Victor-Ioan Macovei
Computational Physics MSc
Babeș-Bolyai University

July 9, 2024

HOMEWORK 1 — Warm-Up

Requirements and Setup

Resources

- **Development Environment**:
  - Visual Studio Code with C/C++ extension by Microsoft (required but can be replaced);
  - Python: Version 3.x. (required);
  - VS Code Extensions: Python, GitLens, GitHub Pull Requests and Issues (not required);
- **Compiler and Debugger**: GCC for C/C++ and GDB for debugging (required but can be replaced);
- **Version Control**: Git for cloning the repository (not required);
- **Python Dependencies**: Use of `pip` to install requirements from a `requirements.txt` file (required);
- **Continuous Integration**: GitHub Actions for Continuous Integration (not required).

Setup Guide

- **Cloning the Repository**: The repository was created on the local machine using Git. It is available at GitHub - ssmHW;
- **Setting Up the Development Environment**:
  - Visual Studio Code was downloaded and installed;
  - Required extensions, including "C/C++" by Microsoft and optionally Python, GitLens, GitHub Pull Requests, and Issues, were installed through VS Code's Extensions;
- **Installing Compiler and Debugger**: For Windows, MinGW, which includes GCC and GDB, was installed. The MinGW bin directory was added to the system's PATH environment variable;
- **Configuring the Development Environment**: `tasks.json` and `launch.json` were correctly configured, ensuring the `miDebuggerPath` in `launch.json` was updated to the correct path of GDB on the machine, if different from the default;
- **Building and Running the Program**:
  - The program was compiled utilizing the default build task defined in `tasks.json`;
  - For debugging, the Debug view was switched to, the appropriate configuration selected from the dropdown menu;
- **Running Python Programs**: Python programs were run by setting the working directory to `ssmHW` and executing `python week1_WarmUp/ex1_SingleDieThrowing/src/simulate_dice.py` in terminal;
- **Continuous Integration**:
  - GitHub Actions for Continuous Integration was utilized, with workflow configurations found in `.github/workflows`, automatically running specified workflows for Python and C programs upon commits and pull requests;
  - Finally, the project was pushed to GitHub.

1 Return-Map Test of Built-in Uniform RNG

The return-map test examines the sequence of numbers generated by an RNG by plotting each number against the subsequent number. This section involves testing a built-in uniform RNG using the return-map method to evaluate its randomness.

General Approach

The approach involves generating a sequence of uniformly distributed numbers using a built-in RNG and plotting each number against the subsequent one. The return-map should ideally show no discernible pattern if the RNG is truly uniform and random.

Source Files Documentation

`main.cpp` - Main program to generate uniform numbers and perform return-map test:

- **int main():**
  - Description: Generates `N` uniformly distributed numbers using the built-in RNG and saves them to a file.

`analysis.py` - Analyzes and visualizes the uniform numbers:

- **def plot_return_map():**
  - Parameters: None
  - Description: Plots each uniform number against the subsequent one to create the return-map.

2 Pearson Correlation Test of Built-in RNG

Pearson correlation tests measure the linear correlation between different parts of a sequence generated by an RNG. This section involves studying the Pearson correlation of a built-in RNG to determine its randomness quality.

General Approach

The approach involves generating a sequence of numbers using a built-in RNG and calculating the Pearson correlation coefficients for various lags. Low correlation values indicate good randomness properties.

Source Files Documentation

`main.cpp` - Main program to perform Pearson correlation test on RNG:

- **int main():**
  - Description: Generates `N` uniformly distributed numbers using the built-in RNG, calculates and saves the Pearson correlation coefficients for lags up to `max_lag`.

`analysis.py` - Analyzes and visualizes Pearson correlation:

- **def plot_pearson_corr():**
  - Parameters: None
  - Description: Plots the Pearson correlation coefficients against the lag values.

