raport

June 20, 2024

0.0.1 Przedmiot: Symulacje komputerowe

0.0.2 Prowadzący: dr inż. Rafał Połoczański

0.0.3 Autorzy: Bartłomiej Mielcarz 264347, Maria Nowacka 275981 , Roksana Rymarek 276056

1 zadanie 1

[2]:
```
import numpy as np, matplotlib.pyplot as plt, time
```

1.1 Generator ACORN (Additive Congruential Random Number)

Generator ACORN k-tego rzędu oparty jest na ciągach liczbowych zdefiniowanych poprzez równania rekurencyjne

$$X_n^0 = X_{n-1}^0,\ n \geq 1 X_n^m = (X_n^{m-1} + X_{n-1}^m) \bmod M, m = 1, \ldots, k,\ n \geq 1 Y_n^k = \frac{X_n^k}{M},\ n \geq 1.$$

Jeśli ziarno (pierwszy element wektora o elementach $X_0^m \in 0...M$) , czyli $X_0^0 \in 1...M-1$, jest względnie pierwsze z M oraz M jest dostatecznie dużą liczbą naturalną, to generator posiada pożądane cechy a ciąg Y_n^k dla $n \geq 1$ przypomina ciąg IID zmiennych losowych z rozkładu jednostajnego $U(0,1)$. Zaimplementuj generator jako funkcję `ACORN(N, k, M, Lag)` zwracającą `N` liczb pseudolosowych Y_n^k, dla $n = 1, ..., N$, korzystając z algorytmu k-tego rzędu, przy ominięciu pierwszych `Lag` wyrazów. Przykładowymi parametrami mogą być $\texttt{M} = 2^{89} - 1$ (liczba pierwsza Mersenne'a), $\texttt{k} = 9$, $X_0^m = 0$ dla $m \neq 0$ (mogą być dowolne), $\texttt{Lag} = 10^3$ (czy to na pewno OK?). Sprawdź poprawność wyniku robiąc wykresy ciągu Y_n^k, funkcji $Y_{n-1}^k \mapsto Y_n^k$ oraz histogramy. Porównaj wydajność z wbudowanym generatorem numpy. Szczegóły oraz odniesienia do literatury znaleźć możemy na stronie autorów algorytmu https://acorn.wikramaratna.org/index.html.

1.2 Implementacja

[ ]:
```
def ACORN(N, k=9, M = 2**89 - 1, Lag=10**3):
  c = 110011
  X = [[0]*(N+1+Lag) for _ in range(k+1)]
  X[0][0] = c
  n = len(X) # wiersze
  m = len(X[0]) # kolumny
  for i in range(n):
    for j in range(1,m):
```

```
        X[i][j] = (X[i-1][j] + X[i][j-1])%M
    Yk = [X[k][j]/M for j in range(m-1)]
    return Yk[Lag::]
```

Jako `Lag` początkowo wstawiliśmy 1000. Patrząc na późniejsze wykresy zobaczymy, że bez pominięcia tych pierwszych `Lag` wartości, próbka nie zachowuje się jak rozkład jednostajny.

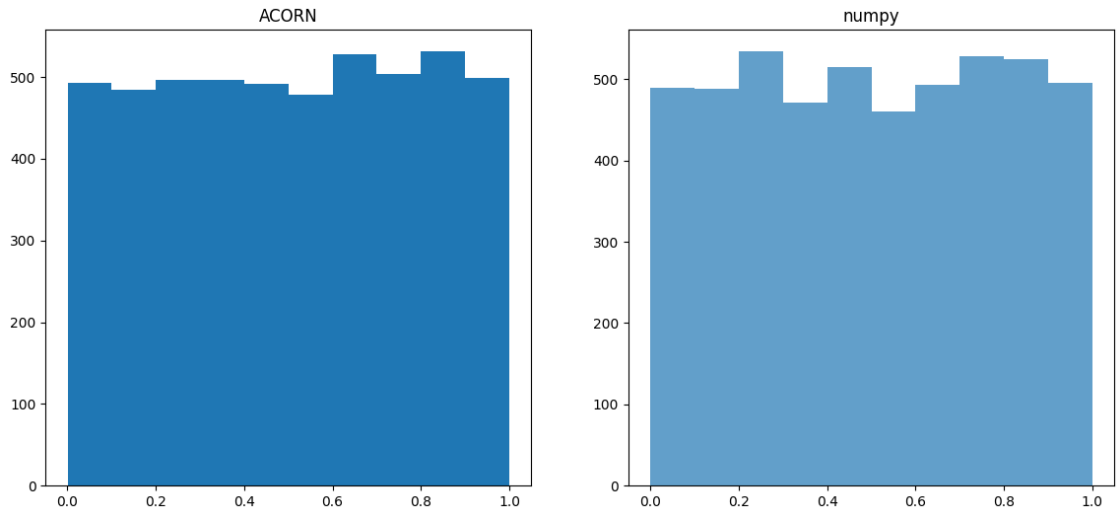
Wywołanie funkcji - próbka o długości 5000

[ ]:
```
size = 5000
a=ACORN(size)
```

1.3 Porównanie histogramów: metoda `ACORN` i wbudowana funkcja z `numpy`.

[ ]:
```
n = np.random.rand(size)
plt.figure(figsize=(14, 6))
plt.subplot(1,2,1)
plt.hist(a)
plt.title('ACORN')
plt.subplot(1,2,2)
plt.hist(n, alpha = 0.7)
plt.title('numpy')
```

