Symulcaje Raport końcowy

Kinga Heda, Bartosz Łuksza i Karol Cieślik

Raport

Spis treści

1 Wstęp.

Niniejsze sprawozdanie zbierze poznane metody symulacyjne poznane na kursie. Całość symulacyjna była wykonana w Pythonie.

Wykorzystane narzędzia:

import numpy as np

import matplotlib.pyplot as plt

from scipy.stats import arcsine

import time

2 Zad 1.

2.1 Wstęp

Generator ACORN (Additive Congruential Random Number) to algorytm służący do generowania ciągów liczb pseudolosowych. Oparty jest na ciągach liczbowych zdefiniowanych poprzez równania rekurencyjne:

$$X_n^0 = X_{n-1}^0, \quad n \geqslant 1$$

$$X_n^m = (X_n^{m-1} + X_{n-1}^m) \mod M, \quad m = 1, \ldots, k, \quad n \geqslant 1$$

$$Y_n^k = \frac{X_n^k}{M}, \quad n \geqslant 1$$

2.2 Implementacja generatora ACORN

```
1  def ACORN(N: int, k: int, M: int, Lag: int, seed: int = (2**13) - 1)
       -> np.array:
2      """Funkcja generuje N pr bek z generatora ACORN
3
4      Args:
5          N: liczba generowanych zmiennych
6          k: rz d algorytmu
7          M: dzielnik
8          Lag: op  nienie w dzia Ćaniu algorytmu
9
10     Returns:
```

```
        tuple: list Ź N pr bek z generatora ACORN

    Example:
        >>> ACORN(1, 9, 2**30 - 1, 1000)
        array([0.7813808])

    """
    X = np.zeros((k, N + Lag))

    X[:, 0] = seed
    X[0, 1:] = seed

    for i in range(1, k):
        for j in range(1, N + Lag):
            X[i, j] = (X[i - 1, j] + X[i, j - 1]) % M

    Y = (X[k - 1, Lag : Lag + N]) / M

    return Y


# Parametery
N = 1000
k = 9
M = 2**30 - 1
Lag = 10**3

Y_ACORN = ACORN(N, k, M, Lag)
```

Listing 1: Implementacja generatora ACORN

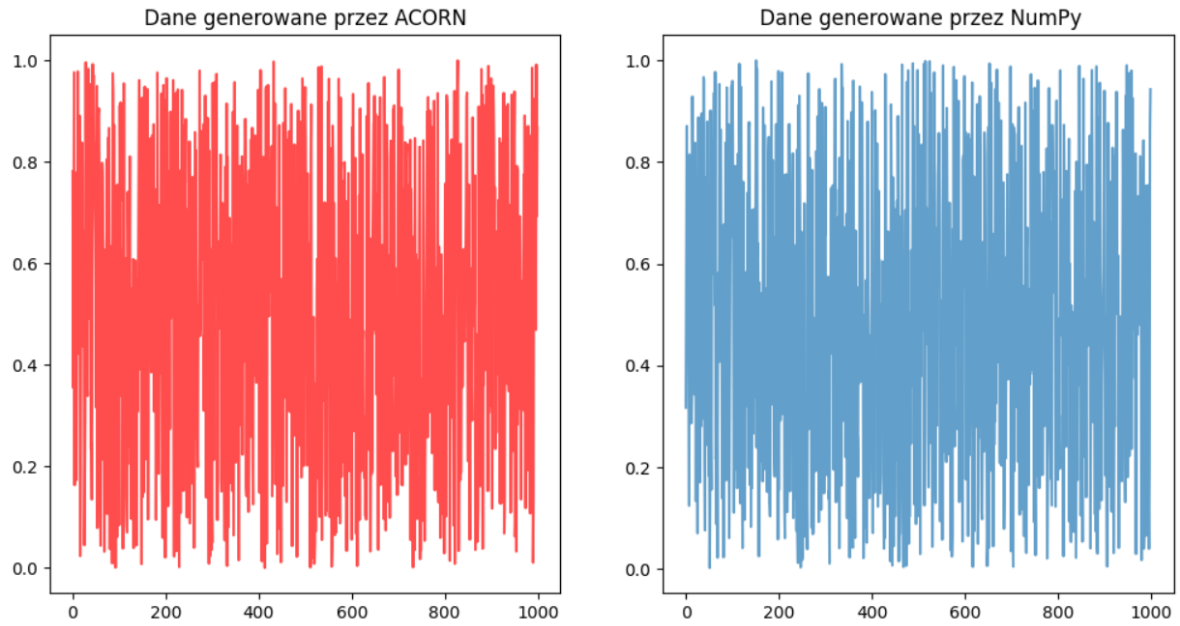
2.3 Wykresy

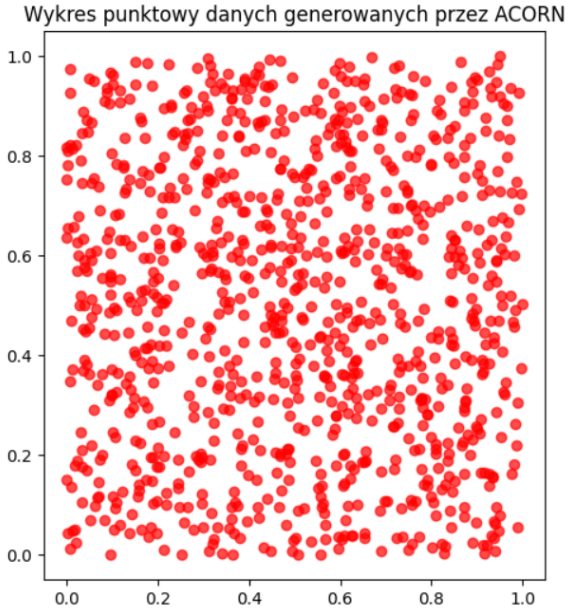
Dla sprawdzenia poprawności zaimplementowanego algorytmu porównujemy go do wbudowanego w python generatora liczb pseudolosowych z biblioteki numpy.

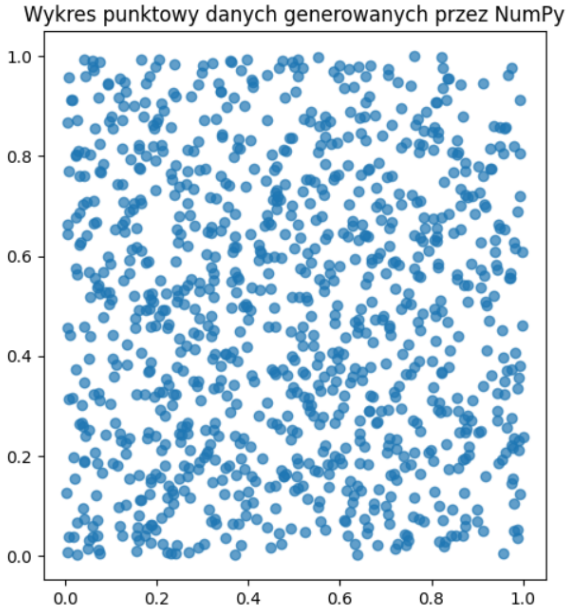
```
Y_numpy = [np.random.uniform(0, 1) for i in range(N)]

plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(Y_ACORN, color="r", alpha=0.7)
plt.title("Dane␣generowane␣przez␣ACORN")
plt.subplot(1, 2, 2)
plt.plot(Y_numpy, alpha=0.7)
plt.title("␣Dane␣generowane␣przez␣NumPy")
plt.show()

```

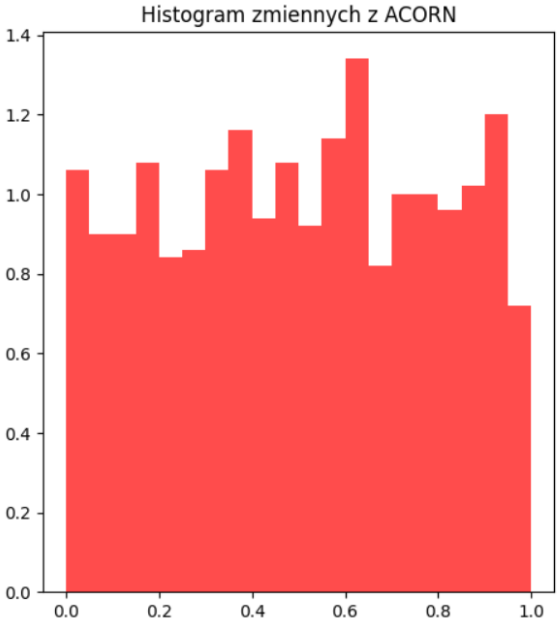
```
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.scatter(Y_ACORN[:-1], Y_ACORN[1:], color="r", alpha=0.7)
plt.title("Wykres punktowy danych generowanych przez ACORN")
plt.subplot(1, 2, 2)
plt.scatter(Y_numpy[:-1], Y_numpy[1:], alpha=0.7)
plt.title("Wykres punktowy danych generowanych przez NumPy")
plt.show()

plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.hist(Y_ACORN, bins=20, density=True, alpha=0.7, color="r")
plt.title("Histogram zmiennych z ACORN")
plt.subplot(1, 2, 2)
plt.hist(Y_numpy, bins=20, density=True, alpha=0.7)
plt.title("Histogram zmiennych z NumPy")
```

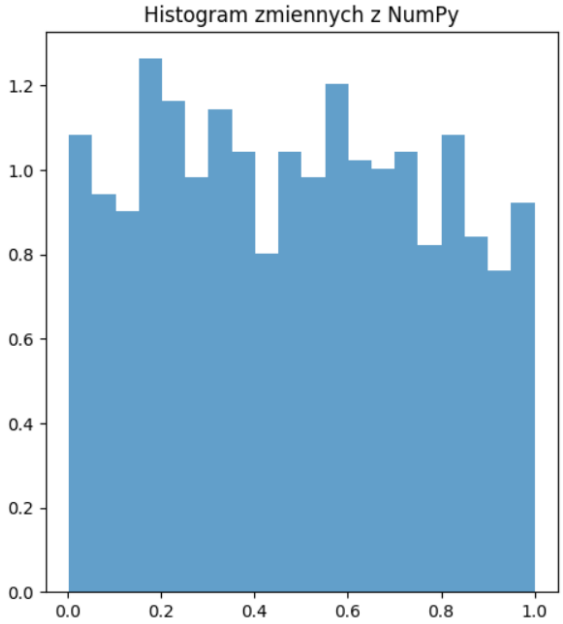
Listing 2: Sprawdzenie podobieństw Generatora ACORN i NumPy




Wykres punktowy danych generowanych przez ACORN
0.0
0.2
0.4
0.6
0.8
1.0


Wykres punktowy danych generowanych przez NumPy
0.0
0.2
0.4
0.6
0.8
1.0


Histogram zmiennych z ACORN
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4

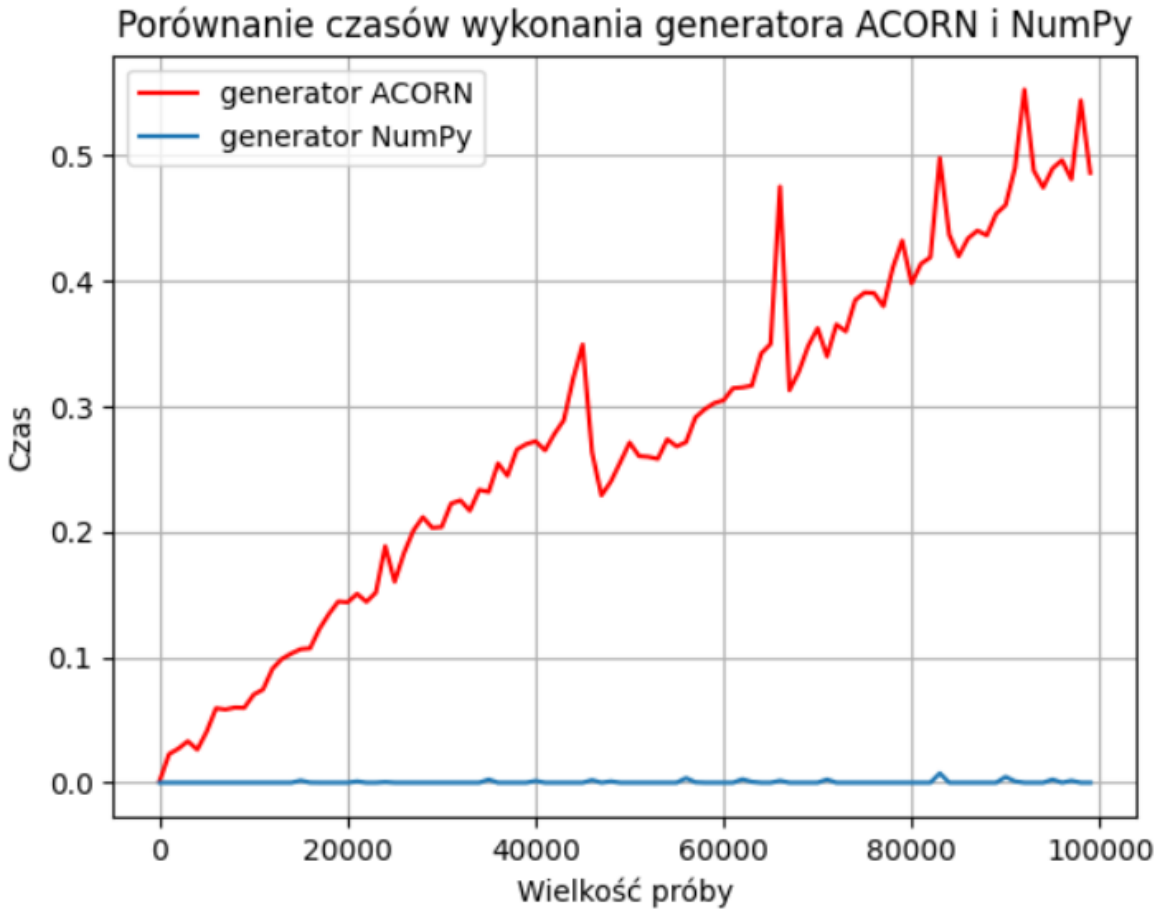

Histogram zmiennych z NumPy
0.0
0.2
0.4
0.6
0.8
1.0
1.2

2.4 Wydajność generatora

Porównujemy czas symulacji zaimplementowanego generatora z wbudowanym generatorem ACORN badając go dla różnego rozmiaru próbek.

```
N = np.arange(0, 100000, 1000)

acorn_times = []
numpy_times = []

for n in N:
    start = time.time()
    ACORN(n, k, M, Lag)
    t = time.time() - start
    acorn_times.append(t)

for n in N:
    start = time.time()
    np.random.uniform(0, 1, n)
    t = time.time() - start
    numpy_times.append(t)

plt.plot(N, acorn_times, color="r", label="generator␣ACORN")
plt.plot(N, numpy_times, label="generator␣NumPy")
plt.title("Por wnanie␣czas w␣wykonania␣generatora␣ACORN␣i␣NumPy")
plt.xlabel("Wielko Ż ␣pr by")
plt.ylabel("Czas")
plt.legend()
plt.grid()
plt.show()
```

Listing 3: Badanie wydajności generatora ACORN.



2.5 Wnioski

Na podstawie wygenerowanych wykresów można stwierdzić, że oba generatory liczb losowych mają dobre właściwości statystyczne. Wykresy punktowe dla obu generatorów pokazują równomierne rozmieszczenie punktów, co sugeruje niezależność generowanych wartości. Analizując rozkład jednostajny U(0,1), zauważamy, że histogramy obu generatorów wykazują równomierne rozłożenie wartości w przedziale od 0 do 1, co potwierdza, że liczby losowe są generowane zgodnie z oczekiwaniami.