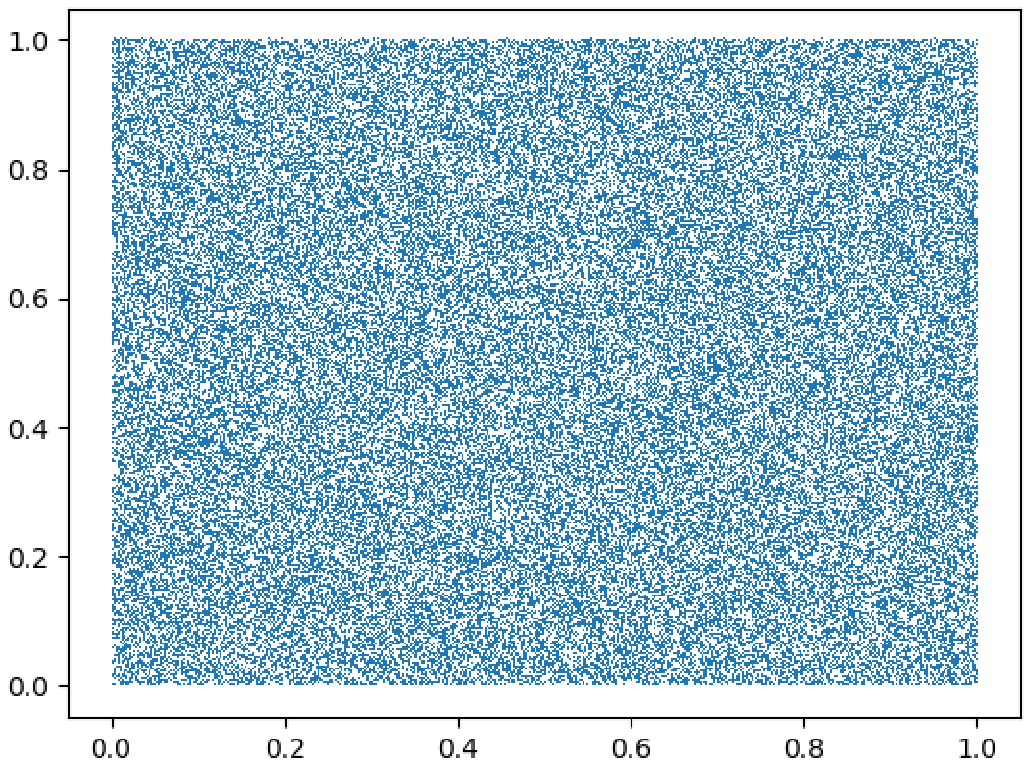

Figure 1: Return-map of uniformly generated numbers.