[ ]:
```
Text(0.5, 1.0, 'numpy')
```



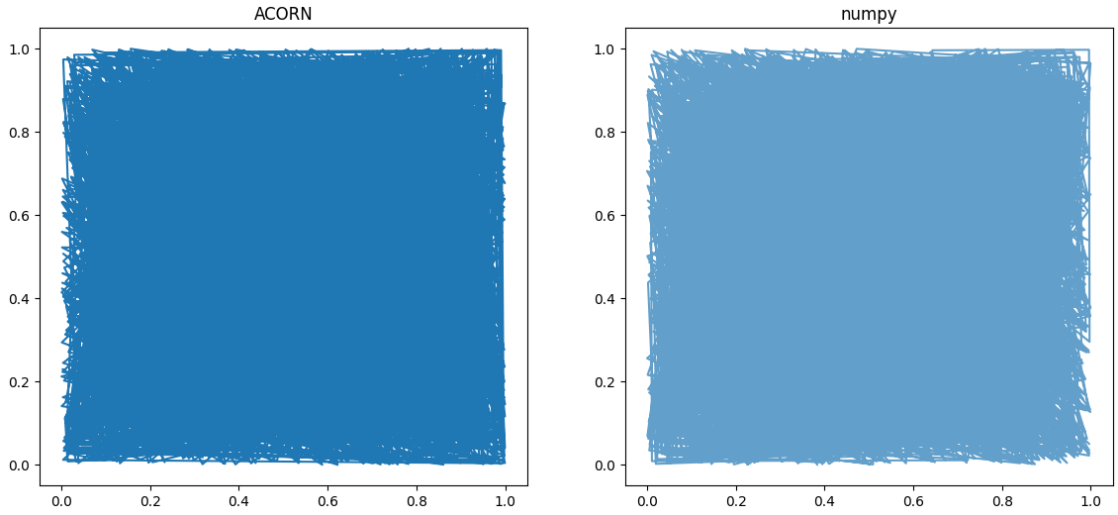
Jak widać histogramy są do siebie podobne i pokazują rozkład próbek - obie wyglądają jak rozkład jednostajny.

1.4 Wykres funkcji

Poniżej widzimy wykres przedstawiający funkcję $Y_{n-1}^k \mapsto Y_n^k$. Jak widać, dla wbudowanego generatora wykres tej samej funkcji przedstawia się podobnie. Wykres przedstawia połączenie następnego wyrazu ciągu z poprzednim - jak widzimy, kolejne wyrazy ciągu są od siebie niezależne, więc algorytm działa pod tym względem poprawnie.

[ ]:
```
plt.figure(figsize=(14, 6))
plt.subplot(1,2,1)
plt.plot(a[0:-1], a[1::])
plt.title('ACORN')
plt.subplot(1,2,2)
plt.plot(n[0:-1], n[1::], alpha = 0.7)
plt.title('numpy')
```

[ ]:
```
Text(0.5, 1.0, 'numpy')
```

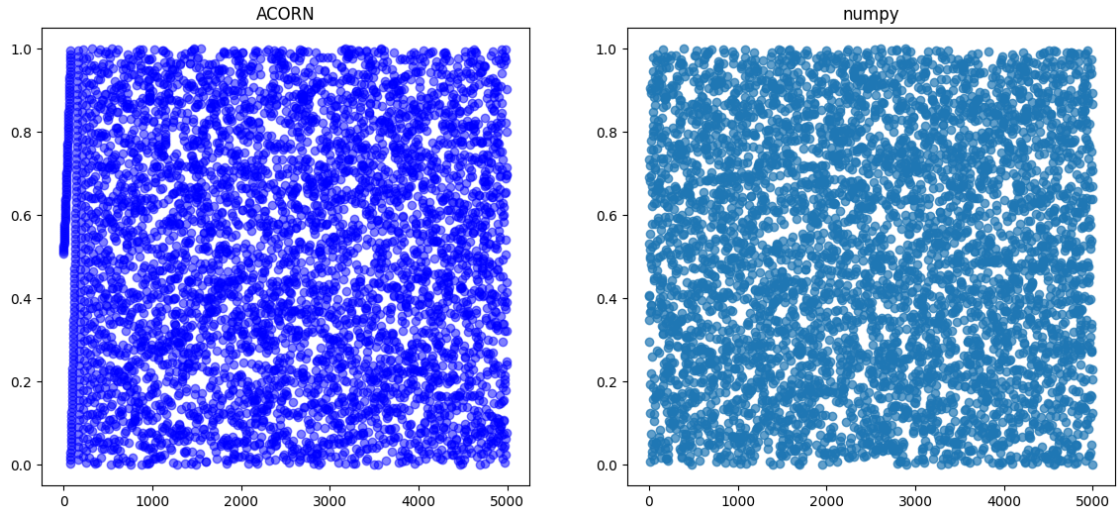


1.5 Wykres ciągu

Przedstawienie ciągu Y_n^k na wykresie pokazuje nam, jak ważne jest pominięcie pierwszych `Lag` wyrazów ciągu. Początkowo ciąg jest wyraźnie rosnący (przedziałami), a losowość zaczyna się po kilkuset wyrazach.

[ ]:
```
plt.figure(figsize=(14, 6))
plt.subplot(1,2,1)
plt.scatter(range(len(a)), a, alpha = 0.5, color='blue')
plt.title('ACORN')
plt.subplot(1,2,2)
plt.scatter(range(len(n)), n, alpha = 0.7)
plt.title('numpy')
```