Analizując wykres porównania czasów wykonania obu generatorów widzimy, że wraz ze wzrostem wielkości próby wzrasta czas generowania się próbek metodą ACORN, natomiast w bibliotece NumPy przyjmuje on wartości bardzo bliske 0. Stąd wnioskujemy, żegenerator NumPy jest bardziej wydajny oraz jest zdecydowanie szybszy.

3 Zad 2.

W poniższym zadaniu rozważymy kilka metod generowania rozkładu normalnego i porównamy ich efektywność.

3.1 Wstęp.

Generatory liczb pseudolosowych są kluczowymi narzędziami w informatyce, generującymi ciągi liczb, które naśladują losowość. Porównanie różnych generatorów pozwala ocenić ich jakość i odpowiednio dobrać do konkretnych zastosowań. Przyjrzymy się metodom: Boxa-Mullera, Marsaglii, Tuzina oraz Ziggurratu.

3.2 Opis i implementacja generatorów.

3.2.1 Generator Boxa - Mullera

Metoda ta generuje die niezależne próbki z rozkładu jednostajnego przekształcając matematycznie dwie próbki z rozkładu jednostajnego.

```python
def box_muller() -> tuple:
    """Funkcja generuje liczby pseudolosowe metod  Boxa-Mullera

    Returns:
        tuple: dwie liczby z rozk Ćadu noralnego

    Example:
        >>> box_muller()
        (-0.2450599351750017, 0.045448728066960306)
    """
    u1 = np.random.uniform(0, 1)
    u2 = np.random.uniform(0, 1)

    z1 = np.sqrt(-2 * np.log(u1)) * np.cos(2 * np.pi * u2)
    z2 = np.sqrt(-2 * np.log(u1)) * np.sin(2 * np.pi * u2)

    return z1, z2
```

Listing 4: Implementacja metody Boxa - Mullera

3.2.2 Generator Marsaglii

Metoda generuje dwie próbki z rozkładu normalnego wykorzystując transformację zmiennych losowych z rozkładu jednostajnego.

```python
def marsaglia(mi: int, sigma: int) -> tuple:
    """Generuje liczby pseudolosowe metod  Marsaglii.

    Args:
        mi:  rednia  rozk Ćadu normalnego.
        sigma: Odchylenie standardowe rozk Ćadu normalnego.

    Returns:
        tuple: Dwie liczby z rozk Ćadu normalnego o zadanej  Żredniej
           mi i odchyleniu standardowym sigma.

    Example:
        >>> marsaglia(0, 1)
        (0.3518546871028868, 2.4160008431319757)
    """
    u1 = np.random.uniform(-1, 1)
    u2 = np.random.uniform(-1, 1)

    s = u1**2 + u2**2
    while s >= 1 or s == 0:
        u1, u2 = 2 * np.random.rand(2) - 1
        s = u1**2 + u2**2

    z1 = u1 * np.sqrt(-2 * np.log(s) / s)
    z2 = u2 * np.sqrt(-2 * np.log(s) / s)

    return sigma * z1 + mi, sigma * z2 + mi
```

Listing 5: Implementacja metody Marsaglii

3.2.3 Generator Tuzin

Metoda generuje 12 niezależnych próbek z rozkładu jednostajnego, następnie zwraca liczbę, która jest ich sumą minus 6.

```python
def tuzin(mi: int, sigma: int) -> float:
    """Generuje liczb Ź pseudolosow  z rozk Ćadu normalnego metod  "
       tuzin".

    Args:
        mi:  rednia  rozk Ćadu normalnego.
        sigma: Odchylenie standardowe rozk Ćadu normalnego.

    Returns:
        float: Liczba z rozk Ćadu normalnego o zadanej  Żredniej  mi i
           odchyleniu standardowym sigma.
```

```
10
11      Example:
12          >>> tuzin(0, 1)
13          (0.3518546871028868, 2.4160008431319757)
14      """
15      U = [np.random.uniform() for i in range(12)]
16      X = np.sum(U) - 6
17      return sigma * X + mi
```

Listing 6: Implementacja metody Tuzin

3.2.4 Generator Zigguratu

Algorytm ziggurat jest algorytmem próbkowania odrzucającego. Losowo generuje punkt w rozkładzie nieco większym niż żądany rozkład, a następnie sprawdza, czy wygenerowany punkt znajduje się wewnątrz żądanego rozkładu. Jeśli nie, spróbuje ponownie.

```
1  def gestosc_normal(x: float) -> float:
2      """Zwraca warto Ż  g Źsto Żci prawdopodobie stwa standardowego
           rozk Ćadu normalnego w punkcie x.
3
4      Args:
5          x: Warto Ż  wej Żciowa.
6
7      Return:
8          float: Warto Żci funkcji g Źsto Żci prawdopodobie stwa
               standardowego rozk Ćadu normalnego w punkcie x.
9
10     Example:
11         >>> gestosc_normal(1)
12         0.24197072451914337
13     """
14     return (1 / np.sqrt(2 * np.pi)) * np.exp(-0.5 * x**2)
15
16
17 def odwrotna_gestosc_normal(y: float) -> float:
18     """Zwraca odwrotno Ż  funkcji dystrybuanty standardowego
           rozk Ćadu normalnego w punkcie y.
19
20     Args:
21         y: Warto Ż  wej Żciowa.
22
23     Returns:
24         float: Warto Żci odwrotno Żci funkcji dystrybuanty
               standardowego rozk Ćadu normalnego w punkcie y.
25
26     Example:
```

```
27          >>> odwrotna_gestosc_normal(0.24197072451914337)
28          1.0
29      """
30      return np.sqrt(-2 * np.log(np.sqrt(2 * np.pi) * y))
31
32
33  def przedzialy(N: int = 256, x0: float = 3.44) -> tuple:
34      """Generuje przedziaĆy do metody Ziggurata.
35
36      Args:
37          N: Liczba przedziaĆ w. DomyŻlnie 256.
38          x0: Warto Ż  pocz tkowa. DomyŻlnie 3.44.
39
40      Returns:
41          tuple: Dwie tablice zawieraj ce przedziaĆy (x) i
                odpowiadaj ce im g ŹstoŻci (y).
42
43      Example:
44          >>> przedzialy(16, 3.44)
45          (array([3.44      , 3.23222921, 3.0974141 , 2.99590161,
                2.91367136,
46                  2.84408838, 2.78347103, 2.72955708, 2.68085359,
                    2.63632174,
47                  2.59520857, 2.55695021, 2.52111306, 2.48735621,
                    2.4554066 ,
48                  2.42504197, 0.        ]),
49          array([0.00107467, 0.00214935, 0.0032931 , 0.00448664,
                0.00572062,
50                  0.00698942, 0.00828926, 0.00961742, 0.0109718 ,
                    0.0123508 ,
51                  0.01375308, 0.01517758, 0.0166234 , 0.01808976,
                    0.01957603,
52                  0.02108164]))
53      """
54      x = np.zeros(N + 1)
55      y = np.zeros(N)
56
57      x[0] = x0
58      x[-1] = 0
59      y[0] = gestosc_normal(x[0])
60
61      A = x[0] * y[0]
62      for i in range(N - 1):
63          y[i + 1] = A / x[i] + y[i]
64          x[i + 1] = odwrotna_gestosc_normal(y[i + 1])
65
66      return x, y
67
```

```
68
69  xs, ys = przedzialy()
70
71
72  def ziggurat(xs: np.array, ys: np.array, N: int = 256) -> float:
73      """Generuje liczby pseudolosowe metod  Ziggurata.
74
75      Args:
76          xs: Tablica przedzia Ć w.
77          ys: Tablica odpowiadaj cych im g Źsto Żci.
78          N: Liczba przedzia Ć w. Domy Żlnie 256.
79
80      Returns:
81          float: Liczba z rozk Ćadu normalnego.
82
83      Example:
84          >>> xs, ys = przedzialy()
85          >>> ziggurat(xs, ys)
86              -0.9590442042002225
87
88      """
89      while True:
90          i = np.random.randint(0, N - 1)
91          u0 = np.random.uniform(0, 1)
92          u1 = np.random.uniform(0, 1)
93
94          choice = np.random.choice([-1, 1])
95
96          x = u0 * xs[i]
97          y = ys[i] + u1 * (ys[i + 1] - ys[i])
98
99          if i == 0:
100             x = -np.log(u0) / xs[0]
101             y = -np.log(u1)
102             if 2 * y > x**2:
103                 return (x + xs[0]) * choice
104         else:
105             if x < xs[i + 1]:
106                 return x * choice
107
108             if y < gestosc_normal(x):
109                 return x * choice
```

Listing 7: Implementacja metody Zigguratu

3.3 Poprawność metod.

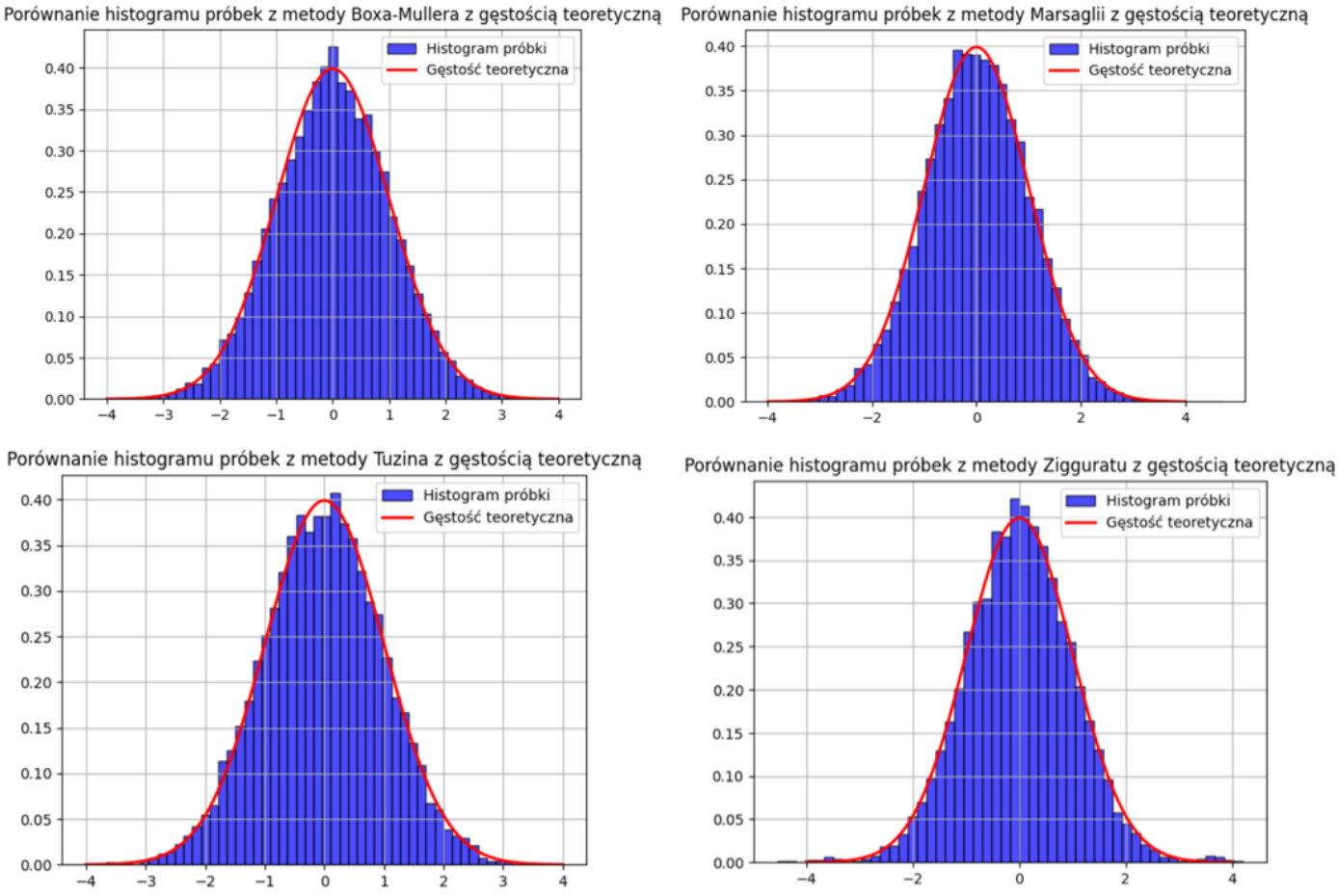
W celu sprawdzenia poprawności aimplementowanych metod generujemy 10000 prób z każdego rozkładu i porównujemy ich histogram z gęstością rozkładu normlanego.

```
samples_box_muller = []
for i in range(10000):
    bm = box_muller()
    samples_box_muller.append(bm[0])
    samples_box_muller.append(bm[1])

samples_marsaglia = []
for i in range(10000):
    m = marsaglia(0, 1)
    samples_marsaglia.append(m[0])
    samples_marsaglia.append(m[1])

samples_tuzin = [tuzin(0, 1) for _ in range(10000)]

samples_ziggurat = [ziggurat(xs, ys) for _ in range(10000)]

plt.hist(
    samples_box_muller,
    bins=50,
    density=True,
    alpha=0.7,
    color="blue",
    edgecolor="black",
    label="Histogram␣pr bki",
)
x = np.linspace(-4, 4, 1000)
plt.plot(x, gestosc_normal(x), color="red", lw=2, label="G Źsto Ż ␣
    teoretyczna")
plt.title("Por wnanie␣histogramu␣pr bek␣z␣metody␣Boxa-Mullera␣z␣
    g Źsto Żci ␣teoretyczn ")
plt.legend()
plt.grid()
plt.show()

plt.hist(
    samples_marsaglia,
    bins=50,
    density=True,
    alpha=0.7,
    color="blue",
    edgecolor="black",
    label="Histogram␣pr bki",
)
```

```
x = np.linspace(-4, 4, 1000)
plt.plot(x, gestosc_normal(x), color="red", lw=2, label="G Źsto Ż ␣
    teoretyczna")
plt.title("Por wnanie␣histogramu␣pr bek␣z␣metody␣Marsaglii␣z␣
    g Źsto Żci ␣teoretyczn ")
plt.legend()
plt.grid()
plt.show()

plt.hist(
    samples_tuzin,
    bins=50,
    density=True,
    alpha=0.7,
    color="blue",
    edgecolor="black",
    label="Histogram␣pr bki",
)
x = np.linspace(-4, 4, 1000)
plt.plot(x, gestosc_normal(x), color="red", lw=2, label="G Źsto Ż ␣
    teoretyczna")
plt.title("Por wnanie␣histogramu␣pr bek␣z␣metody␣Tuzina␣z␣
    g Źsto Żci ␣teoretyczn ")
plt.legend()
plt.grid()
plt.show()

plt.hist(
    samples_ziggurat,
    bins=50,
    density=True,
    alpha=0.7,
    color="blue",
    edgecolor="black",
    label="Histogram␣pr bki",
)
x = np.linspace(-4, 4, 1000)
plt.plot(x, gestosc_normal(x), color="red", lw=2, label="G Źsto Ż ␣
    teoretyczna")
plt.title("Por wnanie␣histogramu␣pr bek␣z␣metody␣Zigguratu␣z␣
    g Źsto Żci ␣teoretyczn ")
plt.legend()
plt.grid()
plt.show()
```

Listing 8: Sprawdzenie poprawności metod

Uzyskane w ten sposób histogramy wyglądają następująco:



Rysunek 1: Histogramy porównane z gęstością teoretyczną.