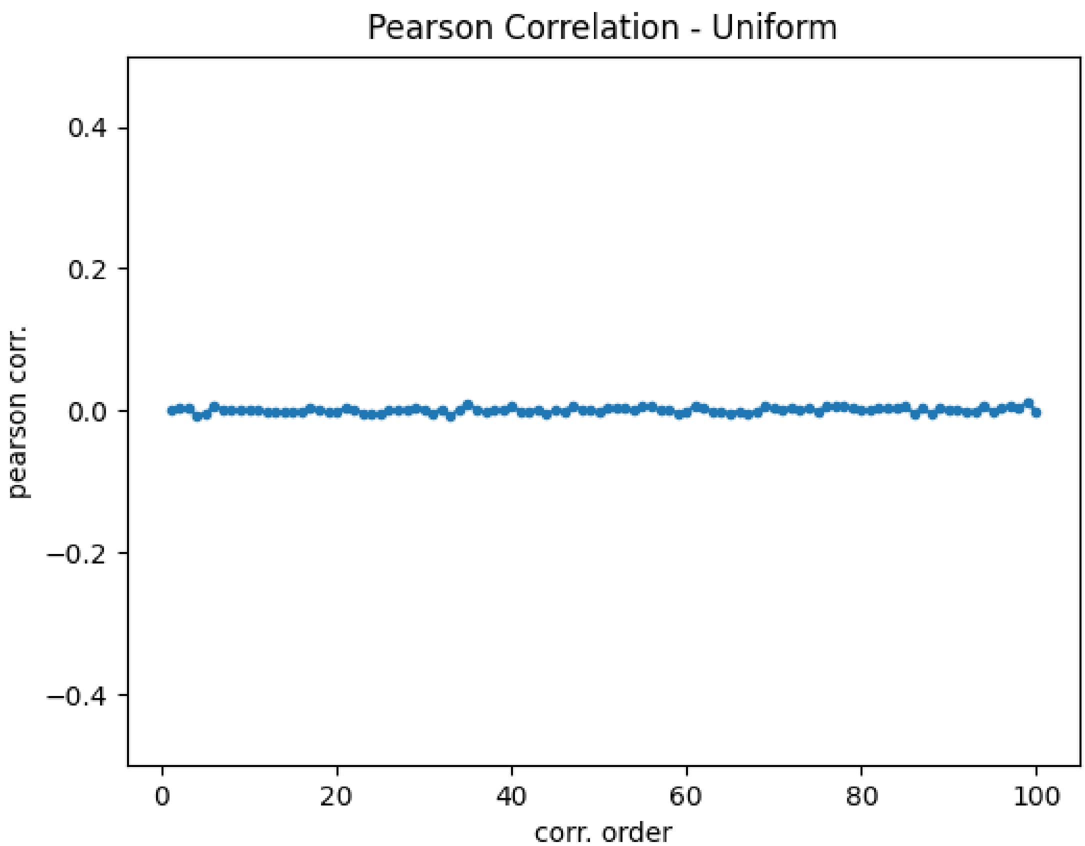

Figure 2: Pearson correlation plot.

3 Linear Congruential RNG and Parameter Effects

Linear congruential generators (LCGs) are simple RNGs that use linear congruence relations to produce sequences of pseudo-random numbers. This section involves writing an LCG and studying the effects of its parameters on period, return-map, and Pearson correlation.

General Approach

The approach involves implementing an LCG and generating sequences with different parameters. The period is checked, and return-maps and Pearson correlation coefficients are analyzed to understand the effects of parameter variations.

Source Files Documentation

`linear_congruential_rng.h` **and** `linear_congruential_rng.cpp` - Implements Linear Congruential Generator:

- **LinearCongruentialRNG(long int a, long int c, long int m, int seed):**
  - Parameters: `long int a`, `long int c`, `long int m`, `int seed`
  - Description: Constructor that initializes the LCG with given parameters. Sets the current state to the seed value.
- **int generate():**
  - Parameters: None
  - Output: `int`
  - Description: Generates the next random number in the sequence using the LCG formula.
- **std::vector<double> generateSequence(int length, const std::string& filename):**
  - Parameters: `int length`, `const std::string& filename`
  - Output: `std::vector<double>`
  - Description: Generates a sequence of random numbers of specified length and writes them to a file. Returns the generated sequence as a vector.
- **int checkPeriod(int periodMax):**
  - Parameters: `int periodMax`
  - Output: `int`
  - Description: Checks the period of the LCG by detecting the repetition of states within the specified maximum period. Returns the detected period or -1 if not found within `periodMax`.

`main.cpp` - Main program to study LCG parameters:

- **int main():**
  - Description: Generates sequences using different LCG parameters, checks their periods, and calculates and saves the Pearson correlation coefficients for sequences generated by different LCGs.

`analysis.py` - Analyzes and visualizes LCG results:

- **def plot_prng_subplots():**
  - Parameters: None
  - Description: Plots the return maps for sequences generated by different LCGs.
- **def plot_lcg_pearson():**
  - Parameters: None
  - Description: Plots the Pearson correlation coefficients for sequences generated by different LCGs.