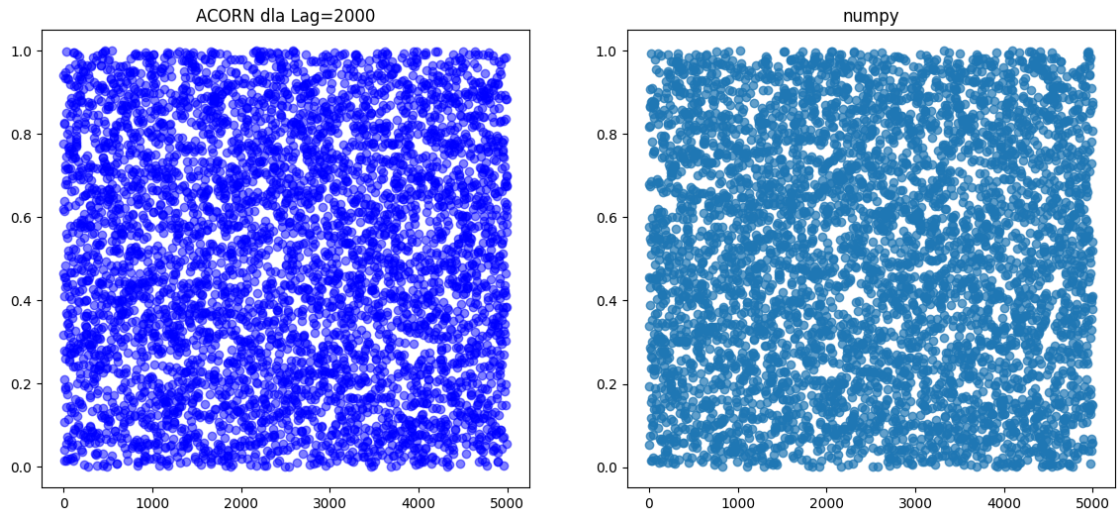
[ ]: Text(0.5, 1.0, 'numpy')



Po wprowadzeniu poprawki i zwięszeniu Lag do 2000 wyrazów widzimy, że generator działa lepiej - ciąg jest rzeczywiście losowy (pseudolosowy).

[ ]:
```
plt.figure(figsize=(14, 6))
plt.subplot(1,2,1)
plt.scatter(range(size), ACORN(size, Lag = 2*10**3), alpha = 0.5, color='blue')
plt.title('ACORN dla Lag=2000')
plt.subplot(1,2,2)
plt.scatter(range(size), np.random.rand(size), alpha = 0.7)
plt.title('numpy')
```

[ ]: Text(0.5, 1.0, 'numpy')



Dla `Lag` = 2000 pozostałe wykresy prezentują się tak samo jak przy `Lag` = 1000 (ewentualne różnice są nieznaczące).

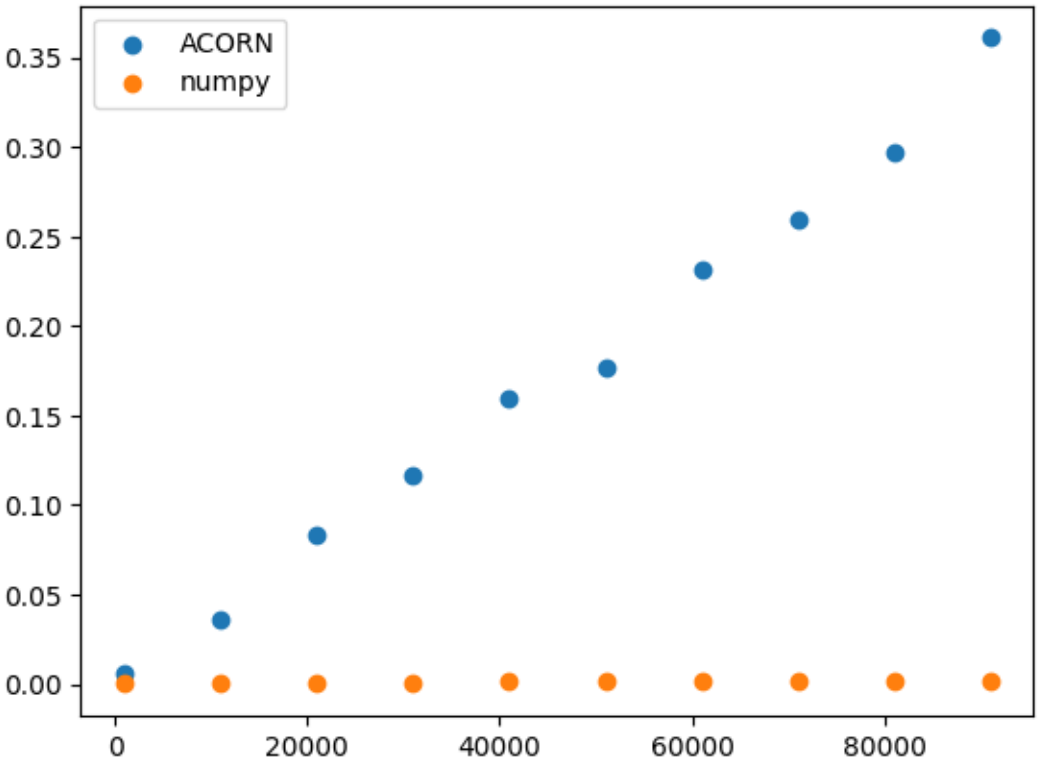
1.6 Wydajność

Porównajmy jeszcze czas generowania próbek dla różnych wartości parametru `N`.

[ ]:
```
timesa=[]
timesn=[]
for i in range(10**3, 10**5, 10**4):
  ta, tn = 0, 0
  l = 50
  for _ in range(l):
    t1 = time.time()
    ACORN(i)
    ta+=time.time()-t1
    t2 = time.time()
    np.random.rand(i)
    tn += time.time()-t2
  timesa.append(ta/l)
  timesn.append(tn/l)
```

[ ]:
```
plt.scatter(list(range(10**3, 10**5, 10**4)), timesa, label="ACORN")
plt.scatter(list(range(10**3, 10**5, 10**4)), timesn, label="numpy")
plt.legend()
```

[ ]:
```
<matplotlib.legend.Legend at 0x7a0448d32590>
```



1.7 Wnioski

Jak widać na wykresach, generator działa prawidłowo - wygenerowana próbka, po odrzuceniu pierwszych `Lag` wartości ma rozkład jednostajny na odcinku [0,1]. Wadą algorytmu jest jego wydajność - dla badanych ziaren `ACORN` działa wolniej od wbudowanego generatora `numpy`. Ziarno deklarowane w funkcji sprawia, że wyniki symulacji są powtarzalne, co może być pomocne w różnego rodzaju testach. Według naszych testów wybór ziarna nie wpływa znacznie na wydajność kodu.

