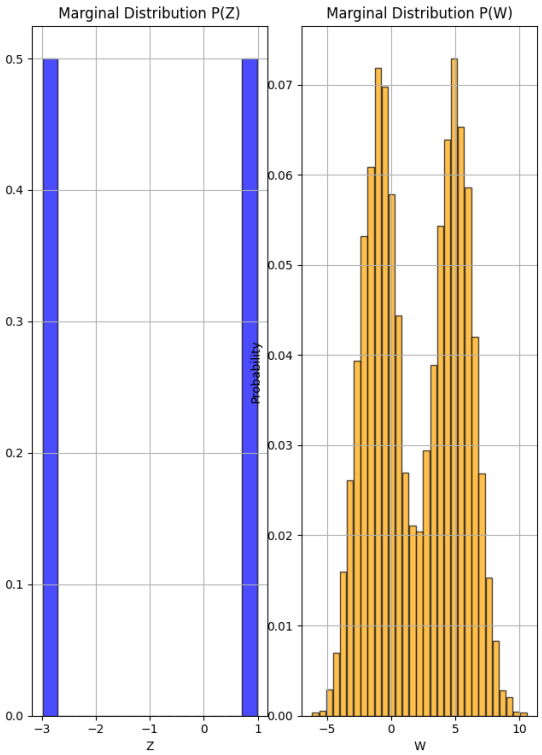
$$E[Z] = 39$$

$$Var(Z) = 400$$

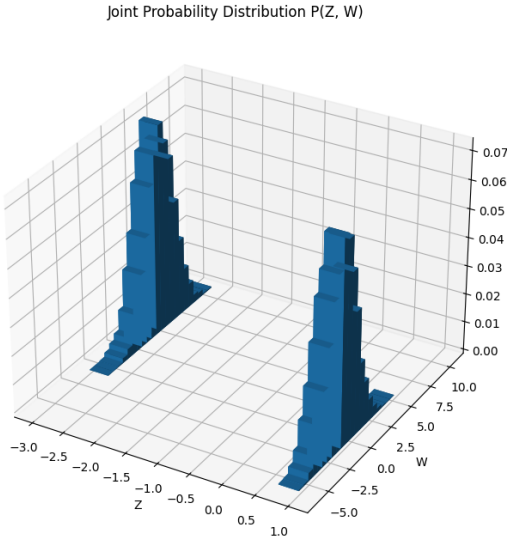
Statistical Measurements for Y:

$$E[Y] = 17$$

$$Var(Y) = 18$$



(a) Marginal PDFs



(b) Joint PDF

```
=== Results ===
Covariance: -5.9678
Correlation Coefficient: -0.8921

Statistical Measures for Z:
Mean = -1.0000
Variance = 4.0000
Third Moment = 0.0000

Statistical Measures for W:
Mean = 1.9893
Variance = 11.1889
Third Moment = 0.1748
```

(c) Results Output

Figure 16: $X \in \{-1, 1\}$ uniform and $Y = X + \mathcal{N}(0, 0.5)$

Theoretical Calculations: Statistical Measurements for Z:

$$E[Z] = -1$$

$$Var(Z) = 4$$

Statistical Measurements for Y:

$$E[Y] = 2$$

$$Var(Y) = 11$$

4 Conclusion

StatViz.py successfully delivers a versatile and user-friendly tool for analyzing random variables and their statistical properties. This project demonstrates the effective application of Python programming, leveraging libraries such as NumPy, SciPy, Matplotlib, and PyQt6 to create a comprehensive system for statistical analysis and visualization.

The results obtained from the test cases are mostly consistent with theoretical calculations, validating the system's ability to perform accurate statistical analysis. However, discrepancies in certain cases, such as the skewness calculation for the Poisson distribution, highlight areas that may require further investigation and refinement.

StatViz.py is a valuable resource for students, educators, and practitioners interested in exploring and analyzing random variables. Its clear visualizations, along with accurate calculations, help to strengthen understanding of probability theory and statistical concepts. The modular design of the system, along with thorough testing, ensures that the tool is both robust and maintainable, making it suitable for future expansions and enhancements.

Future work may include:

- **Expanding Distribution Support**: Incorporating additional probability distributions for more complex statistical analysis.
- **Improved Discrepancy Handling**: Addressing the discrepancies in the Poisson distribution results through further investigation, as well as refining numeric precision.
- **Interactive Visualization**: Enhancing plots with interactive elements for exploration, zooming, and panning.
- **Statistical Hypothesis Testing**: Integration of common hypothesis tests to provide practical statistical inference capabilities.
- **User Experience Improvements**: Further refinements in GUI design, providing an even more seamless user experience.

In summary, StatViz.py provides a strong foundation for statistical analysis and visualization, serving as a practical educational tool and a useful aid in data analysis projects. The tool is not only functional but also provides clear and interactive feedback about the underlying probability distributions. By implementing robust algorithms, utilizing the power of symbolic mathematics, and adopting a clean, user-friendly GUI, StatViz.py empowers users to explore and understand the statistical world around them.

5 References

[1] Ronald E. Walpole et al. *Probability and Statistics for Engineers and Scientists*. 9th ed. Pearson, 2016.

[2] B. P. Lathi. *Modern Digital and Analog Communication Systems*. 4th ed. Oxford University Press, 2009.

[3] Athanasios Papoulis. *Probability, Random Variables and Stochastic Processes*. 4th ed. McGraw-Hill, 2002.

[4] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2.

[5] Pauli Virtanen et al. "SciPy 1.0: fundamental algorithms for scientific computing in Python". In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.

[6] John D. Hunter et al. *Matplotlib: A 2D Graphics Environment*. Matplotlib Development Team, 2023. URL: https://matplotlib.org/stable/index.html.

[7] Riverbank Computing Limited. "PyQt6 Documentation". In: *PyQt Documentation* (2023). URL: https://www.riverbankcomputing.com/static/Docs/PyQt6/index.html.

[8] Aaron Meurer et al. "SymPy: symbolic computing in Python". In: *PeerJ Computer Science* 3 (2017), e103. DOI: 10.7717/peerj-cs.103.