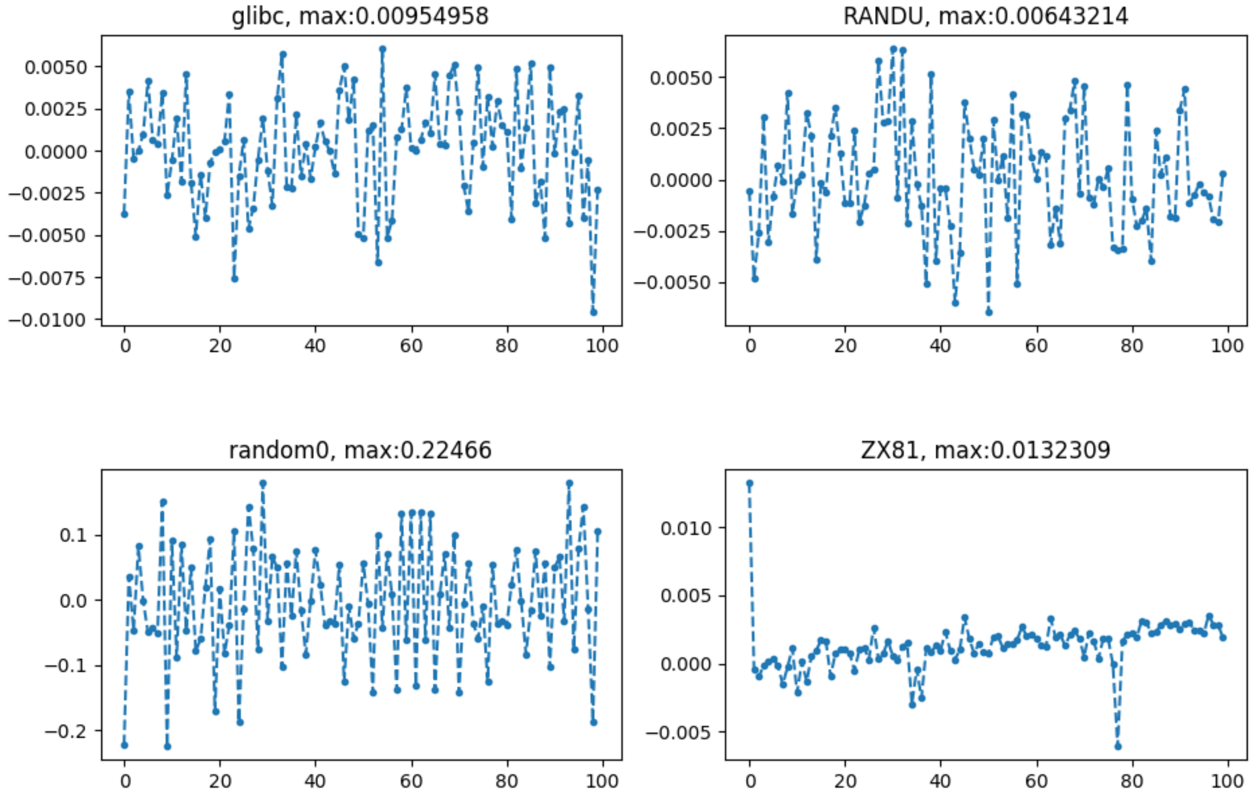

Figure 3: LCG Pearson correlation subplots.

4 Exponential Decay Test for Built-in Uniform RNG

The exponential decay test examines how the number of empty pixels decays when numbers are mapped to a grid. This section involves testing a built-in uniform RNG to verify the exponential decay property.

General Approach

The approach involves generating uniformly distributed numbers using a built-in RNG and mapping them to a grid. By counting the number of empty pixels over multiple trials, the decay pattern can be analyzed and verified as exponential.

Source Files Documentation

`main.cpp` - Main program to perform exponential decay test:

- **int main():**
  - Description: Performs the exponential decay test by mapping uniformly distributed numbers to a grid and counting empty pixels. Saves the results to a file.