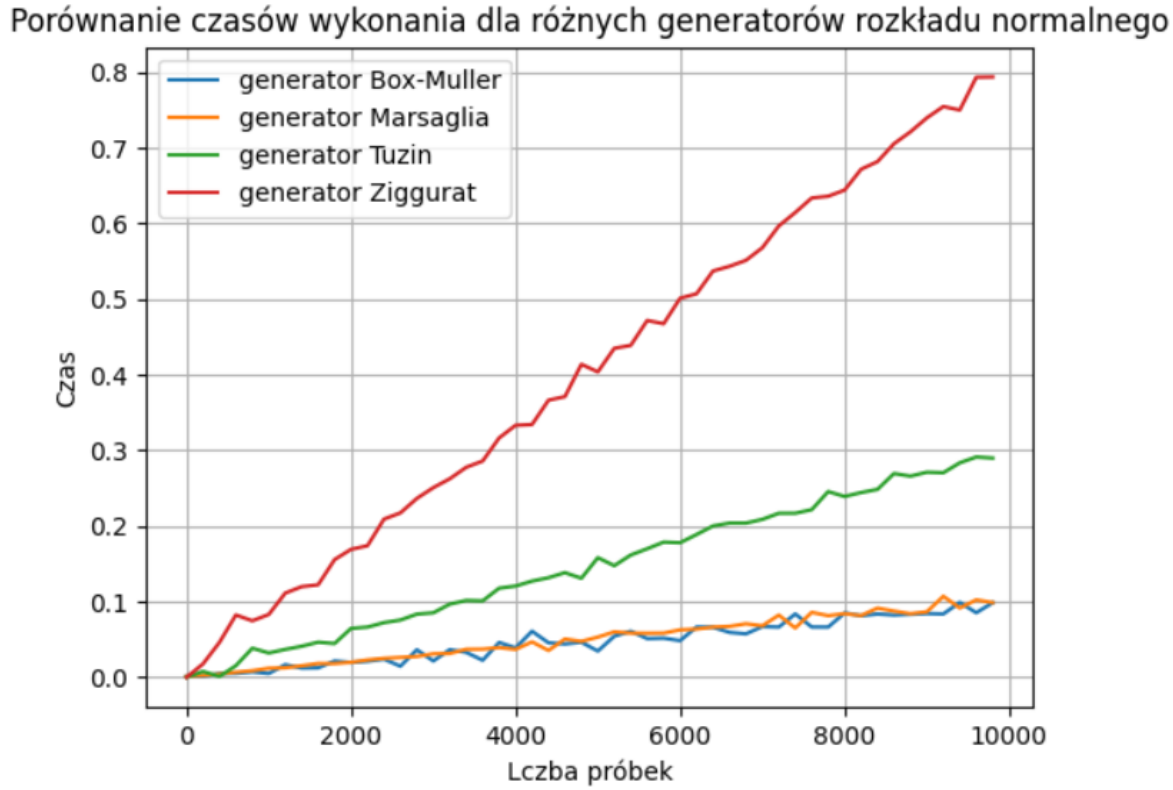
3.4 Wydajność metod.

W celu znalezienia najbardziej wydajnej metody badamy czas generowania się danych metod dla różnej wielkości próbek. Czasy te zostaną przedstawione na wykresie.

```
N = np.arange(0, 10000, 200)

box_muller_times = []
marsaglia_times = []
tuzin_times = []
ziggurat_times = []

for n in N:
```

```
    start = time.time()
    samples = [box_muller() for _ in range(n)]
    t = time.time() - start
    box_muller_times.append(t)

for n in N:
    start = time.time()
    samples = [marsaglia(0, 1) for _ in range(n)]
    t = time.time() - start
    marsaglia_times.append(t)

for n in N:
    start = time.time()
    samples = [tuzin(0, 1) for _ in range(n)]
    t = time.time() - start
    tuzin_times.append(t)

for n in N:
    start = time.time()
    samples = [ziggurat(xs, ys) for _ in range(n)]
    t = time.time() - start
    ziggurat_times.append(t)

plt.plot(N, box_muller_times, label="generator␣Box-Muller")
plt.plot(N, marsaglia_times, label="generator␣Marsaglia")
plt.plot(N, tuzin_times, label="generator␣Tuzin")
plt.plot(N, ziggurat_times, label="generator␣Ziggurat")
plt.xlabel("Lczba␣pr bek")
plt.ylabel("Czas")
plt.title("Por wnanie␣czas w␣wykonania␣dla␣r  nych␣generator w␣
    rozk Ćadu␣normalnego")
plt.legend()
plt.grid()
plt.show()
```

Listing 9: Porównanie wydajności metod



Rysunek 2: Porównanie wydajności metod.

3.5 Wnioski.

Na podstawie sporządzonych histogramów możemy stwierdzić, że każda z przedstawionych metod generowania rozkładu normalnego jest poprawna. Po badaniu wydajności najszybszymi metodami okazał się generator Boxa-Mullera oraz Marsaglii. Najdłużej generował się generator Ziggurat. Zauważamy również, że wszytkie funkcje czasu rosną liniowo.

4 Zad 3.

W niniejszym zadaniu będziemy rozważać wykorzystanie metod Monte Carlo w metodach redukcji wariancji

4.1 Wstęp.

Metody redukcji wariancji w metodach Monte Carlo są używane, aby zwiększyć efektywność i dokładność szacowania wartości oczekiwanych. Dwie popularne metody to:

Metoda odbić lustrzanych (antithetic variates),
Metoda zmiennej kontrolnej (control variates).

Wyniki Estymacji liczby π w zależności od metody zaprezentowane są na wykresie 1.

Porównanie wyników błędów oraz wariancji estymatorów każdej z metod odpowiednio na wykresach 2 i 3.

4.2 Opis Metod.

Metoda odbić lustrzanych:
polega na generowaniu par zmiennych losowych, które są od siebie zależne w sposób, który zmniejsza wariancję. Przykładowo, jeśli U jest losową zmienną z rozkładu jednostajnego na $[0, 1]$, to zmienną lustrzaną będzie $1 - U$.

Metoda zmiennej kontrolnej.
Metoda zmiennej kontrolnej polega na użyciu dodatkowej zmiennej losowej, której wartość oczekiwana jest znana, aby skorygować szacunki. Jeśli Y jest naszą oryginalną zmienną losową, a C jest zmienną kontrolną z wartością oczekiwaną μ_c to nowa zmienna losowa $Y' = Y - C + \mu_c$ ma mniejszą wariancję.

4.3 Metoda Monte Carlo.

Zadanie polega na obliczeniu całki:

$$I = \int_0^1 \frac{4}{1+x^2} dx$$

Ta całka reprezentuje pole pod krzywą $\frac{4}{1+x^2}$ od 0 do 1. Możemy wykorzystać metodę Monte Carlo do oszacowania tej całki poprzez generowanie losowych punktów x w przedziale $[0, 1]$ i obliczanie wartości funkcji w tych punktach.

Zaimplementujemy to zadanie używając zarówno:
metody odbić lustrzanych(Antithetic variates), jak i
metody zmiennej kontrolnej(Control variates).

```
Wynik Monte Carlo bez redukcji wariancji: 3.139094727575495
Wynik metody odbić lustrzanych: 3.1409554254321987
Wynik metody zmiennej kontrolnej: 3.141292028716415
```

Rysunek 3: Porównanie wyników.

4.4 Skrypt W pythonie.

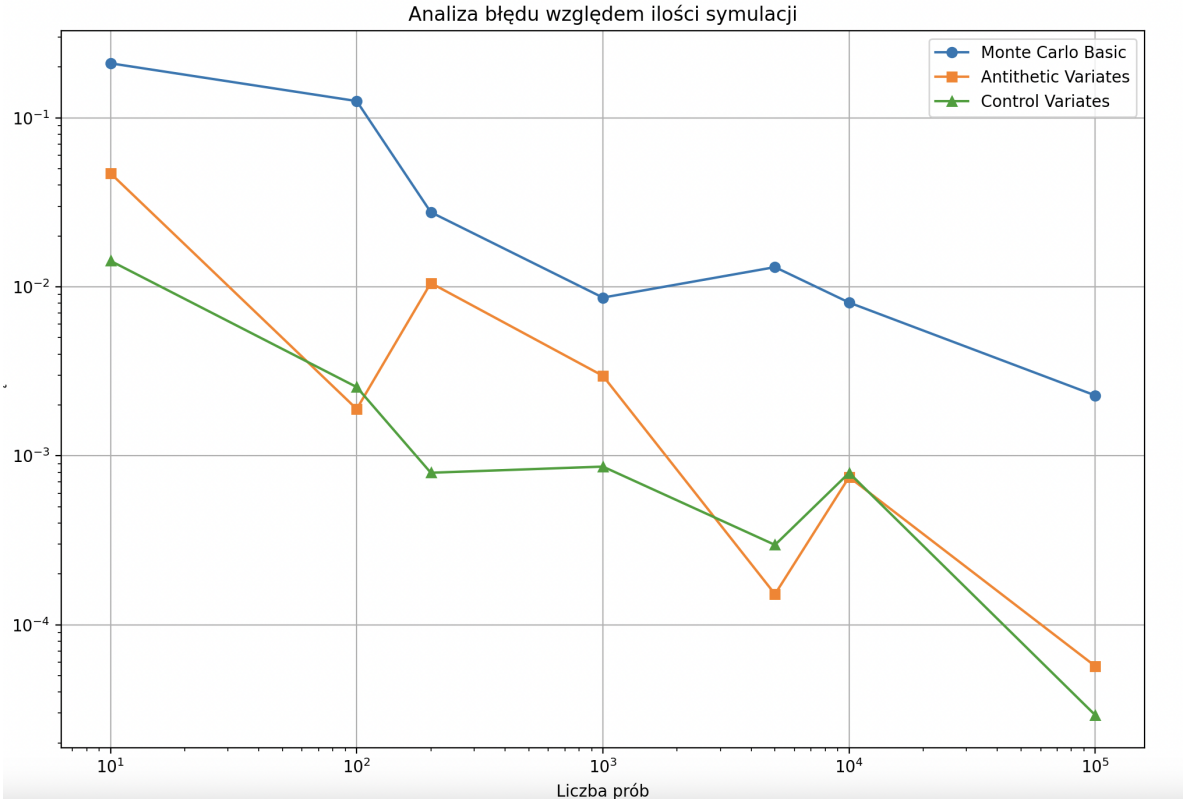
```python
import numpy as np
import matplotlib.pyplot as plt

# Funkcja podcalkowa
def f(x: np.ndarray) -> np.ndarray:
    """
    Funkcja podcalkowa.

    Args:
        x (np.ndarray): Array wartosci x.

    Returns:
        np.ndarray: Oblicza wartosc funkcji: 4 / (1 + x**2).
    """
    return 4 / (1 + x**2)


# Monte Carlo klasyczna
def monte_carlo_basic(n: int) -> float:
    """
    Monte Carlo calkowanie bez "redukcji wariancji."

    Args:
        n (int): Wielkosc proby.

    Returns:
        float: Szanowana wartosc calki.
    """
    x = np.random.uniform(0, 1, n)
    return np.mean(f(x))


# Metoda odbic lustrzanych
def monte_carlo_antithetic(n:int) -> float:
```

```
    """
     Calkowanie Monte Carlo przy uzyciu: antithetic variates method.

    Args:
        n (int): Wielkosc proby.

    Returns:
        float: Szanowana wartosc calki.
    """
    x = np.random.uniform(0, 1, n // 2)
    y = 1 - x
    return np.mean((f(x) + f(y)) / 2)


# Metoda zmiennej kontrolnej (uzyjemy funkcji liniowej jako zmiennej kontrolnej)
def monte_carlo_control_variate(n: int) -> float:
    """
    Monte Carlo integration using control variates method.

    Args:
        n (int): Wielkosc proby.

    Returns:
        float: Szanowana wartosc calki.
    """
    x = np.random.uniform(0, 1, n)
    control_variate = x
    mean_control_variate = 0.5
    y = f(x)
    alpha = np.cov(y, control_variate)[0, 1] / np.var(control_variate)
    return np.mean(y - alpha * (control_variate - mean_control_variate))


# Wielkosc proby:
n = 100000

# Zastosowanie funkcji
basic_result = monte_carlo_basic(n)
antithetic_result = monte_carlo_antithetic(n)
control_variate_result = monte_carlo_control_variate(n)

# Wyniki
print(f"Wynik Monte Carlo bez redukcji wariancji: {basic_result}")
print(f"Wynik metody odbic lustrzanych: {antithetic_result}")
print(f"Wynik metody zmiennej kontrolnej: {control_variate_result}")

```

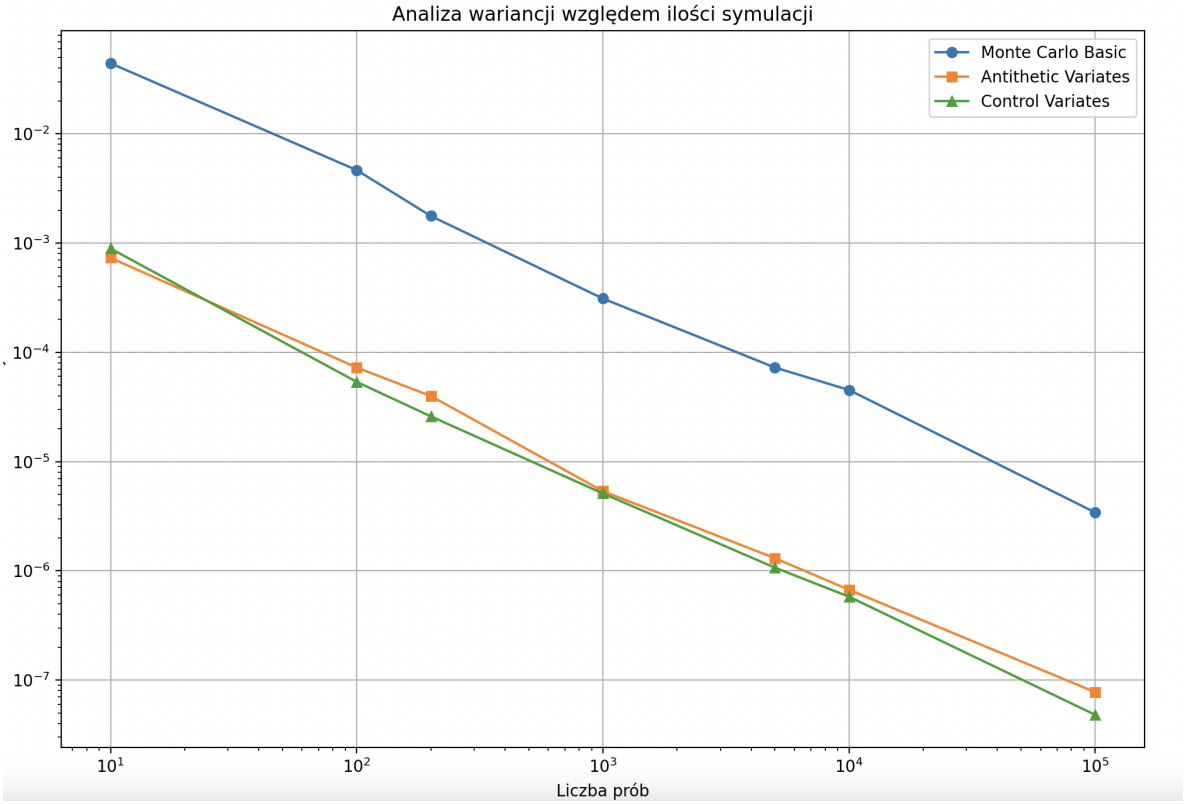
```
# Dokladna wartosc calki
exact_value = np.pi

# Liczby prob
n_values = [10, 100, 200, 1000, 5000, 10000, 100000]

# Wyniki i bledy dla kazdej z metod
basic_results = []
antithetic_results = []
control_variate_results = []

basic_errors = []
antithetic_errors = []
control_variate_errors = []

for n in n_values:
    basic_result = monte_carlo_basic(n)
    antithetic_result = monte_carlo_antithetic(n)
    control_variate_result = monte_carlo_control_variate(n)

    basic_error = np.abs(basic_result - exact_value)
    antithetic_error = np.abs(antithetic_result - exact_value)
    control_variate_error = np.abs(control_variate_result -
       exact_value)

    basic_results.append(basic_result)
    antithetic_results.append(antithetic_result)
    control_variate_results.append(control_variate_result)

    basic_errors.append(basic_error)
    antithetic_errors.append(antithetic_error)
    control_variate_errors.append(control_variate_error)

# Wykresy bledow
plt.figure(figsize=(12, 8))

plt.plot(n_values, basic_errors, marker="o", label="Monte␣Carlo␣Basic"
    )
plt.plot(n_values, antithetic_errors, marker="s", label="Antithetic␣
    Variates")
plt.plot(n_values, control_variate_errors, marker="^", label="Control␣
    Variates")

plt.xscale("log")
plt.yscale("log")
plt.xlabel("Liczba␣prob")
plt.ylabel("Blad")
plt.title("Analiza␣bledu␣wzgledem␣ilosci␣symulacji")
```

```
125 plt.legend()
126 plt.grid(True)
127 plt.show()
128
129 # Wariancje dla kazdej z metod
130 basic_variance = [np.var([monte_carlo_basic(n) for _ in range(100)])
        for n in n_values]
131 antithetic_variance = [np.var([monte_carlo_antithetic(n) for _ in
        range(100)]) for n in n_values]
132 control_variate_variance = [np.var([monte_carlo_control_variate(n) for
        _ in range(100)]) for n in n_values]
133
134 # Wykresy wariancji
135 plt.figure(figsize=(12, 8))
136
137 plt.plot(n_values, basic_variance, marker="o", label="Monte␣Carlo␣
        Basic")
138 plt.plot(n_values, antithetic_variance, marker="s", label="Antithetic␣
        Variates")
139 plt.plot(n_values, control_variate_variance, marker="^", label="
        Control␣Variates")
140
141 plt.xscale("log")
142 plt.yscale("log")
143 plt.xlabel("Liczba␣prob")
144 plt.ylabel("Wariancja")
145 plt.title("Analiza␣wariancji␣wzgledem␣ilosci␣symulacji")
146 plt.legend()
147 plt.grid(True)
148 plt.show()
149
150 # Tabela wynikow i bledow
151 print("␣")
152 print(f"{'Liczba␣prob':<15}␣{'Wynik␣(Basic)':<15}␣{'Blad␣(Basic)':<15}
        ␣{'Wynik␣(Antithetic)':<20}␣{'Blad␣(Antithetic)':<20}␣{'Wynik␣(
        Control␣Variate)':<25}␣{'Blad␣(Control␣Variate)':<25}")
153 for n, basic_result, basic_error, antithetic_result, antithetic_error,
         control_variate_result, control_variate_error in zip(n_values,
        basic_results, basic_errors, antithetic_results, antithetic_errors,
         control_variate_results, control_variate_errors):
154     print(f"{n:<15}␣{basic_result:<15.10f}␣{basic_error:<15.10f}␣{
            antithetic_result:<20.10f}␣{antithetic_error:<20.10f}␣{
            control_variate_result:<25.10f}␣{control_variate_error:<25.10f}
            ")
```