2 zadanie 2

Na listach zadań poznaliśmy kilka metod generowania rozkładu normalnego - nie są to jedyne metody znane i powszech- nie używane metody. W pierwszym kroku sprawdź dotychczas zaimplementowane algorytmy, czy działają poprawnie. W drugim kroku, zaimplementuj dwie dodatkowe metody generowania rozkładu normalnego:

- metodę ”tuzina przykładowa implementacja w linku poniżej
- metodę zigguratu

https://en.wikipedia.org/wiki/Ziggurat_algorithm używając rozkładu wykładniczego. Przetestuj ile razy musimy obliczać funkcję wykładniczą dla różnych wyborów schodków. Finalnie, porównaj efektywność generowania rozkładu normalnego za pomocą samodzielnie zaimplementowanych metod. Rozważ zarówno czas wykonania algorytmu, jak i dokładność symulacji.

[ ]:
```
import numpy as np
import matplotlib.pyplot
import time
import math
from scipy.special import erfinv
```

2.1 Metoda Boxa-Mullera

Algorytm

1. Wygeneruj zmienną losową $U1$ z rozkładu jednostajnego na przedziale $(0, 1)$.
2. Wygeneruj zmienną losową $U2$ z rozkładu jednostajnego na przedziale $(0, 1)$.
3. Oblicz $R = -2ln\ (U1)$.
4. Oblicz $\Theta = 2\pi U2$
5. Oblicz $Z0 = Rcos\ (\Theta)$.
6. Oblicz $Z1 = Rsin\ (\Theta)$.
7. Zwróć wartości $Z0$ i $Z1$

[ ]:
```
#Implementacja Boxa-Mullera
def box_muller(n):
    U1 = np.random.uniform(0, 1, n//2)
    U2 = np.random.uniform(0, 1, n//2)
    R = np.sqrt(-2 * np.log(U1))
    Theta = 2 * np.pi * U2
    X = R * np.cos(Theta)
    Y = R * np.sin(Theta)
    return np.concatenate((X, Y))
```

2.2 Metoda Transformacji Odwrotnej Dystrybuanty

Algorytm :

1. Wygeneruj zmienną losową U z rozkładu jednostajnego na przedziale $(0, 1)$.
2. Oblicz wartość X za pomocą wzoru $X = 2 \cdot erfinv(2U - 1)$.
3. Powtórz kroki 1-2 nn razy, aby wygenerować nn próbek.
4. Zwróć listę wygenerowanych próbek.

[ ]:
```
# Implementacja transformacji odwrotnej dystrybuanty
def inverse_transform(n):
    samples = []
    for _ in range(n):
        U = np.random.uniform(0, 1)
        sample = np.sqrt(2) * erfinv(2 * U - 1)
        samples.append(sample)
```

```
    return samples
```

2.3 Metoda "Tuzina"

Algorytm

1. Zainicjuj listę próbek. Dla każdego z n próbek:
2. Wygeneruj 12 niezależnych zmiennych losowych Ui z rozkładu jednostajnego na przedziale $(0, 1)$.
3. Oblicz sumę tych 12 zmiennych S.
4. Oblicz X jako $S - 6$.
5. Oblicz Y jako $sX + m$.
6. Dodaj próbkę YY do listy próbek.'
7. Zwróć listę próbek.

[ ]:
```
#Implemencja metody ,,TUZINA,,
def dozen_method(n, m=0, s=1):
    samples = []
    for _ in range(n):
        U = np.random.uniform(0, 1, 12)
        S = np.sum(U)
        X = S - 6
        Y = s * X + m
        samples.append(Y)
    return samples
```

2.4 Metoda Zigguratu

Algorytm:

Ustawienia wstępne:

1. $n = 255$

$\nu = 0.00492867323399$

$r = 3.6541528853610088$

2. $x_{255} = r$
3. Generowanie wartości x_i dla $i = 254, 253, ..., 1$

$x_i = \nu/(x_i + 1) + x_{i+1}$

4. Generowanie próbki:

Wybierz $i = 1 + int(nU1)$

5. $x = x_iU2$

6. Jeżeli $\mid x \mid < x_{i-1}$, to $X = x$ i przejdź do punktu 10.

7. Jeżeli $i \neq n$:

$y = (\phi(x_{i-1}) - \phi(xi))U3$ 8. Jeżeli $y < (\phi(x) - \phi(xi))$, to $X = x$ i przejdź do punktu 10.

9. Jeżeli $\mid x \mid \geq x_{i-1}$:

$d = -1 + 2U4 \quad f = U5 \quad x = -ln\ (\mid d \mid)/3$

$y = -ln\ (f)$

Jeżeli $2y \leq x2$, to przejdź do punktu 9.1.

Jeżeli $d > 0$, to $X = x + 3$. W przeciwnym razie $X = -x - 3$.

10. Transformacja do $Y = sX + m$

[ ]:
```
import numpy as np
from scipy import integrate
import time
import matplotlib.pyplot as plt


f_normalny = lambda x: np.exp(-(x**2) / 2)
f_normalny_odwrotna = lambda y: np.sqrt(-2 * np.log(y))

#Inicjalizacja zmiennych:
n=256
x_1 = 3.6541528853610088
y_1 = f_normalny(x_1)


#Tablica przechowująca współrzędne brzegowe prostokątów (x i y)
tablica = np.zeros(shape=(n, 2))
tablica[0, 0] = x_1
tablica[0, 1] = y_1


pole_ogona = integrate.quad(f_normalny, x_1, np.inf)[0]
pole_0 = x_1 * y_1 + pole_ogona

#Generowanie warstw
for i in range(1, n - 1):
    tablica[i, 1] = tablica[i - 1, 1] + pole_0 / tablica[i - 1, 0]
    tablica[i, 0] = f_normalny_odwrotna(tablica[i, 1])
tablica[n - 1, 0] = 0
tablica[n - 1, 1] = 1

def ziggurat_method(n_samples, mu=0, sigma=1):
    samples = np.zeros(n_samples)
```