`analysis.py` - Analyzes and visualizes exponential decay test results:

- **def plot_exponential():**

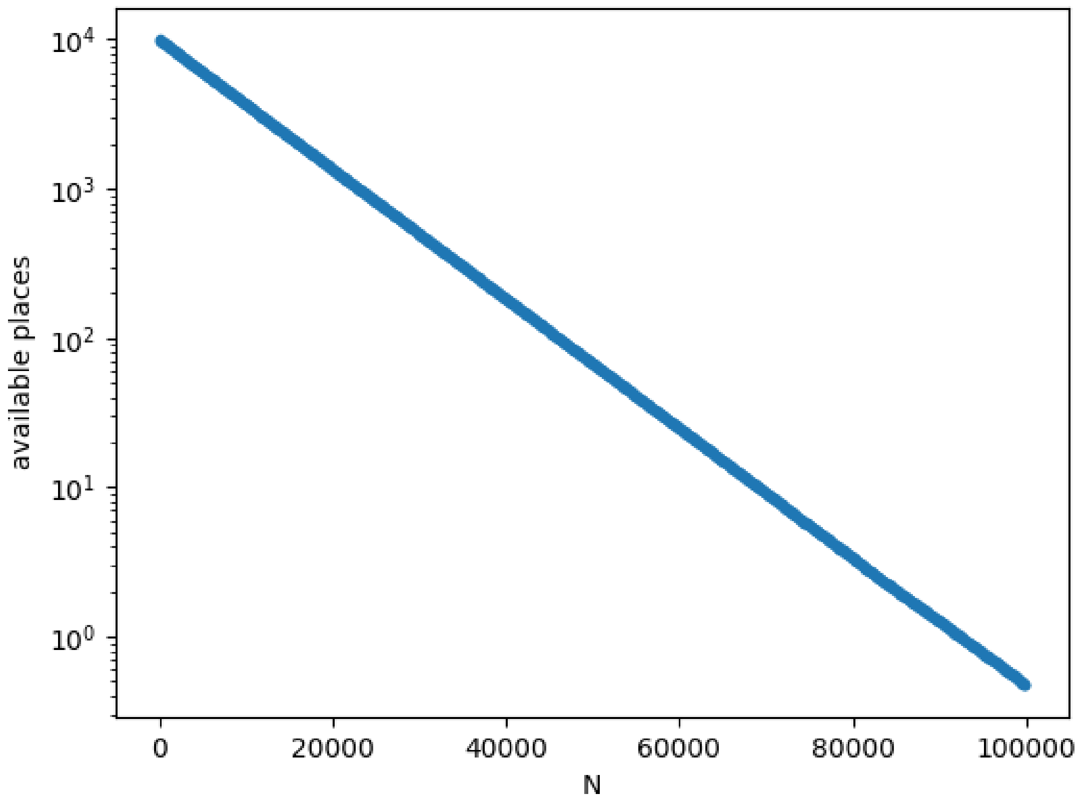

Figure 4: Exponential distribution semilogy plot.

- Parameters: None
- Description: Plots the exponential decay of available places over multiple trials.

5 Custom Distribution Generators

Custom distribution generators produce numbers that follow specific probability distributions. This section involves generating sequences for various distributions and verifying them using histograms.

General Approach

The approach involves implementing custom generators to produce sequences following $x^2, 1-x^2$, and semi-circular distributions. The generated sequences are verified using histograms to ensure they match the expected distributions.

Source Files Documentation

`distribution_generators.h` and `distribution_generators.cpp` - Implements custom distribution generators:

- **DistributionGenerators(int seed):**

    - Parameters: `int seed`
    - Description: Constructor that initializes the distribution generators with a given seed.

- **std::vector<double> generateX2_01(int length, const std::string& filename):**
    - Parameters: `int length`, `const std::string& filename`
    - Output: `std::vector<double>`
    - Description: Generates a sequence following the distribution $X^2(0,1)$ and writes it to a file. Returns the generated sequence as a vector.

- **std::vector<double> generateX2_12(int length, const std::string& filename):**
    - Parameters: `int length`, `const std::string& filename`
    - Output: `std::vector<double>`
    - Description: Generates a sequence following the distribution $X^2(1,2)$ and writes it to a file. Returns the generated sequence as a vector.

- **std::vector<double> generate1MinusX2(int length, const std::string& filename):**
    - Parameters: `int length`, `const std::string& filename`
    - Output: `std::vector<double>`
    - Description: Generates a sequence following the distribution $1 - X^2$ using the Newton-Raphson method and writes it to a file. Returns the generated sequence as a vector.