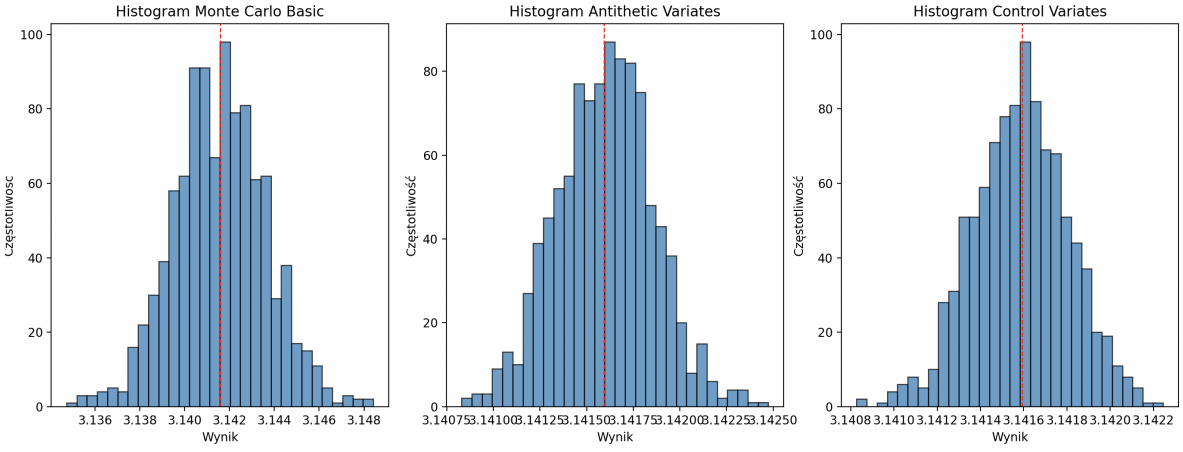
Listing 10: Skrypt do Zad 3.



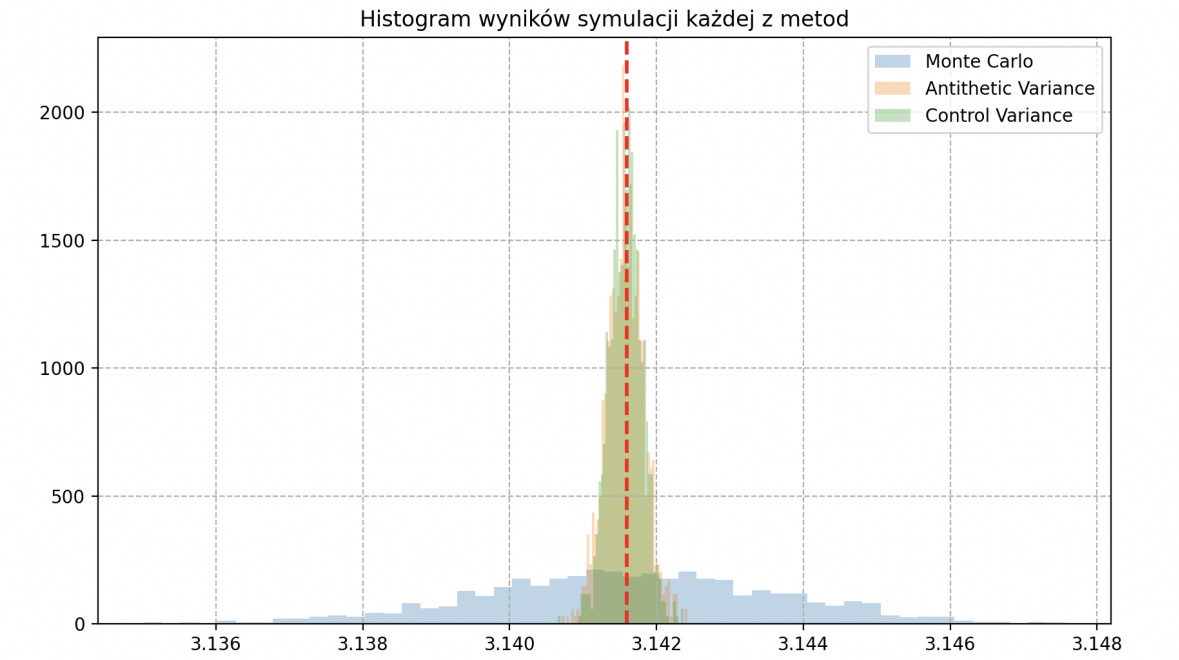
Rysunek 4: Wykres Błędu



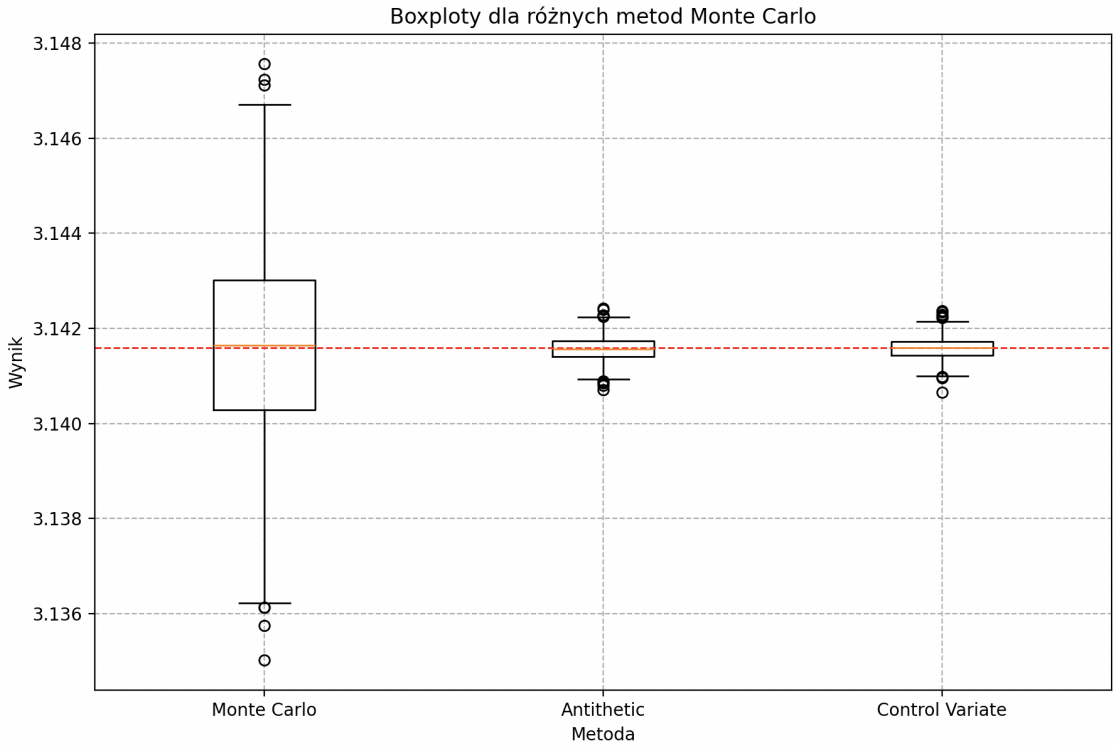
Rysunek 5: Porównanie Wariancji



Rysunek 6: Porównanie Histogramów każdej z metod.



Rysunek 7: Histogramy na jendym.



Rysunek 8: Boxploty każdej z metod.

4.5 Wnioski.

Szacowanie liczby π:
Metoda Monte Carlo daje przybliżenie liczby π poprzez losowanie próbek i obliczanie średniej wartości funkcji $\frac{4}{1+x^2}$. Wyniki zależą od liczby symulacji (im więcej próbek, tym dokładniejsze oszacowanie).

Redukcja wariancji metodą odbić lustrzanych: Wprowadzenie par antytetycznych zmiennych losowych (antithetic variates) znacząco zmniejsza wariancję estymatora. Wyniki uzyskane za pomocą tej metody są bardziej stabilne i dokładniejsze dla tej samej liczby symulacji w porównaniu do standardowej metody Monte Carlo.

Analiza błędu: Wykresy pokazują, że metoda odbić lustrzanych redukuje błąd szybciej niż standardowa metoda Monte Carlo. Przy dużych liczbach symulacji i więcej metoda odbić lustrzanych daje znacznie mniejsze błędy, co widać na wykresach 2 i 3.

5 Zad 4.

5.1 Wstęp

W zadaniu czwartym analizujemy warunkową wartość oczekiwaną zmiennej Y warunkowanej zmienną X. Wartość oczekiwana $f(X)$ zmiennej Y spełnia właściwość:

$$\mathbb{E}(Y|X=x) = f(x) = \arg\min_g \mathbb{E}\left((Y-g(X))^2\right).$$

Jest to najlepsze przybliżenie w sensie L^2 zmiennej Y korzystające z danych pochodzących ze zmiennej X.

5.2 Część pierwsza

Rozważamy sytuację gdy X i Y są zmiennymi niezależnymi, a $\mathbb{E}(Y) = 0$. W takim przypadku dla Z zdefiniowanego jako: $Z = XY + \sin X$ zachodzi $\mathbb{E}(Z|X) = \sin X$. Aby to sprawdzić przeprowadzo następujące kroki:

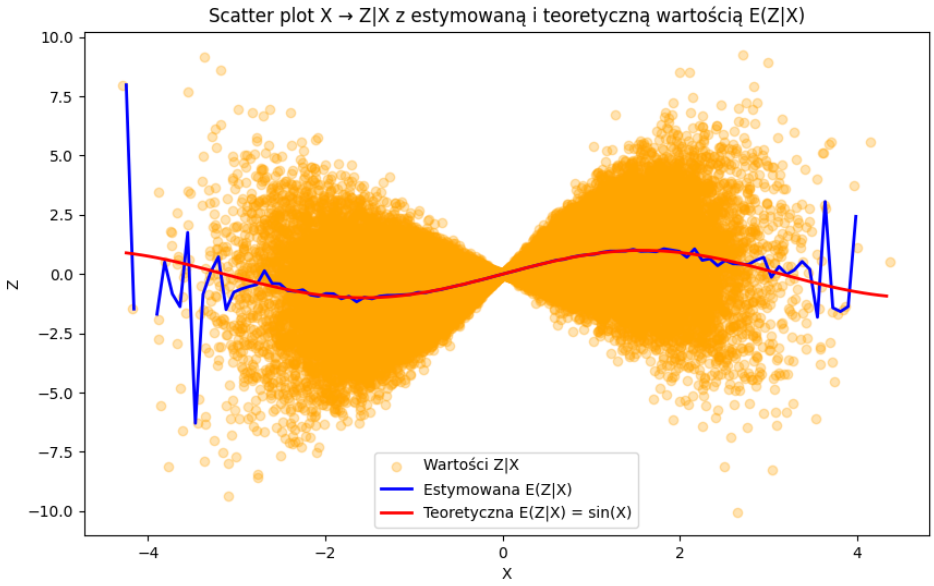
- Wygenerowano zmienne X i Y i.i.d $\sim \mathcal{N}(0,1)$.
- Obliczono $Z = XY + \sin X$.
- Obliczono warunkową wartość oczekiwaną $\mathbb{E}(Z|X)$ w przedziałach.
- Obliczono środki przedziałów.
- Wyliczono teoretyczną wartość $\mathbb{E}(Z|X) = \sin X$ w środkach przedziałów.
- Stworzono wykres typu scatterplot $X \mapsto Z|X$
- Naniesiono na wykres estymowaną $\mathbb{E}(Z|X)$ oraz teoretyczną $\mathbb{E}(Z|X) = \sin X$

Poniżej znajduje się kod w języku Python, który realizuje powyższe kroki i rysuje odpowiednie wykresy:

```
    def plot_conditional_expectation(
    num_samples: int = 100000, num_bins: int = 100
) -> None:
    """
```

```
    Plots the conditional expectation E(Z|X) using random samples X and Y, where Z = X * Y + sin(X).
    Compares the estimated E(Z|X) with the theoretical value sin(X).

    Parameters:
    num_samples (int): Number of random samples to generate for X and Y.
    num_bins (int): Number of bins to use for estimating E(Z|X).

    Returns:
    None

    Example usage:
    plot_conditional_expectation(num_samples=100000, num_bins=100)
    """

    # Generate random samples for X and Y
    X = np.random.standard_normal(num_samples)
    Y = np.random.standard_normal(num_samples)

    # Calculate Z based on the formula Z = X * Y + sin(X)
    Z = X * Y + np.sin(X)

    # Calculate the conditional expectation E(Z|X) in bins
    # 'binned_statistic' divides X data into bins and computes the mean of Z in each bin
    bin_means, bin_edges, _ = binned_statistic(X, Z, statistic="mean", bins=num_bins)

    # Calculate the bin centers
    bin_centers = (bin_edges[:-1] + bin_edges[1:]) / 2

    # The theoretical value E(Z|X) = sin(X) at the bin centers
    E_Z_given_X_theoretical = np.sin(bin_centers)

    # Create scatter plot X -> Z|X
    plt.figure(figsize=(10, 6))  # Set plot size
    plt.scatter(
        X, Z, alpha=0.3, label="Values of Z|X", color="orange"
    )  # Plot points (X, Z)
    plt.plot(
        bin_centers, bin_means, "b-", label="Estimated E(Z|X)", linewidth=2
    )  # Plot estimated E(Z|X)
    plt.plot(
        bin_centers,
        E_Z_given_X_theoretical,
        "r-",
```

```
        label="Theoretical E(Z|X) = sin(X)",
        linewidth=2,
    )  # Plot theoretical E(Z|X)
    plt.xlabel("X")  # X-axis label
    plt.ylabel("Z")  # Y-axis label
    plt.title(
        "Scatter plot X     Z|X with estimated and theoretical value E
           (Z|X)"
    )  # Plot title
    plt.legend()  # Add legend
    plt.show()  # Display plot


plot_conditional_expectation()
```