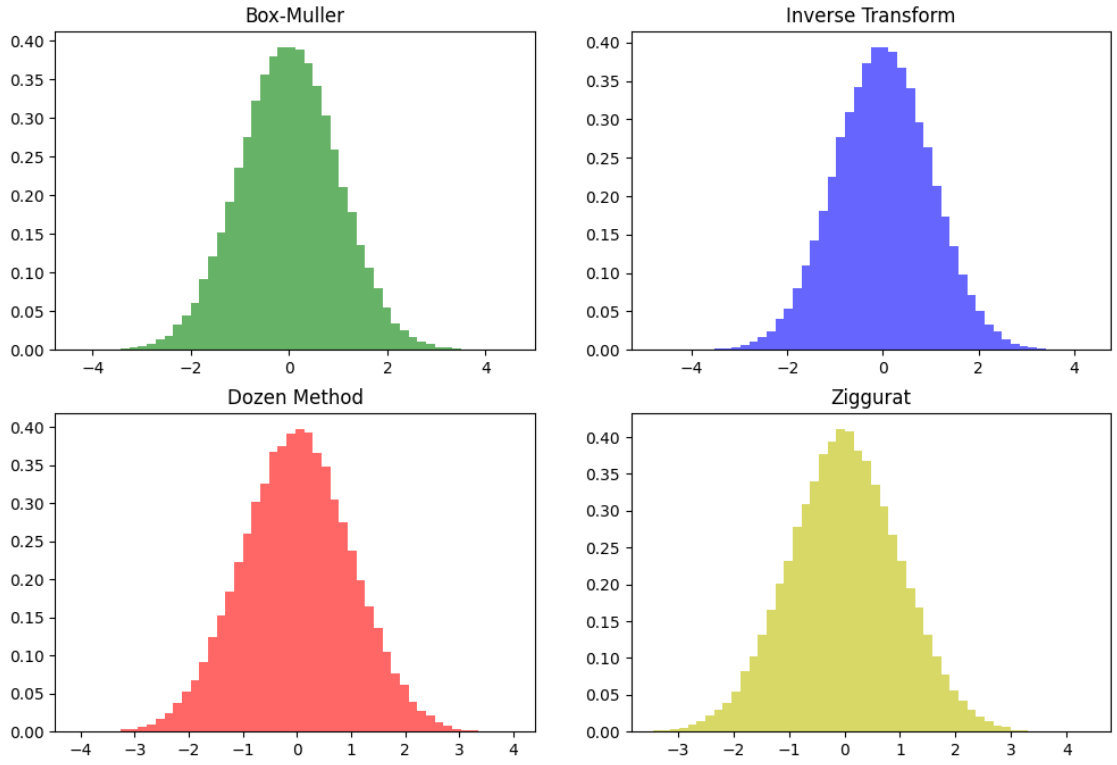
```
    for i in range(n_samples):
        while True:
            warstwa = np.random.randint(0, n)  # Corrected to n
            U = 2 * np.random.uniform() - 1

            # Calculate x
            x = U * tablica[warstwa, 0]

            # Check conditions
            if warstwa < n - 1 and np.abs(x) < tablica[warstwa + 1, 0]:
                samples[i] = x
                break

            # Generating a point from the tail using the so-called fallback␣
↪algorithm
            if warstwa == 0:
                while True:
                    x = -np.log(np.random.uniform()) / tablica[0, 0]
                    y = -np.log(np.random.uniform())
                    if 2 * y > x**2:
                        samples[i] = x + tablica[0, 0]
                        break
                break

            # Check condition using the exponential function
            else:
                y = tablica[warstwa, 1] + np.random.uniform() *␣
↪(tablica[warstwa - 1, 1] - tablica[warstwa, 1])
                if y < f_normalny(x):
                    samples[i] = x
                    break

    return samples * sigma + mu
```

2.5 WYKRESY

[ ]:
```
num_samples=100000
box_muller_samples = box_muller(num_samples)
inverse_transform_samples =inverse_transform(num_samples)
dozen_example_samples=dozen_method(num_samples)
ziggurat_exp_samples=ziggurat_method(num_samples)
```

[ ]:
```
# Histogramy
import matplotlib.pyplot as plt
plt.figure(figsize=(12, 8))
plt.subplot(2, 2, 1)
plt.hist(box_muller_samples, bins=50, density=True, alpha=0.6, color='g')
```