- **std::vector<double> generateSemiCircular(int length, const std::string& filename):**
    - Parameters: `int length`, `const std::string& filename`
    - Output: `std::vector<double>`
    - Description: Generates a sequence following the semi-circular distribution and writes it to a file. Returns the generated sequence as a vector.

- **double newtonMethod(double x, double p, double (*func)(double, double), double (*derivFunc)(double, double)):**
    - Parameters: `double x`, `double p`, `double (*func)(double, double)`, `double (*derivFunc)(double, double)`
    - Output: `double`
    - Description: Uses the Newton-Raphson method to find the root of the given function. Returns the computed root value.

`analysis.py` - Analyzes and visualizes custom distributions:

- **def plot_distribution(file_name, func, title):**
    - Parameters: `file_name`, `func`, `title`
    - Description: Plots the distribution of data from the specified file against the given theoretical function. Saves the plot to a file.

6 Box-Muller Gaussian RNG

The Box-Muller algorithm is a method to generate Gaussian-distributed numbers from uniformly distributed random numbers. This section involves writing a Gaussian RNG using the Box-Muller algorithm and applying various tests on it.

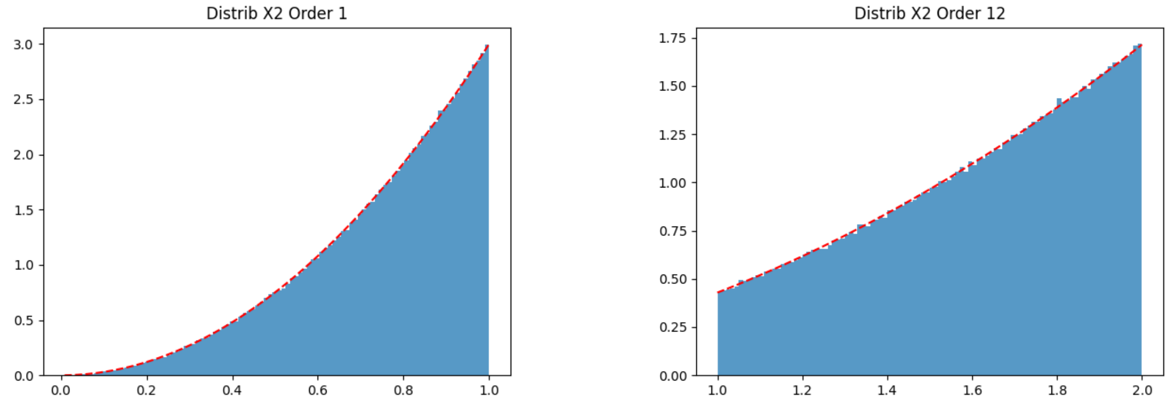

Figure 5: Distribution plots for x^2 distributions.

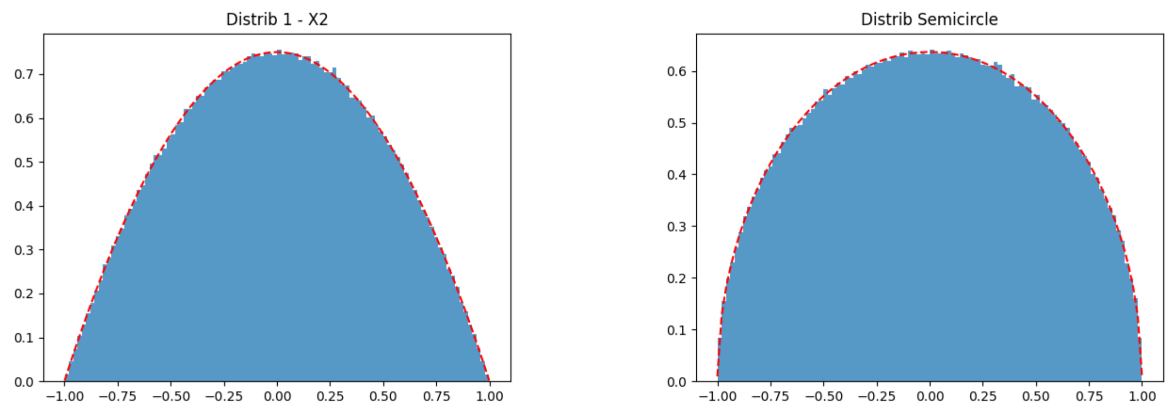

Figure 6: Distribution plots for $1 - x^2$ and semi-circular distributions.