Rysunek 9: Scatter plot X → Z—X z estymowaną i teoretyczną wartością E(Z—X)

5.3 Część druga

Rozważamy sytuację, gdy N jest procesem Poissona o intensywności λ. Dla $T \geqslant t \geqslant 0$ zachodzi $\mathbb{E}(N_t|N_T) = \frac{tN_T}{T}$. Oznacza to, że zakładając przybywanie klientów do sklepu zgodnie z procesem Poissona i mając dane na temat dotychczasowej liczby klientów w sklepie w chwili T (czyli N_T), najlepszym przybliżeniem liczby klientów w chwili $t < T$ równej N_t jest $\frac{tN_T}{T}$. Aby to sprawdzić, przeprowadzono następujące kroki:

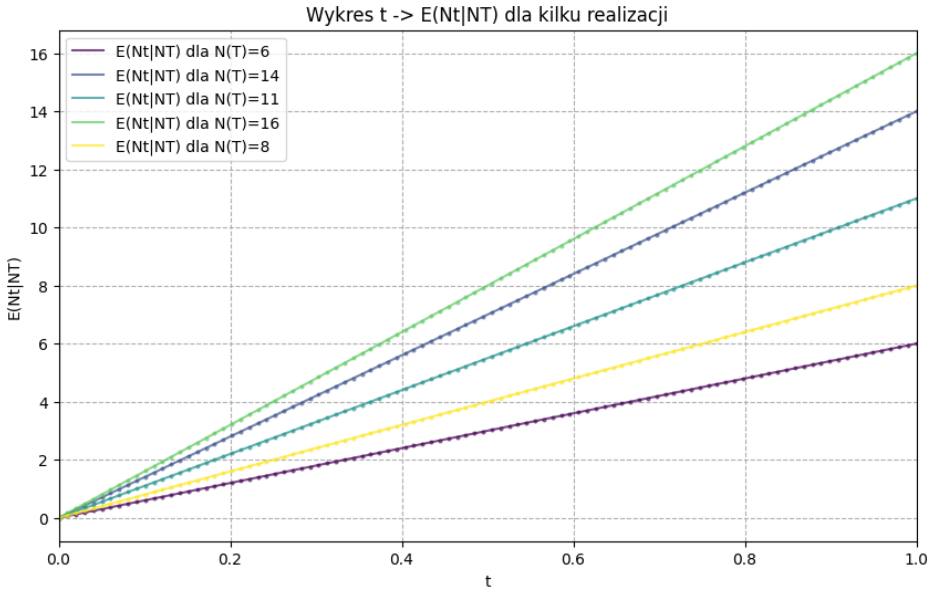
- Przyjęto proces Poissona o intensywności λ.
- Wygenerowano losowe próbki N_T z rozkładu Poissona dla chwili T.
- Obliczono wartość oczekiwaną $\mathbb{E}(N_t|N_T)$ jako $\frac{tN_T}{T}$ dla różnych wartości t w przedziale $[0, T]$.
- Stworzono wykres zależności $t \mapsto \mathbb{E}(N_t|N_T)$ dla kilku możliwych realizacji N_T.
- Dla każdej realizacji procesu Poissona wygenerowano czas zdarzeń i odpowiadające im liczby zdarzeń.
- Stworzono wykres krokowy dla indywidualnych realizacji procesu Poissona.
- Na wykresie naniesiono zarówno estymowaną wartość $\mathbb{E}(N_t|N_T)$, jak i rzeczywiste realizacje procesu Poissona.

Poniżej znajduje się kod w języku Python, który realizuje powyższe kroki i rysuje odpowiednie wykresy:

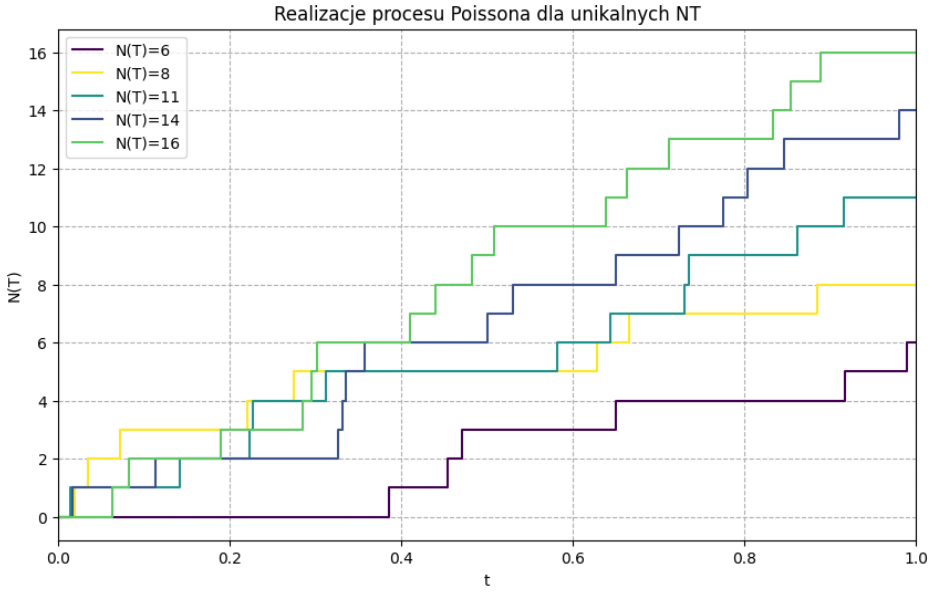
```
    def plot_E_Nt_given_NT(
    T: float = 1,
    lambda_intensity: float = 10,
    num_realizations: int = 5,
    line_style: str = "solid",
) -> None:
    """
    Plots the expected value E(Nt|NT) for a Poisson process with
       intensity ż and individual realizations of this process.

    Parameters:
    T (float): Total time period for the Poisson process.
    lambda_intensity (float): Intensity (rate) of the Poisson process.
    num_realizations (int): Number of different realizations of the
       Poisson process to be plotted.
    line_style (str): Line style for the plots ('solid' or 'dashed').

    Returns:
    None

    Example usage:
    plot_E_Nt_given_NT(T=1, lambda_intensity=10, num_realizations=5,
       line_style='solid')
    """

    # Generate time values from 0 to T
    t_values = np.linspace(0, T, 100)
```

```
    # Generate color palette for different realizations
    colors = plt.cm.viridis(np.linspace(0, 1, num_realizations))

    # Set to store unique NT samples
    NT_samples = set()

    # Initialize the plot
    plt.figure(figsize=(10, 6))
    color_map = {}

    # Loop for each realization
    for i in range(num_realizations):
        # Generate a unique sample from the Poisson distribution for
            NT
        NT_sample = np.random.poisson(lambda_intensity * T)
        while NT_sample in NT_samples:
            NT_sample = np.random.poisson(lambda_intensity * T)
        NT_samples.add(NT_sample)

        color_map[NT_sample] = colors[i]

        # Calculate the expected value E(Nt|NT)
        E_Nt_given_NT_sample = (t_values * NT_sample) / T
        color = color_map[NT_sample]

        # Plot the expected value E(Nt|NT)
        plt.plot(
            t_values,
            E_Nt_given_NT_sample,
            linestyle=line_style,
            alpha=0.7,
            color=color,
            label=f"E(Nt|NT) for N(T)={NT_sample}",
        )
        plt.plot(
            t_values, E_Nt_given_NT_sample, "o", markersize=2, alpha
                =0.5, color=color
        )

    # Plot settings
    plt.xlabel("t")
    plt.ylabel("E(Nt|NT)")
    plt.xlim(0, T)
    plt.title("Plot t -> E(Nt|NT) for several realizations")
    plt.grid(True, linestyle="--")
    plt.legend(loc="upper left")
    plt.show()

```

```
    # Initialize the plot for the Poisson process realizations
    plt.figure(figsize=(10, 6))
    for NT_sample in NT_samples:
        # Generate event times and corresponding event counts
        t_realization = np.append(0, np.sort(np.random.uniform(0, T,
            NT_sample)))
        Nt_realization = np.arange(0, NT_sample + 1)

        color = color_map[NT_sample]
        plt.step(
            np.append(t_realization, T),
            np.append(Nt_realization, NT_sample),
            where="post",
            label=f"N(T)={NT_sample}",
            color=color,
        )

    # Plot settings for the Poisson process realizations
    plt.xlabel("t")
    plt.ylabel("N(T)")
    plt.xlim(0, T)
    plt.title("Realizations of the Poisson process for unique NT")
    plt.grid(True, linestyle="--")
    plt.legend(loc="upper left")
    plt.show()


# Call the function with default parameters
plot_E_Nt_given_NT(num_realizations=5, line_style="solid")
```



Rysunek 10: Wykres t $->$ $E(N_t|N_T)$ dla kilku realizacji)



Rysunek 11: Realizacje procesu Poissona dla unikalnych NT

5.4 Część trzecia

Rozważamy sytuację, gdy N jest procesem Poissona o intensywności λ. Dla $t \geqslant s \geqslant 0$ zachodzi $\mathbb{E}(N_t|\mathcal{F}_s) = N_s + \lambda(t-s)$, gdzie $\mathcal{F}_s$ to filtracja naturalna procesu N_s. Oznacza to, że zakładając przybywanie klientów do sklepu zgodnie z procesem Poissona i mając dane na temat dotychczasowej liczby klientów w sklepie w każdej chwili ω spełniającej $0 \leqslant \omega \leqslant s$ (czyli $\mathcal{F}_s$), najlepszym przybliżeniem liczby klientów w chwili $t \geqslant s$ jest $N_s + \lambda(t-s)$. Aby to sprawdzić, przeprowadzono następujące kroki:

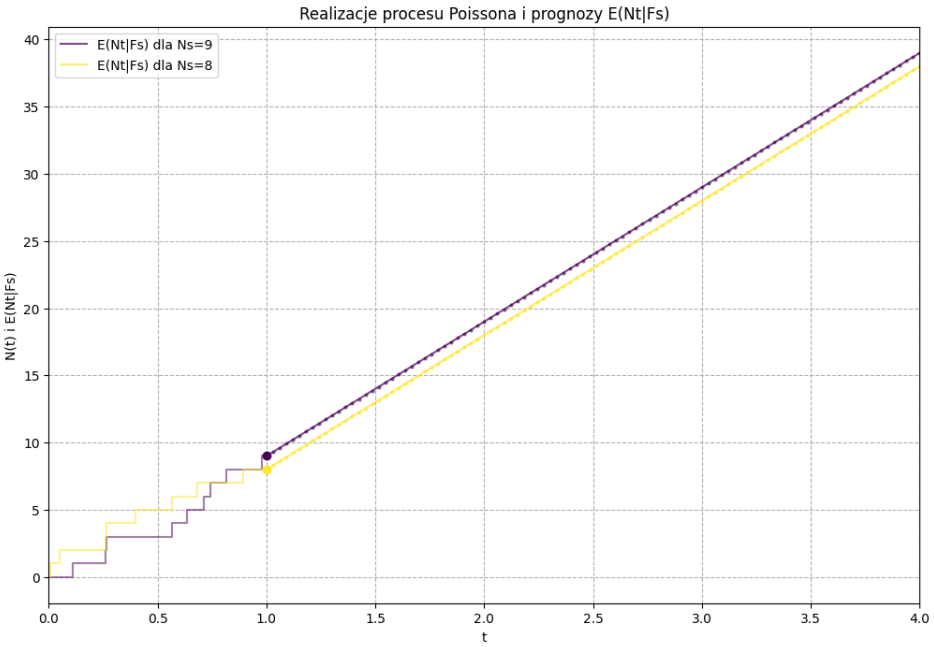
- Przyjęto proces Poissona o intensywności λ.
- Wygenerowano losowe próbki N_s z rozkładu Poissona dla chwili s.
- Obliczono wartość oczekiwaną $\mathbb{E}(N_t|\mathcal{F}_s)$ jako $N_s + \lambda(t-s)$ dla różnych wartości t w przedziale $[s, T]$.
- Dla każdej realizacji procesu Poissona wygenerowano czas zdarzeń i odpowiadające im liczby zdarzeń do czasu s.
- Stworzono wykres zależności $t \mapsto \mathbb{E}(N_t|\mathcal{F}_s)$ dla kilku możliwych realizacji N_s.
- Naniesiono na wykres zarówno estymowaną wartość $\mathbb{E}(N_t|\mathcal{F}_s)$, jak i rzeczywiste realizacje procesu Poissona.

Poniżej znajduje się kod w języku Python, który realizuje powyższe kroki i rysuje odpowiednie wykresy:

```
import numpy as np
import matplotlib.pyplot as plt

def plot_E_Nt_given_Fs_10_steps(
    s: float = 1, lambda_intensity: float = 10, num_realizations: int
       = 5, T: float = 4
) -> None:
    """
    Plots the expected value E(Nt|Fs) for a Poisson process with
       intensity ż and individual realizations of this process.

    Parameters:
    s (float): Time at which the filtration Fs is considered.
    lambda_intensity (float): Intensity (rate) of the Poisson process.
    num_realizations (int): Number of different realizations of the
       Poisson process to be plotted.
    T (float): Total time period for the Poisson process.

```

```
    Returns:
    None

    Example usage:
    plot_E_Nt_given_Fs_10_steps(s=1, lambda_intensity=10,
       num_realizations=5, T=4)
    """

    # Generate time values from s to T
    t_values = np.linspace(s, T, 100)
    # Generate color palette for different realizations
    colors = plt.cm.viridis(np.linspace(0, 1, num_realizations))
    # Set to store unique NT samples
    NT_samples = set()
    color_map = {}

    # Initialize the plot
    plt.figure(figsize=(12, 8))

    # Loop for each realization
    for i in range(num_realizations):
        # Generate a unique sample from the Poisson distribution for
           NT
        NT_sample = np.random.poisson(lambda_intensity * s)
        while NT_sample in NT_samples:
            NT_sample = np.random.poisson(lambda_intensity * s)
        NT_samples.add(NT_sample)

        # Generate event times and corresponding event counts up to
           time s
        t_realization = np.append(0, np.sort(np.random.uniform(0, s,
           NT_sample)))
        Nt_realization = np.arange(0, NT_sample + 1)

        # Assign color to the current realization
        color = colors[i]
        color_map[NT_sample] = color

        # Calculate the expected value E(Nt|Fs)
        E_Nt_given_Fs = NT_sample + lambda_intensity * (t_values - s)

        # Plot the trajectory up to 10 steps
        plt.step(
            np.append(t_realization, s),
            np.append(Nt_realization, NT_sample),
            where="post",
            alpha=0.5,
            color=color,
```

```
        )
        plt.scatter([s], [NT_sample], color=color, zorder=5)

        # Plot the forecast
        plt.plot(
            t_values,
            E_Nt_given_Fs,
            linestyle="solid",
            alpha=0.7,
            color=color,
            label=f"E(Nt|Fs) for Ns={NT_sample}",
        )
        plt.plot(t_values, E_Nt_given_Fs, "o", markersize=2, alpha
           =0.5, color=color)

    # Plot settings
    plt.xlabel("t")
    plt.ylabel("N(t) and E(Nt|Fs)")
    plt.xlim(0, T)
    plt.title("Poisson process realizations and forecasts E(Nt|Fs)")
    plt.grid(True, linestyle="--")
    plt.legend(loc="upper left")
    plt.show()

# Call the function with default parameters
plot_E_Nt_given_Fs_10_steps(num_realizations=2, T=4)
```



Rysunek 12: Realizacje procesu Poissona i prognozy E(Nt—Fs)

5.5 Wnioski

W przeprowadzonym zadaniu analizowaliśmy warunkową wartość oczekiwaną zmiennej Y warunkowanej zmienną X oraz zachowanie procesu Poissona w różnych kontekstach. Poniżej przedstawiono główne wnioski z poszczególnych części zadania:

5.5.1 Część pierwsza

Analiza zmiennych niezależnych X i Y pokazała, że dla $Z = XY + \sin X$ rzeczywiście zachodzi $\mathbb{E}(Z|X) = \sin X$. Symulacje potwierdziły teoretyczne oczekiwania:

- Generowane losowe próbki X i Y pozwoliły na obliczenie zmiennej Z zgodnie z założonym wzorem.
- Obliczone wartości warunkowej wartości oczekiwanej $\mathbb{E}(Z|X)$ były zgodne z teoretycznymi wartościami $\sin X$.
- Wykres typu scatterplot $X \mapsto Z|X$ jasno pokazał zgodność pomiędzy estymowanymi a teoretycznymi wartościami $\mathbb{E}(Z|X)$.