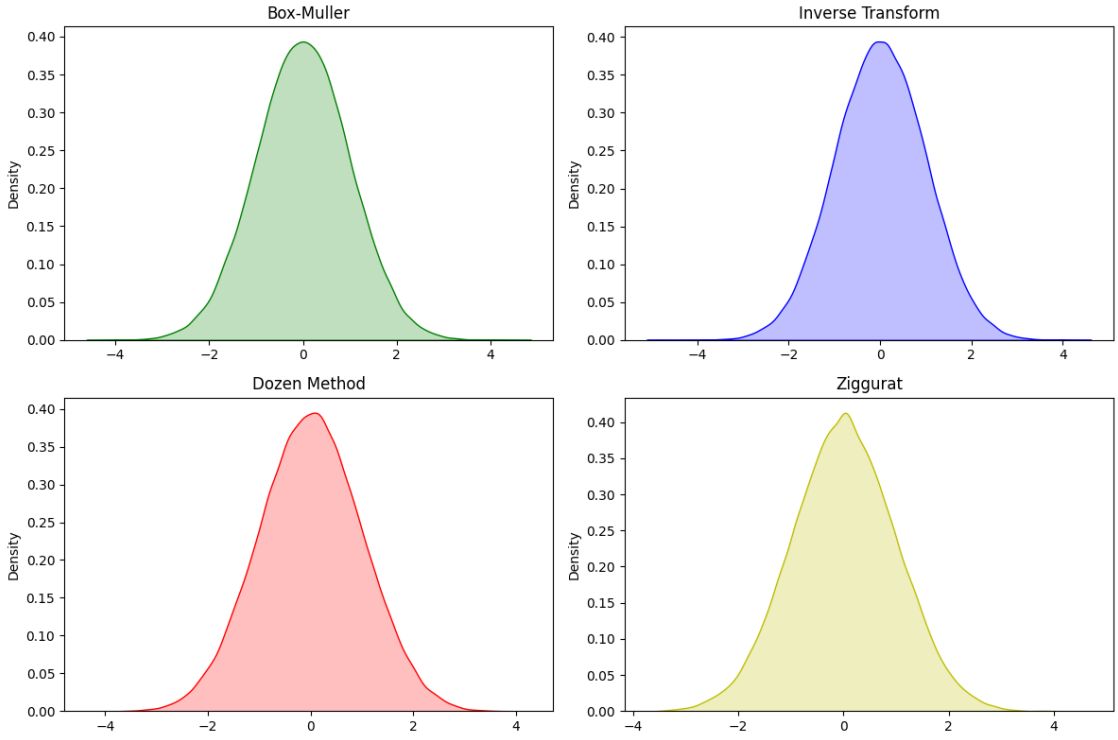
```
plt.title('Box-Muller')
plt.subplot(2, 2, 2)
plt.hist(inverse_transform_samples, bins=50, density=True, alpha=0.6, color='b')
plt.title('Inverse Transform')
plt.subplot(2, 2, 3)
plt.hist(dozen_example_samples, bins=50, density=True, alpha=0.6, color='r')
plt.title('Dozen Method')
plt.subplot(2, 2, 4)
plt.hist(ziggurat_exp_samples, bins=50, density=True, alpha=0.6, color='y')
plt.title('Ziggurat')
plt.show()
```



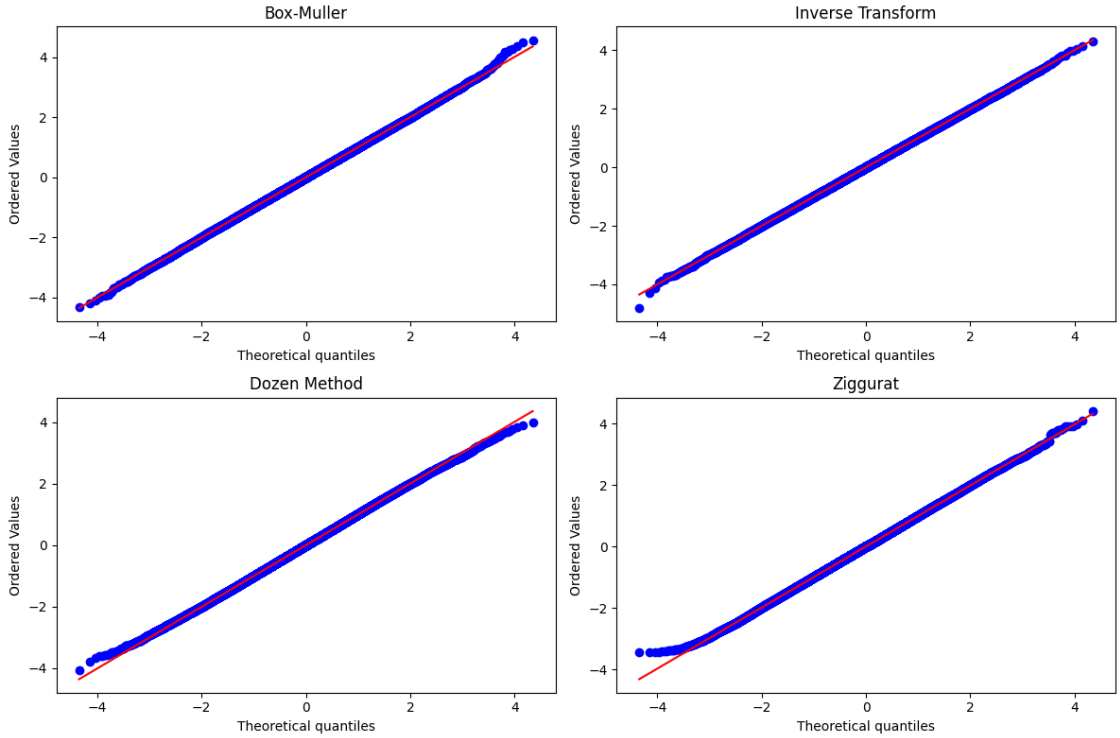
[ ]:
```
#Gęstości
import seaborn as sns
plt.figure(figsize=(12, 8))
plt.subplot(2, 2, 1)
sns.kdeplot(box_muller_samples, shade=True, color='g')
plt.title('Box-Muller')
plt.subplot(2, 2, 2)
sns.kdeplot(inverse_transform_samples, shade=True, color='b')
plt.title('Inverse Transform')
plt.subplot(2, 2, 3)
```

```
sns.kdeplot(dozen_example_samples, shade=True, color='r')
plt.title('Dozen Method')
plt.subplot(2, 2, 4)
sns.kdeplot(ziggurat_exp_samples, shade=True, color='y')
plt.title('Ziggurat')
plt.tight_layout()
plt.show()
```

```
<ipython-input-11-309f667195f9>:4: FutureWarning:

`shade` is now deprecated in favor of `fill`; setting `fill=True`.
This will become an error in seaborn v0.14.0; please update your code.

  sns.kdeplot(box_muller_samples, shade=True, color='g')
<ipython-input-11-309f667195f9>:7: FutureWarning:

`shade` is now deprecated in favor of `fill`; setting `fill=True`.
This will become an error in seaborn v0.14.0; please update your code.

  sns.kdeplot(inverse_transform_samples, shade=True, color='b')
<ipython-input-11-309f667195f9>:10: FutureWarning:

`shade` is now deprecated in favor of `fill`; setting `fill=True`.
This will become an error in seaborn v0.14.0; please update your code.

  sns.kdeplot(dozen_example_samples, shade=True, color='r')
<ipython-input-11-309f667195f9>:13: FutureWarning:

`shade` is now deprecated in favor of `fill`; setting `fill=True`.
This will become an error in seaborn v0.14.0; please update your code.

  sns.kdeplot(ziggurat_exp_samples, shade=True, color='y')
```