General Approach

The approach involves implementing the Box-Muller algorithm to generate Gaussian-distributed numbers. The generated numbers are then tested using histogram, return-map, and Pearson correlation tests to verify their properties.

Source Files Documentation

`gaussian_rng.h` and `gaussian_rng.cpp` - Implements Gaussian RNG using Box-Muller algorithm:

- **GaussianRNG(int seed):**
  - Parameters: `int seed`
  - Description: Constructor that initializes the Gaussian RNG with a given seed.
- **std::vector<double> boxMuller(int length, const std::string& filename):**
  - Parameters: `int length`, `const std::string& filename`
  - Output: `std::vector<double>`

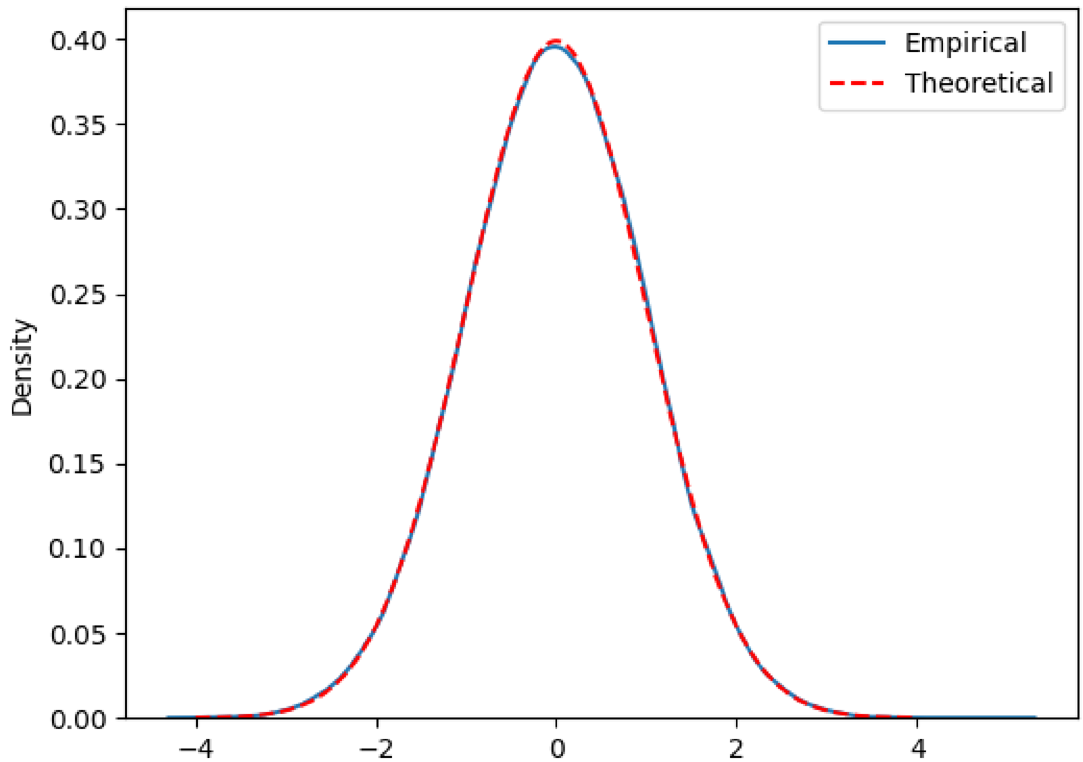

Figure 7: Kernel Density Estimation vs. Theoretical Normal Distribution.