5.5.2 Część druga

Analiza procesu Poissona o intensywności λ dla $t < T$ wykazała, że wartość oczekiwana $\mathbb{E}(N_t|N_T)$ jest zgodna z teoretycznym modelem:

- Generowane losowe próbki N_T dla chwili T pozwoliły na estymację wartości N_t dla różnych chwil t.
- Obliczone wartości $\mathbb{E}(N_t|N_T)$ jako $\frac{tN_T}{T}$ były zgodne z przewidywaniami teoretycznymi.
- Wykresy przedstawiające zależność $t \mapsto \mathbb{E}(N_t|N_T)$ dla różnych realizacji N_T potwierdziły teoretyczne oczekiwania.

5.5.3 Część trzecia

Analiza procesu Poissona o intensywności λ dla $t \geqslant s \geqslant 0$ wykazała zgodność wartości oczekiwanych $\mathbb{E}(N_t|\mathcal{F}_s)$ z modelem teoretycznym:

- Generowane losowe próbki N_s dla chwili s pozwoliły na estymację wartości N_t dla różnych chwil t w przedziale $[s, T]$.
- Obliczone wartości $\mathbb{E}(N_t|\mathcal{F}_s)$ jako $N_s + \lambda(t - s)$ były zgodne z przewidywaniami teoretycznymi.
- Wykresy przedstawiające zależność $t \mapsto \mathbb{E}(N_t|\mathcal{F}_s)$ dla różnych realizacji N_s potwierdziły teoretyczne oczekiwania.

Podsumowując, przeprowadzone symulacje i analizy potwierdziły teoretyczne modele dotyczące warunkowej wartości oczekiwanej oraz procesów Poissona. Wyniki symulacji były zgodne z przewidywaniami teoretycznymi, co świadczy o poprawności zastosowanych metod oraz modeli. Zadanie to pozwoliło na praktyczne zweryfikowanie teoretycznych koncepcji za pomocą symulacji komputerowych.

6 Zad 5.

6.1 Wstęp

W zadaniu piątym zajmiemy się analizą procesu ruiny w modelu Cramera-Lundberga. Ten model jest stosowany do opisu procesów ekonomicznych na małą skalę, takich jak

te zachodzące wewnątrz pojedynczego przedsiębiorstwa. Załóżmy, że $X_t$to proces ruiny, czyli stan finansowy (budżet) pewnej firmy ubezpieczeniowej. Model ten jest określony za pomocą poniższego wzoru:

$$X_t = u + ct - \sum_{i=0}^{N_t} \xi_i,$$

gdzie:

- $u > 0$ jest początkowym kapitałem firmy,
- $c > 0$ to stały przychód przypadający na jednostkę czasu,
- $\xi_i > 0$ jest zmienną losową z rozkładu wykładniczego $Exp(\eta)$, i odpowiada np. wielkości strat
- N_t jest procesem Poissona o intensywości λ

gdzie $\xi_i \perp \xi_j$ dla $i \neq j$, $\mathbb{E}(\xi_i) = \eta$. Czasem ruiny klasycznej nazywamy zmienną $\tau = \inf\{t > 0 | X_t < 0\}$. Prawdopodobieństwem ruiny w czasie nieskończonym nazywamy funkcję $\psi(u, c) = \mathbb{P}(\tau < \infty)$. Wzór Pollaczka-Chinczyna mówi, że

$$\psi(u, c) = \frac{\eta\lambda}{c} e^{-\left(\frac{1}{\eta} - \frac{\lambda}{c}\right)u}.$$

6.2 Część pierwsza

Aby zweryfikować symulacyjnie te wyniki, skorzystamy z pomocniczego prawdopodobieństwa ruiny w czasie skończonym T, tj. $\Psi(u, c, T) = \mathbb{P}(\tau < T)$, $T > 0$, dla odpowiednio dużego T. Wykonano następujące kroki:

- Wygenerowano proces Poissona za pomocą funkcji poisson_process.
- Zasymulowano proces ruiny kapitału za pomocą funkcji proces_ruiny.
- Zaimplementowano wzór Pollaczka-Khinchina do obliczenia prawdopodobieństwa ruiny
- Obliczono prawdopodobieństwo ruiny kapitału metodą Monte Carlo za pomocą funkcji ruin_prob_estim.

- Porównano wyniki symulacji z wartościami teoretycznymi uzyskanymi z wzoru Pollaczka-Chinczyna za pomocą funkcji pollaczeck_khinchine.

```
    def poisson_process(rate: float, time_duration: float) -> tuple:
    """
    Generates a Poisson process based on the given intensity rate and
       time duration.

    The function generates the number of events from a Poisson
       distribution and their occurrence times as cumulative sums
    from an exponential distribution. A dictionary is also created
       where the keys are event times and the values
    are the event numbers.

    Parameters:
        rate (float): Intensity rate of the Poisson process.
            - Must be positive.
        time_duration (float): Duration of the simulation.
            - Must be positive.

    Returns:
        tuple: A tuple containing:
            - num_events (int): Number of events generated during the
               simulation.
            - event_times (np.ndarray): Array of event occurrence
               times.
                - Time from the start of the simulation to each event.
            - N_t (dict): Dictionary where the keys are event times
               and the values are the event numbers.

    Example usage:
        >>> rate = 0.5
        >>> time_duration = 10
        >>> num_events, event_times, N_t = poisson_process(rate,
           time_duration)
        >>> print(num_events)  # Example result: 5
        >>> print(event_times)  # Example result: [0.52529719
           2.83973255 4.71950743 7.49885524 7.9769648 ]
        >>> print(N_t)  # Example result: {0.5252971895590703: 1,
           2.8397325522665495: 2, 4.719507434161063: 3,
           7.498855235336915: 4, 7.976964802216262: 5}
    """
    # Generate the number of events from a Poisson distribution
    num_events = np.random.poisson(rate * time_duration)

    # Generate the event times as cumulative sums from an exponential
       distribution
```

```
    event_times = np.cumsum(np.random.exponential(1 / rate, num_events))

    # Create the N_t dictionary using the zip and dict functions
    N_t = dict(zip(event_times, range(1, num_events + 1)))

    # Return the number of events, event times, and the N_t dictionary
    return num_events, event_times, N_t
```

```
    def proces_ruiny(eta: float, u: float, c: float, T: float, lambda_: float) -> int:
    """
    Simulates the ruin process of capital.

    The ruin process of capital describes whether the capital of an insurance company will be ruined over a given period of time,
    taking into account the inflows (premiums) and outflows (claims).

    Parameters:
        eta (float): Mean value of claims.
            - Must be positive.
        u (float): Initial capital.
            - Must be positive.
        c (float): Premium rate per time unit.
            - Must be positive.
        T (float): Duration of the simulation.
            - Must be positive.
        lambda_ (float): Intensity of the Poisson process (average event frequency).
            - Must be positive.

    Returns:
        int: Returns 1 if the capital is ruined (capital < 0 at any point), otherwise 0.

    Example usage:
        >>> eta = 7
        >>> u = 12
        >>> c = 1
        >>> T = 10
        >>> lambda_ = 0.5
        >>> result = proces_ruiny(eta, u, c, T, lambda_)
        >>> print(result)  # Example result: 1 (ruin occurred)
    """
    # Generate the number of events, event times, and event number dictionary
    num_events, event_times, N_T = poisson_process(lambda_, T)

```

```
    if num_events == 0:
        return 0  # If there are no events, ruin did not occur

    # Generate claim durations and cumulative sums of claim durations
    etas_cumsum = np.cumsum(np.random.exponential(eta, num_events))

    # Calculate the capital state R at different time points
    R = u + c * event_times - etas_cumsum

    # Check if the capital ruin occurred
    return int(np.min(R) < 0)  # Returns 1 if ruin occurred, otherwise 0
```

```
    def pollaczeck_khinchine(u: float, c: float, eta: float, lambd: float) -> float:
    """
    Implementation of the Pollaczek-Khinchine formula to calculate the probability of ruin.

    The Pollaczek-Khinchine formula is used in queueing theory to determine the probability of capital ruin
    for a given initial capital ‘u‘, premium rate ‘c‘, average claim value ‘eta‘, and Poisson process intensity ‘lambd‘.

    Parameters:
        u (float): Initial capital.
            - Must be positive.
        c (float): Premium rate per time unit.
            - Must be positive.
        eta (float): Average claim value.
            - Must be positive.
        lambd (float): Intensity of the Poisson process (average event frequency).
            - Must be positive.

    Returns:
        float: Probability of capital ruin.

    Example usage:
        >>> u = 10
        >>> c = 5
        >>> eta = 2
        >>> lambd = 1
        >>> ruin_probability = pollaczeck_khinchine(u, c, eta, lambd)
        >>> print(ruin_probability)  # Example result: 0.01991482734714558
    """
    # Implementation of the Pollaczek-Khinchine formula
```

```
    return eta * lambd / c * np.exp(-(1 / eta - lambd / c) * u)
```

```
    def ruin_prob_estim(
    eta: float, u: float, c: float, T: float, lambd: float, mc: int
) -> float:
    """
    Estimating the probability of capital ruin using the Monte Carlo
       method.

    The function simulates the capital ruin process multiple times (
       specified number of times) and calculates
    the average probability of ruin based on the simulation results.

    Parameters:
        eta (float): Average claim value.
            - Must be positive.
        u (float): Initial capital.
            - Must be positive.
        c (float): Premium rate per time unit.
            - Must be positive.
        T (float): Duration of the simulation.
            - Must be positive.
        lambd (float): Intensity of the Poisson process (average event
            frequency).
            - Must be positive.
        mc (int): Number of Monte Carlo simulations.
            - Must be positive.

    Returns:
        float: Estimated probability of capital ruin.

    Example usage:
        >>> eta = 7
        >>> u = 12
        >>> c = 1
        >>> T = 10
        >>> lambd = 0.5
        >>> mc = 10000
        >>> ruin_probability = ruin_prob_estim(eta, u, c, T, lambd, mc
           )
        >>> print(ruin_probability)  # Example result: 0.7276
    """
    if_ruin = np.array([proces_ruiny(eta, u, c, T, lambd) for _ in
       range(mc)])
    return np.mean(if_ruin)
```

```
    mc = 10**3  # number of Monte Carlo trials
u = 5  # initial capital
```

```
c = 2  # constant growth rate
lambd = 1  # Poisson process parameter
eta = 1  # parameter of the xi variable
T1 = 5  # example time horizon
T2 = 1000
T3 = 10000
pc = pollaczeck_khinchine(u, c, eta, lambd)

estim1 = ruin_prob_estim(eta, u, c, T1, lambd, mc)
estim2 = ruin_prob_estim(eta, u, c, T2, lambd, mc)
estim3 = ruin_prob_estim(eta, u, c, T3, lambd, mc)

print(f"Estimated probability of ruin for T={T1}: {estim1}")
print(f"Estimated probability of ruin for T={T2}: {estim2}")
print(f"Estimated probability of ruin for T={T3}: {estim3}")
print(f"Probability of ruin from Pollaczek-Khinchine formula: {pc}")

Estimated probability of ruin for T=5: 0.019
Estimated probability of ruin for T=1000: 0.04
Estimated probability of ruin for T=10000: 0.053
Probability of ruin from Pollaczek-Khinchine formula:
   0.0410424993119494
```

6.3 Część druga

Aby przeprowadzić drugą część zadania, sporządzono wykresy funkcji $u \mapsto \psi(u, c_0)$ dla ustalonych c_0 oraz $c \mapsto \psi(u_0, c)$ dla ustalonych u_0 (kilka trajektorii na jednym wykresie) i porównano je z wartościami wyestymowanymi. Wykonano następujące kroki:

- Zdefiniowano zakresy i liczby punktów dla parametrów c_0, c, oraz u_0, u.
- Obliczono teoretyczne wartości prawdopodobieństwa ruiny za pomocą wzoru Pollaczka-Khinchina dla różnych wartości c_0 i u_0.
- Zasymulowano prawdopodobieństwo ruiny kapitału metodą Monte Carlo dla różnych wartości c_0 i u_0 za pomocą funkcji `ruin_prob_estim`.
- Sporządzono wykresy zależności $u \mapsto \psi(u, c_0)$ dla różnych wartości c_0 oraz $c \mapsto \psi(u_0, c)$ dla różnych wartości u_0, porównując wartości teoretyczne z wyestymowanymi.

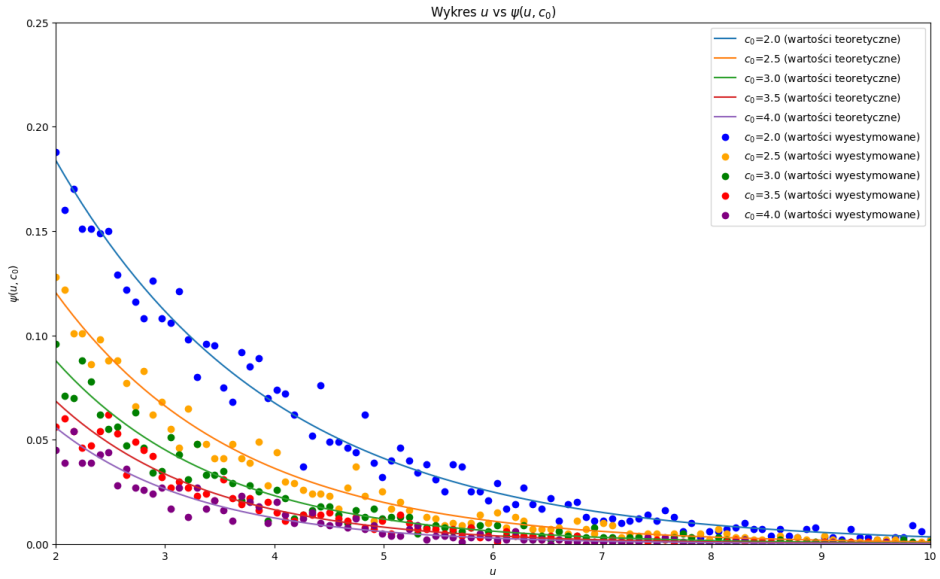
```
    # Ranges and number of points for the parameter c0
c0_min = 2  # Minimum value of the parameter c0
c0_max = 4  # Maximum value of the parameter c0
```

```
dc0 = 5  # Number of equally spaced points between c0_min and c0_max
c0s = np.linspace(c0_min, c0_max, dc0)  # Array of c0 values

# Ranges and number of points for the parameter c
c_min = 1  # Minimum value of the parameter c
c_max = 100  # Maximum value of the parameter c
dc = 100  # Number of equally spaced points between c_min and c_max
cs = np.linspace(c_min, c_max, dc)  # Array of c values

# Ranges and number of points for the parameter u0
u0_min = 1  # Minimum value of the parameter u0
u0_max = 10  # Maximum value of the parameter u0
du0 = 5  # Number of equally spaced points between u0_min and u0_max
u0s = np.linspace(u0_min, u0_max, du0)  # Array of u0 values

# Ranges and number of points for the parameter u
u_min = 2  # Minimum value of the parameter u
u_max = 10  # Maximum value of the parameter u
du = 100  # Number of equally spaced points between u_min and u_max
us = np.linspace(u_min, u_max, du)  # Array of u values
```

```
    # Generate results for each element in c0s
pcs_list = [pollaczeck_khinchine(us, c, eta, lambd) for c in c0s]

# Unpack the results into variables pcs1, pcs2, pcs3, pcs4, pcs5
pcs1, pcs2, pcs3, pcs4, pcs5 = pcs_list
```

```
    # Initialize simulation arrays
sim = np.zeros((10, du))

# Loop to calculate ruin_prob_estim for different values
for i in range(du):
    for j in range(5):
        sim[j, i] = ruin_prob_estim(eta, us[i], c0s[j], 30, lambd, mc)
    for j in range(5, 10):
        sim[j, i] = ruin_prob_estim(eta, u0s[j - 5], cs[i], 30, lambd,
            mc)

# Assign the results to corresponding variables
sim1, sim2, sim3, sim4, sim5, sim6, sim7, sim8, sim9, sim10 = sim

plt.figure(figsize=(15, 9))
plt.plot(
    us,
    list(zip(pcs1, pcs2, pcs3, pcs4, pcs5)),
    label=[
        f"$c_0$={c0s[0]} (theoretical values)",
        f"$c_0$={c0s[1]} (theoretical values)",
```