[ ]:
```
#QQplot
import scipy.stats as stats
plt.figure(figsize=(12, 8))
plt.subplot(2, 2, 1)
stats.probplot(box_muller_samples, dist="norm", plot=plt)
plt.title('Box-Muller')
plt.subplot(2, 2, 2)
stats.probplot(inverse_transform_samples, dist="norm", plot=plt)
plt.title('Inverse Transform')
plt.subplot(2, 2, 3)
stats.probplot(dozen_example_samples, dist="norm", plot=plt)
plt.title('Dozen Method')
plt.subplot(2, 2, 4)
stats.probplot(ziggurat_exp_samples, dist="norm", plot=plt)
plt.title('Ziggurat')
plt.tight_layout()
plt.show()
```



2.6 Testy statystyczne

[ ]:
```
from scipy.stats import kstest, norm
print("Testy statystyczne:")
print("Box-Muller:", kstest(box_muller_samples, 'norm'))
print("Inverse Transform:", kstest(inverse_transform_samples, 'norm'))
print("Dozen Method:", kstest(dozen_example_samples, 'norm'))
print("Ziggurat:", kstest(ziggurat_exp_samples, 'norm'))
```

```
Testy statystyczne:
Box-Muller: KstestResult(statistic=0.0019502481469655386,
pvalue=0.8406531871715701, statistic_location=-0.17413482650785148,
statistic_sign=1)
Inverse Transform: KstestResult(statistic=0.0036394747846939524,
pvalue=0.1410293269202837, statistic_location=0.4269976347833425,
statistic_sign=-1)
Dozen Method: KstestResult(statistic=0.0029690780090192437,
pvalue=0.34063493662617805, statistic_location=1.157711366200556,
statistic_sign=-1)
Ziggurat: KstestResult(statistic=0.004224707040166031,
pvalue=0.05617252066372258, statistic_location=-0.4657011709740947,
statistic_sign=-1)
```

Wszystkie metody przeszły test K-S zgodności z rozkładem normalnym (wartość p jest większa od

0.05), co oznacza, że próbki generowane przez każdą z metod są zgodne z teoretycznym rozkładem normalnym. Wyniki testów K-S dla każdej z metod:

-Box-Muller: Statystyka testowa 0.0019, p-wartość 0.84

-Inverse Transform: Statystyka testowa 0.0063, p-wartość 0.14

-Dozen Method: Statystyka testowa 0.0029, p-wartość 0.34

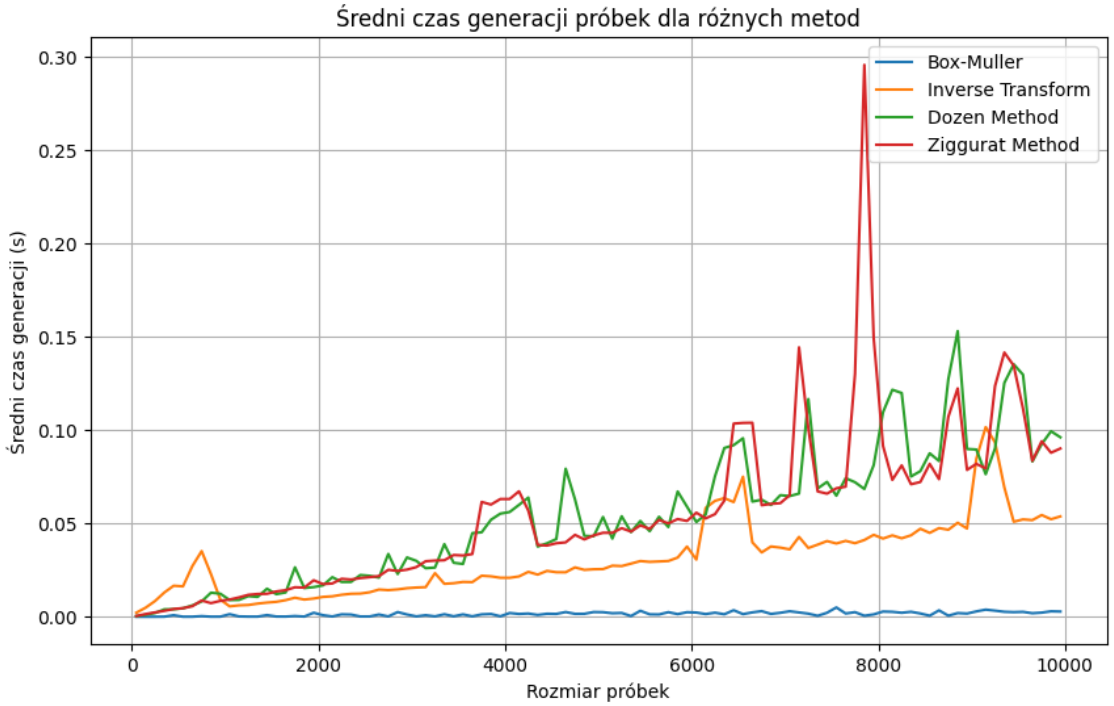
-Ziggurat: Statystyka testowa 0.0042, p-wartość 0.056

2.7 Wykresy szybkości

[ ]:
```python
def czas_sredni(generator):
    lista_rozmiar = np.arange(50, 10000, 100).tolist()
    lista_czasow = []

    for rozmiar in lista_rozmiar:
        suma_czasow = 0
        for _ in range(10):
            start = time.time()
            generator(rozmiar)
            end = time.time()
            suma_czasow += end - start
        lista_czasow.append(suma_czasow / 10)

    return lista_rozmiar, lista_czasow
```