  - Description: Generates a sequence of Gaussian-distributed random numbers using the Box-Muller transform and writes them to a file. Returns the generated sequence as a vector.

`main.cpp` - Main program to test Box-Muller Gaussian RNG:

- **int main():**
  - Description: Generates `N` Gaussian-distributed numbers using the Box-Muller algorithm, saves them to a file, and calculates and saves the Pearson correlation coefficients for the generated Gaussian numbers.

`analysis.py` - Analyzes and visualizes Box-Muller Gaussian RNG results:

- **def plot_kde_vs_theoretical():**
  - Parameters: None
  - Description: Plots the Kernel Density Estimation (KDE) of the Gaussian numbers against the theoretical normal distribution.
- **def plot_pearson_corr():**
  - Parameters: None
  - Description: Plots the Pearson correlation coefficients for the Gaussian numbers.

Submitted by Victor-Ioan Macovei on July 9, 2024.

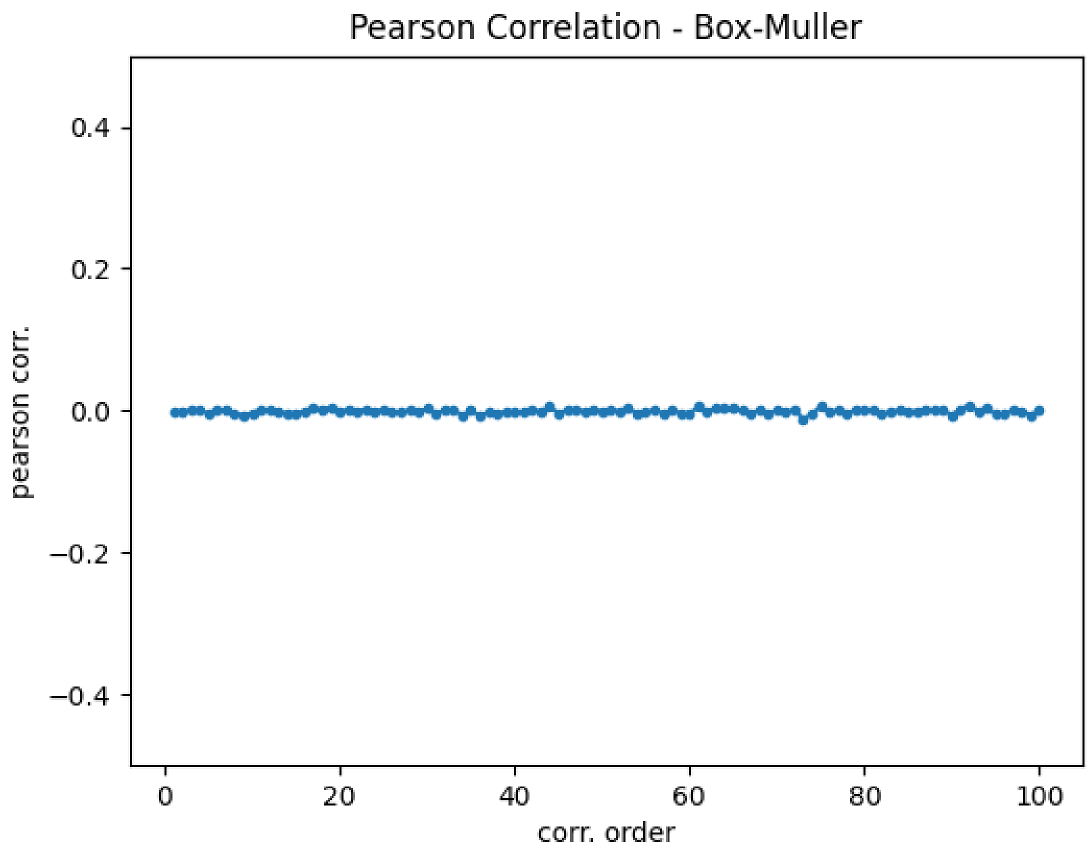

Figure 8: Pearson correlation plot for Box-Muller generated numbers.