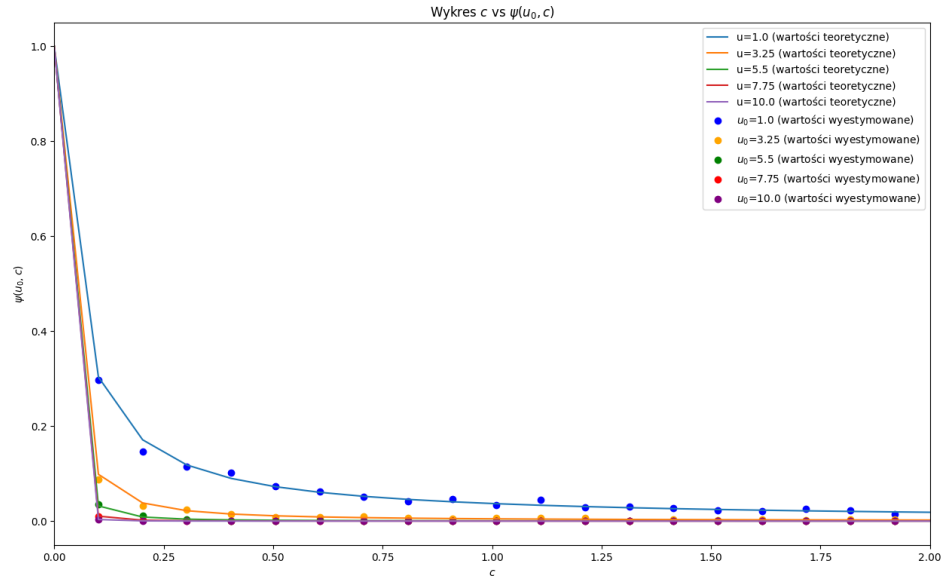
```
        f"$c_0$={c0s[2]} (theoretical values)",
        f"$c_0$={c0s[3]} (theoretical values)",
        f"$c_0$={c0s[4]} (theoretical values)",
    ],
)
plt.scatter(us, sim1, color="blue", label=f"$c_0$={c0s[0]} (estimated values)")
plt.scatter(us, sim2, color="orange", label=f"$c_0$={c0s[1]} (estimated values)")
plt.scatter(us, sim3, color="green", label=f"$c_0$={c0s[2]} (estimated values)")
plt.scatter(us, sim4, color="red", label=f"$c_0$={c0s[3]} (estimated values)")
plt.scatter(us, sim5, color="purple", label=f"$c_0$={c0s[4]} (estimated values)")
plt.legend(loc="best")
plt.title("Plot of $u$ vs $\psi (u, c_0)$")
plt.xlabel("$u$")
plt.ylabel("$\psi (u, c_0)$")
ax = plt.gca()
ax.set_xlim([2, 10])
ax.set_ylim([0, 0.25])
plt.show()
```



Rysunek 13: Wykres u vs $\psi(u, c_0)$

```
    # Generate results for each element in u0s
pcs_list = [pollaczeck_khinchine(u, cs, eta, lambd) for u in u0s]
```

```
3
4  # Unpack the results into variables pcs6, pcs7, pcs8, pcs9, pcs10
5  pcs6, pcs7, pcs8, pcs9, pcs10 = pcs_list
6
7  us2 = np.linspace(0, 10, 100)
8  plt.figure(figsize=(15, 9))
9  plt.plot(
10     us2,
11     list(zip(pcs6, pcs7, pcs8, pcs9, pcs10)),
12     label=[
13         f"u={u0s[0]} (theoretical values)",
14         f"u={u0s[1]} (theoretical values)",
15         f"u={u0s[2]} (theoretical values)",
16         f"u={u0s[3]} (theoretical values)",
17         f"u={u0s[4]} (theoretical values)",
18     ],
19 )
20 plt.scatter(
21     us2[1::], sim6[1::], color="blue", label=f"$u_0$={u0s[0]} (
       estimated values)"
22 )
23 plt.scatter(
24     us2[1::], sim7[1::], color="orange", label=f"$u_0$={u0s[1]} (
       estimated values)"
25 )
26 plt.scatter(
27     us2[1::], sim8[1::], color="green", label=f"$u_0$={u0s[2]} (
       estimated values)"
28 )
29 plt.scatter(
30     us2[1::], sim9[1::], color="red", label=f"$u_0$={u0s[3]} (
       estimated values)"
31 )
32 plt.scatter(
33     us2[1::],
34     sim10[1::],
35     color="purple",
36     label=f"$u_0$={u0s[4]} (estimated values)",
37 )
38 plt.legend(loc="best")
39 plt.title("Plot of $c$ vs $\psi (u_0, c)$")
40 plt.xlabel("$c$")
41 plt.ylabel("$\psi (u_0, c)$")
42 ax = plt.gca()
43 ax.set_xlim([0, 2])
44 plt.show()
```



Rysunek 14: Wykres c vs $\psi(u_0, c)$

6.4 Cześć trzecia

Aby zweryfikować symulacyjnie wyniki dla odwróconego wzoru Pollaczka-Khinchina, skorzystamy z wyznaczonego wzoru:

$$c(u, \psi) = \frac{\lambda u}{W_0\left(\frac{u\psi e^{u/\eta}}{\eta}\right)},$$

gdzie W_0 to gałąź funkcji W Lamberta zdefiniowana poprzez równanie $W_0(xe^x) = x$ dla $x \geqslant 0$. Ta analiza odpowiada szukaniu wymaganej wartości wpłat przy danym kapitale początkowym w celu osiągnięcia wymaganego prawdopodobieństwa ruiny. Wykonano następujące kroki:

- Zaimplementowano odwrócony wzór Pollaczka-Khinchina do obliczenia wartości parametru systemowego `inv_Poll`.
- Obliczono wartości parametru c dla różnych wartości ψ_0.
- Wyestymowano wartości parametru składki c za pomocą funkcji `estimation`.
- Porównano teoretyczne wartości parametru c z wartościami wyestymowanymi i sporządzono odpowiednie wykresy.

```
    def inv_Poll(u: float, psi: float, lambd: float, eta: float) ->
        float:
    """
    Implementation of the inverse Pollaczek-Khinchine formula to
        calculate the system parameter value.

    The inverse Pollaczek-Khinchine formula is used to calculate the
        system parameter `w`,
    which can be utilized for financial risk analysis.

    Parameters:
        u (float): Initial capital.
            - Must be positive.
        psi (float): System parameter value.
            - Must be positive.
        lambd (float): Intensity of the Poisson process (average event
            frequency).
            - Must be positive.
        eta (float): Average claim value.
            - Must be positive.

    Returns:
        float: Calculated system parameter value `w`.

    Example usage:
        >>> u = 10
        >>> psi = 0.5
        >>> lambd = 1
        >>> eta = 2
        >>> w = inv_Poll(u, psi, lambd, eta)
        >>> print(w)  # result: 2.2582133353406615
    """
    w = scipy.special.lambertw(u * psi * np.exp(u / eta) / eta).real
    return lambd * u / w
```

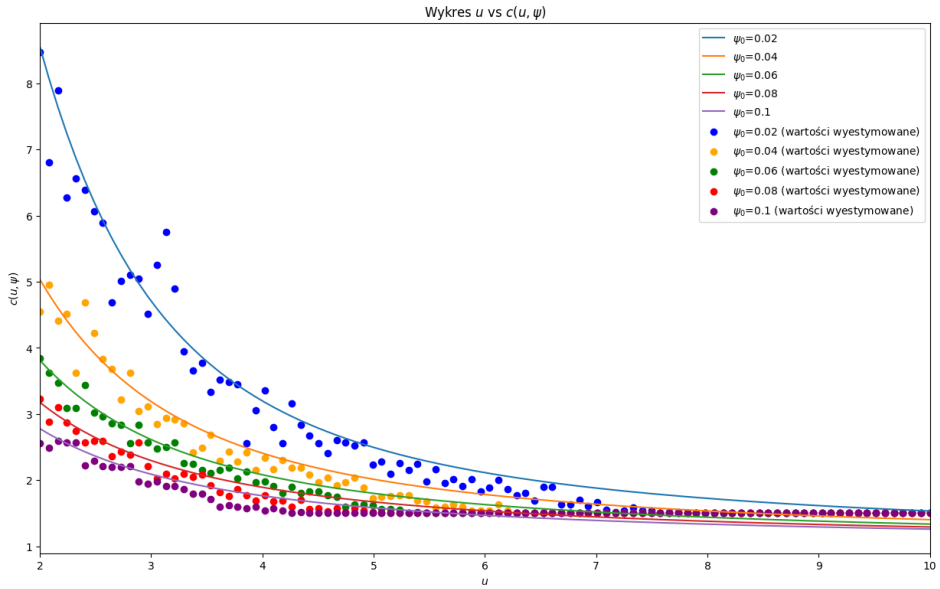
```
    # Ranges and number of points for the parameter psi0
psi0_min = 0.02  # Minimum value of the parameter psi0
psi0_max = 0.1  # Maximum value of the parameter psi0
dpsi0 = 5  # Number of evenly spaced points between psi0_min and
    psi0_max
psi0s = np.linspace(psi0_min, psi0_max, dpsi0)  # Array of psi0 values

# Generating results for each element in psi0s
cs_list = [inv_Poll(us, psi, lambd, eta) for psi in psi0s]

# Unpacking results into variables cs1, cs2, cs3, cs4, cs5
cs1, cs2, cs3, cs4, cs5 = cs_list
```

```
    def estimation(
    T: float,
    n: int,
    u_vector: np.ndarray,
    c_test: np.ndarray,
    psi: float,
    lambd: float = 1,
    eta: float = 1,
) -> np.ndarray:
    """
    Estimation of the premium parameter ‘c‘ using the bisection method
        based on the given ruin probability ‘psi‘.

    The function iteratively estimates the premium parameter ‘c‘ for
       various initial capitals ‘u‘ using
    the bisection method to achieve the target ruin probability ‘psi‘
       in a Monte Carlo simulation.

    Parameters:
        T (float): Simulation duration.
            - Must be positive.
        n (int): Number of Monte Carlo simulations for each case.
            - Must be positive.
        u_vector (np.ndarray): Array containing different initial
           capitals ‘u‘ for which the premium ‘c‘ is estimated.
        c_test (np.ndarray): Array containing possible values of the
           premium parameter ‘c‘ to test using the bisection method.
        psi (float): Target ruin probability.
            - Must be positive.
        lambd (float, optional): Intensity of the Poisson process (
           mean event frequency). Default is 1.
            - Must be positive.
        eta (float, optional): Average claim value. Default is 1.
            - Must be positive.

    Returns:
        np.ndarray: Array containing the estimated premium values ‘c‘
           for each initial capital ‘u‘ from ‘u_vector‘.

    Example usage:
        >>> T = 10
        >>> n = 10000
        >>> u_vector = np.array([10, 20, 30])
        >>> c_test = np.linspace(0.1, 2.0, 100)
        >>> psi = 0.1
        >>> estimated_c_values = estimation(T, n, u_vector, c_test,
           psi)
```

```
        >>> print(estimated_c_values)  # result: [0.53417969
            0.10371094 0.10371094]
    """
    cs = np.zeros(len(u_vector))  # Initialize array for estimated
        premium values ‘c‘

    # Iterate over different initial capitals ‘u‘
    for i, u in enumerate(u_vector):
        c_low, c_high = c_test[0], c_test[-1]  # Initial lower and
            upper bounds for premium ‘c‘

        # Continue loop while the difference between upper and lower
            bounds is greater than 0.01
        while c_high - c_low > 0.01:
            c_mid = (c_low + c_high) / 2  # Midpoint value of premium
                ‘c‘
            psi_test = ruin_prob_estim(eta, u, c_mid, T, lambd, n)  #
                Estimate ruin probability for premium ‘c_mid‘

            # Decrease lower bound if the estimated psi is greater
                than the target ‘psi‘
            if psi_test > psi:
                c_low = c_mid
            else:
                c_high = c_mid  # Otherwise, increase upper bound

        cs[i] = (c_low + c_high) / 2  # Set the midpoint value as the
            estimated premium ‘c‘ for capital ‘u‘

    return cs  # Return array of estimated premium values ‘c‘ for each
        ‘u‘
```

```
    # Parameter definitions
c_test = np.linspace(1.5, 10, 100)
T = 10

# Estimating the value of parameter c for different values of psi0s
c_empiric_list = [estimation(T, 1000, us, c_test, psi) for psi in
    psi0s]

# Unpacking results into variables c_1_empiric, c_2_empiric,
    c_3_empiric, c_4_empiric, c_5_empiric
c_1_empiric, c_2_empiric, c_3_empiric, c_4_empiric, c_5_empiric =
    c_empiric_list

plt.figure(figsize=(15, 9))
plt.plot(
    us,
```

```
    list(zip(cs1, cs2, cs3, cs4, cs5)),
    label=[
        f"$\psi_0$={psi0s[0]}",
        f"$\psi_0$={psi0s[1]}",
        f"$\psi_0$={psi0s[2]}",
        f"$\psi_0$={psi0s[3]}",
        f"$\psi_0$={psi0s[4]}",
    ],
)
plt.scatter(
    us, c_1_empiric, color="blue", label=f"$\psi_0$={psi0s[0]} (estimated values)"
)
plt.scatter(
    us,
    c_2_empiric,
    color="orange",
    label=f"$\psi_0$={psi0s[1]} (estimated values)",
)
plt.scatter(
    us, c_3_empiric, color="green", label=f"$\psi_0$={psi0s[2]} (estimated values)"
)
plt.scatter(
    us, c_4_empiric, color="red", label=f"$\psi_0$={psi0s[3]} (estimated values)"
)
plt.scatter(
    us,
    c_5_empiric,
    color="purple",
    label=f"$\psi_0$={psi0s[4]} (estimated values)",
)
plt.legend(loc="best")
plt.title("Plot $u$ vs $c(u, \psi)$")
plt.xlabel("$u$")
plt.ylabel("$c(u, \psi)$")
ax = plt.gca()
ax.set_xlim([2, 10])
plt.show()
```



Rysunek 15: Wykres u vs $c(u, \psi)$

6.5 Wnioski

W zadaniu piątym analizowaliśmy proces ruiny kapitału w modelu Cramera-Lundberga oraz estymowaliśmy wartości składki potrzebnej do osiągnięcia określonego prawdopodobieństwa ruiny. Poniżej przedstawiono główne wnioski z poszczególnych części zadania:

6.5.1 Część pierwsza

W części pierwszej przeanalizowaliśmy prawdopodobieństwo ruiny w czasie skończonym T oraz porównaliśmy wyniki symulacyjne z teoretycznymi wartościami uzyskanymi z wzoru Pollaczka-Chinczyna:

- Wygenerowano proces Poissona za pomocą funkcji `poisson_process`.
- Zasymulowano proces ruiny kapitału za pomocą funkcji `proces_ruiny`.
- Obliczono teoretyczne prawdopodobieństwo ruiny za pomocą wzoru Pollaczka-Khinchina.
- Estymowano prawdopodobieństwo ruiny metodą Monte Carlo za pomocą funkcji `ruin_prob_estim`.

- Porównano wyniki symulacji z wartościami teoretycznymi, uzyskując zgodność wyników, co potwierdziło poprawność zaimplementowanych metod.

6.5.2 Część druga

W części drugiej sporządzono wykresy funkcji $u \mapsto \psi(u, c_0)$ oraz $c \mapsto \psi(u_0, c)$ i porównano wartości teoretyczne z wyestymowanymi:

- Zdefiniowano zakresy i liczby punktów dla parametrów c_0, c, u_0 i u.
- Obliczono teoretyczne wartości prawdopodobieństwa ruiny za pomocą wzoru Pollaczka-Khinchina dla różnych wartości c_0 i u_0.
- Zasymulowano prawdopodobieństwo ruiny kapitału metodą Monte Carlo dla różnych wartości c_0 i u_0 za pomocą funkcji `ruin_prob_estim`.
- Sporządzono wykresy zależności $u \mapsto \psi(u, c_0)$ oraz $c \mapsto \psi(u_0, c)$, porównując wartości teoretyczne z wyestymowanymi.
- Wyniki wykazały zgodność pomiędzy wartościami teoretycznymi a wyestymowanymi, co potwierdziło poprawność zastosowanych metod symulacyjnych.

6.5.3 Część trzecia

W części trzeciej zweryfikowano wyniki dla odwróconego wzoru Pollaczka-Khinchina:

- Zaimplementowano odwrócony wzór Pollaczka-Khinchina do obliczenia wartości parametru systemowego za pomocą funkcji `inv_Poll`.
- Obliczono wartości parametru składki c dla różnych wartości ψ_0.
- Wyestymowano wartości parametru składki c za pomocą funkcji `estimation`.
- Porównano teoretyczne wartości parametru c z wartościami wyestymowanymi i sporządzono odpowiednie wykresy.
- Wyniki pokazały zgodność pomiędzy wartościami teoretycznymi a wyestymowanymi, co potwierdziło poprawność zastosowanych metod oraz modeli.

Podsumowując, przeprowadzone analizy i symulacje potwierdziły teoretyczne modele dotyczące procesu ruiny kapitału oraz estymacji wartości składki potrzebnej do osiągnięcia określonego prawdopodobieństwa ruiny. Wyniki symulacji były zgodne z przewidywaniami teoretycznymi, co świadczy o poprawności zastosowanych metod oraz modeli. Zadanie to pozwoliło na praktyczne zweryfikowanie teoretycznych koncepcji za pomocą symulacji komputerowych.

7 Zad 6.

W tym zadaniu skupimy się na symulacji procesu Wienera.

7.1 Wstęp.

Prawa Arcusa Sinusa, oraz Ruch Browna(Proces Wienera).

Proces Wienera, znany również jako ruchy Browna, jest fundamentalnym modelem matematycznym w teorii probabilistycznej i analizie stochastycznej. Jego nazwa pochodzi od amerykańskiego matematyka Norberta Wienera. Proces Wienera jest używany do modelowania przypadkowych ruchów w przestrzeni i ma zastosowanie w różnych dziedzinach nauki, w tym w fizyce, finansach, biologii i inżynierii.

Wyniki uzyskane podczas symulacji zaprezentowane są na wykresach 4-6.

7.2 Proces Wienera.

Matematycznie proces Wienera $\{W(t), t \geqslant 0\}$ spełnia następujące warunki:

$W(0) = 0$,
W(t) - W(s) $\sim$ N(0,t - s) dla $0 \leqslant s \geqslant t$,
Proces $W(t)$ ma ciągłe trajektorie.

Rozkład arcsinusowy na przzedziale $[0, 1]$ ma funkcję gęstości daną wzorem:

$$p_X(x) = \frac{1}{x\sqrt{x(1-x)}} 1(x),$$

dla $x \in (0, 1)$. Jest on charakterystyczny tym, że jego wartości są skoncentrowane przy

krańcach przedziału, czyli wokół 0 i 1.

Proces Wienera posiada własność, że różnice jego wartości w dwóch różnych momentach czasu są z rozkładu normalnego (Gaussa) ze średnią równą zero i wariancją równą różnicy czasu między tymi punktami. To pozwala na stosowanie narzędzi analizy statystycznej do badania takich procesów.

7.3 Opis Metod.

Pierwsze prawo arcsinusów:

Twierdzenie:

$$T_+ = \lambda(\{t \in [0,1] \mid W_t > 0\}) \sim \text{arcsin},$$

gdzie λ to miara Lebesgue'a. Oznacza to, że czas, który proces Wienera spędza powyżej osi OX na przedziale [0,1], ma rozkład arcsinusowy.

Jeśli W_t to standardowy proces Wienera na przedziale $[0,1]$, to czas, który proces spędza powyżej osi OX (czyli czas, w którym $W_t > 0$), ma rozkład arcsinusowy. Rozkład arcsinusowy jest skoncentrowany przy krańcach przedziału, co oznacza, że proces Wienera spędza większość czasu albo blisko początku, albo blisko końca przedziału powyżej osi OX, rzadziej zaś w jego środkowej części.

Drugie prawo arcsinusów:

Twierdzenie:

$$L = sup\{t \in [0,1] \mid W_t = 0\} \sim \text{arcsin}\,.$$

Inaczej mówiąc, ostatni moment uderzenia procesu Wienera na odcinku [0, 1] w oś OX ma rozkład arcusa sinusa.

W kontekście procesu Wienera, L jest ostatnim czasem, w którym proces W_t osiąga wartość zero na przedziale [0,1]. To twierdzenie mówi, że ten ostatni czas ma rozkład arcsinusowy. Znów, ze względu na własności rozkładu arcsinusowego, ten czas jest bardziej prawdopodobny bliżej krańców przedziału niż w jego środku.

Trzecie prawo arcsinusów:

Twierdzenie:

Niech M będzie liczbą spełniającą:

$$W_M = sup\{W_t \mid \in [0,1]\},$$

$$T_M = sup\{t \in [0,1] \mid W_t = max_{s\in[0,1]} W_s\} \sim \arcsin .$$

Wtedy M $\sim$ arcsin. Oznacza to, że moment osiągnięcia maksymalnej wartości przez proces Wienera na odcinku [0, 1] ma rozkład arcusa sinusa.

W kontekście procesu Wienera, T_M jest czasem, w którym proces osiąga swoją maksymalną wartość na przedziale [0,1]. To twierdzenie mówi, że ten czas ma rozkład arcsinusowy, co oznacza, że proces Wienera ma większą tendencję do osiągania swojej maksymalnej wartości bliżej początku lub końca przedziału, niż w jego środku.

7.4 Sktypty w Pythonie.

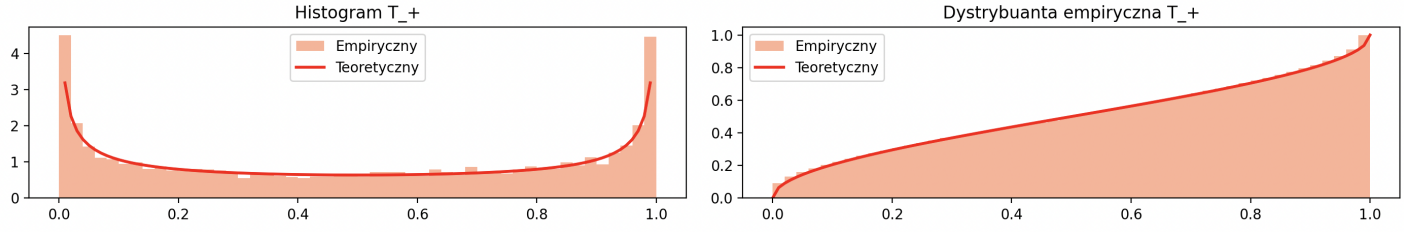
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import arcsine

# Liczba symulacji
n_simulations = 10000
n_steps = 1000
t = np.linspace(0, 1, n_steps + 1)

# Funkcja generujaca proces Wienera
def generate_wiener_process(n_steps: int) -> np.ndarray:
    """
    Generuje Proces Wienera.

    Args:
        n_steps (int): Liczba krokow w procesie.

    Returns:
        np.ndarray: Wysymulowany proces Wienera
    """
    dt = 1 / n_steps
```

```
    dW = np.random.normal(0, np.sqrt(dt), n_steps)
    W = np.concatenate(([0], np.cumsum(dW)))
    return W

# Listy wynikowe
T_plus = []
L = []
M = []

# Generowanie wszystkoch jednoczesnie, ale kazda wersja odpowiednio wedlug swojej definicji
# Symulacja procesu Wienera oraz zbieranie danych

for _ in range(n_simulations):
    # Generowanie procesu Wienera
    W = generate_wiener_process(n_steps)

    # Obliczanie T_+
    T_plus.append(np.sum(W > 0) / n_steps)

    # Znajdowanie punktow przeciecia z zerem
    zero_crossings = np.where(np.diff(np.sign(W)))[0]

    # Sprawdzenie czy istnieja punkty przeciecia z zerem
    if zero_crossings.size > 0:
        # Jesli istnieja, dodaj ostatni czas przeciecia z zerem przed zakonczeniem symulacji do listy L
        L.append(t[zero_crossings[-1]])
    else:
        # Jesli nie istnieja, dodaj 0 do listy L
        L.append(0)

    # Dodawanie czasu osiagniecia maksimum do listy M
    M.append(t[np.argmax(W)])

# Histogramy oraz dystrybuanty empiryczne i teoretyczne
fig, axs = plt.subplots(3, 2, figsize=(14, 7))

# Histogram dla T_+
axs[0, 0].hist(
    T_plus, bins=50, density=True, alpha=0.6, color="coral", label="Empiryczny"
)
x = np.linspace(0, 1, 100)
axs[0, 0].plot(x, arcsine.pdf(x), "r-", lw=2, label="Teoretyczny")
axs[0, 0].set_title("Histogram T_+")
axs[0, 0].legend()

```

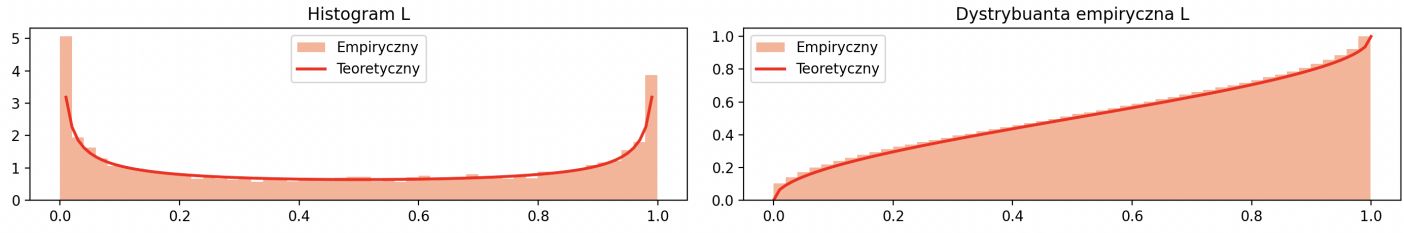
```
# Dystrybuanty dla T_+
axs[0, 1].hist(
    T_plus,
    bins=50,
    density=True,
    cumulative=True,
    alpha=0.6,
    color="coral",
    label="Empiryczny",
)
axs[0, 1].plot(x, arcsine.cdf(x), "r-", lw=2, label="Teoretyczny")
axs[0, 1].set_title("Dystrybuanta empiryczna T_+")
axs[0, 1].legend()

# Histogram dla L
axs[1, 0].hist(L, bins=50, density=True, alpha=0.6, color="coral",
    label="Empiryczny")
axs[1, 0].plot(x, arcsine.pdf(x), "r-", lw=2, label="Teoretyczny")
axs[1, 0].set_title("Histogram L")
axs[1, 0].legend()

# Dystrybuanty dla L
axs[1, 1].hist(
    L,
    bins=50,
    density=True,
    cumulative=True,
    alpha=0.6,
    color="coral",
    label="Empiryczny",
)
axs[1, 1].plot(x, arcsine.cdf(x), "r-", lw=2, label="Teoretyczny")
axs[1, 1].set_title("Dystrybuanta empiryczna L")
axs[1, 1].legend()

# Histogram dla M
axs[2, 0].hist(M, bins=50, density=True, alpha=0.6, color="coral",
    label="Empiryczny")
axs[2, 0].plot(x, arcsine.pdf(x), "r-", lw=2, label="Teoretyczny")
axs[2, 0].set_title("Histogram M")
axs[2, 0].legend()

# Dystrybuanty dla M
axs[2, 1].hist(
    M,
    bins=50,
    density=True,
    cumulative=True,
```

```
    alpha=0.6,
    color="coral",
    label="Empiryczny",
)
axs[2, 1].plot(x, arcsine.cdf(x), "r-", lw=2, label="Teoretyczny")
axs[2, 1].set_title("Dystrybuanta empiryczna M")
axs[2, 1].legend()

plt.tight_layout()
plt.show()
```

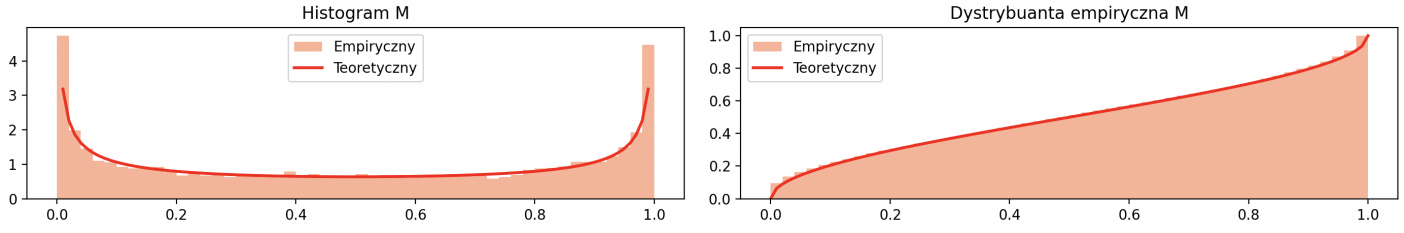
Listing 11: Skrypt Zad 6.



Rysunek 16: Porównanie Wykresów Procesu T_+



Rysunek 17: Porównanie dla Procesu L.



Rysunek 18: Porównanie dla Procesu M.

Na wykresach widzimy, jak teoria mówiąca nam o rozłożeniu danych łączy się z empirycznymi wynikami symulacji.

7.5 Wnioski.

Prawa arcsinusów dostarczają głębokich intuicji na temat rozkładu czasu związanego z pewnymi kluczowymi zdarzeniami w procesie Wienera. Te prawa są szczególnie użyteczne w analizie właściwości procesów stochastycznych i są ważnymi narzędziami w probabilistyce oraz analizie finansowej, gdzie procesy Wienera często modelują ceny lub inne zjawiska losowe.

W finansach proces Wienera jest fundamentem modelowania dynamiki cen aktywów, np. w modelu Blacka-Scholesa do wyceny opcji. Jego własności, takie jak brak dryfu i skalowanie wariancji, czynią go użytecznym do modelowania nieprzewidywalnych zmian rynkowych, ale to temat do rozważenie na innym raporcie.