[ ]:
```python
# Obliczanie czasu dla różnych generatorów
rozmiary, czasy_box_muller = czas_sredni(box_muller)
rozmiary, czasy_inverse_transform = czas_sredni(inverse_transform)
rozmiary, czasy_dozen_method = czas_sredni(dozen_method)
rozmiary, czasy_ziggurat_method = czas_sredni(ziggurat_method)
```

[ ]:
```python
# Tworzenie wykresu
plt.figure(figsize=(10, 6))
plt.plot(rozmiary, czasy_box_muller, label='Box-Muller')
plt.plot(rozmiary, czasy_inverse_transform, label='Inverse Transform')
plt.plot(rozmiary, czasy_dozen_method, label='Dozen Method')
plt.plot(rozmiary, czasy_ziggurat_method, label='Ziggurat Method')
plt.xlabel('Rozmiar próbek')
plt.ylabel('Średni czas generacji (s)')
plt.title('Średni czas generacji próbek dla różnych metod')
plt.legend()
plt.grid(True)
plt.show()
```



Analiza czasu generacji próbek

Średni czas generacji próbek dla różnych metod:

- Metoda Box-Muller (niebieska linia) ma najniższy czas generacji próbek spośród wszystkich metod i jest bardzo stabilna w całym zakresie rozmiaru próbek.
- Metoda odwrotnej transformacji (pomarańczowa linia) ma czas generacji nieco wyższy niż Box-Muller, ale także wykazuje stabilność z niewielkimi wahaniami.
- Metoda Dozen (zielona linia) ma znacznie większe wahania czasu generacji próbek, z wyraźnymi skokami, zwłaszcza w zakresie od 4000 do 9000 próbek.
- Metoda Ziggurata (czerwona linia) wykazuje największe wahania czasu generacji próbek z kilkoma znacznymi skokami. Jest mniej stabilna w porównaniu do innych metod.

2.8 WNIOSKI

Metoda Box-Muller jest najstabilniejsza i najwydajniejsza pod względem czasu generacji próbek, co czyni ją najbardziej optymalnym wyborem dla generowania liczb o rozkładzie normalnym.

Metoda odwrotnej transformacji również wykazuje dobrą stabilność i stosunkowo niski czas generacji, ale jest nieco wolniejsza niż Box-Muller.

Metoda Dozen oraz Metoda Ziggurata mają znaczne wahania w czasie generacji próbek, co może wpływać na ich zastosowanie w praktyce, szczególnie w kontekście dużych próbek.

Wszystkie metody są statystycznie zgodne z rozkładem normalnym, co oznacza, że niezależnie od wyboru metody, próbki będą miały poprawne właściwości statystyczne.

3 zadanie 3

3.1 Redukcja wariancji w metodach Monte Carlo

3.2 Wstęp

Zapoznaj się z metodami redukcji wariancji w metodach Monte Carlo: https://en.wikipedia.org/wiki/Variance_reduction W szczególności skupimy się na dwóch metodach: - Metoda odbić lustrzanych: https://en.wikipedia.org/wiki/Antithetic_variates - Metoda zmiennej kontrolnej: https://en.wikipedia.org/wiki/Control_variates

3.3 Kroki

3.3.1 Krok 1

Metodą Monte Carlo oblicz całkę:

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

szacowanie liczby π

3.3.2 Krok 2

Zastosuj jedną z metod redukcji wariancji do ponownego obliczenia zadanej całki.

3.3.3 Krok 3

Przedstaw analizę błędu względem ilości symulacji w formie tabeli lub wykresu.

Jako naszą zmienną kontrolną wybraliśmy pierwsze 2 wyrazy rozwinięcia funkcji podcałkowej w szereg Taylora. Wyznaczono również analitycznie wartość tej całki na obszarze. Następnie zadeklarowano wszystkie zmienne potrzebne do wizualizacji symulacji, oraz do oceny jak bardzo zmienna kontrolna wpływa na wynik końcowy.

[3]:
```
import numpy as np
import matplotlib.pyplot as plt

def f(x):
    return 4 / (1 + x**2)

def g(x):
    return 4 * (1 - x**2)

Egx = 8 / 3


n = 10000


better_count = 0
worse_count = 0
```

```
I1_values = []
I2_values = []
I_values = []
I1_errors = []
I_errors = []
```

Poniżej główna pętla, w której pierw zliczamy wartość całki korzystając z metody Monte Carlo, po czym korzystając ze wzoru, który wylicza optymalne c.

Niech W = f(X)+c(g(X)−Eg),

wtedy mamy EW = E(f(X))+c(g(X)−E(g(X))) = E(f(X)),

wariancja będzie równa VarW = Var(f(X)) + c^2*Var(g(X))+2cCov(f(X), g(X)).

Po obustronnym zróżniczkowaniu równania względem c i uproszczeniu otrzymujemy

c = -Cov(f(X), g(X))/Var(g(X)), co juz znajduje sie w kodzie.

Przy okazji zliczane są wszystkie wartości przed i po skorzystaniu z metody. Zliczane jest również ile razy skorzystanie ze zmiennej kontrolnej nas zbliżyło do rzeczywistego wyniku.

[4]:
```
for i in range(n):
    Us = np.random.uniform(0, 1, n)
    fus = f(Us)
    gus = g(Us)
    fgus = fus + gus

    VarG = np.var(gus)
    VarF = np.var(fus)
    VarFG = np.var(fgus)

    CovFG = (VarFG - VarF - VarG) / 2
    c = -CovFG / VarG

    I1 = np.mean(fus)
    I2 = c * (np.mean(gus) - Egx)
    I = I1 + I2

    I1_values.append(I1)
    I2_values.append(I2)
    I_values.append(I)

    I1_errors.append(abs(I1 - np.pi))
    I_errors.append(abs(I - np.pi))

    pre_residual = abs(I1 - np.pi)
    post_residual = abs(I - np.pi)
```

```
    if post_residual < pre_residual:
        better_count += 1
    else:
        worse_count += 1
```

Końcówka kodu odpowiada za wyświetlenie wykresów oraz innych wyznaczonych informacji.

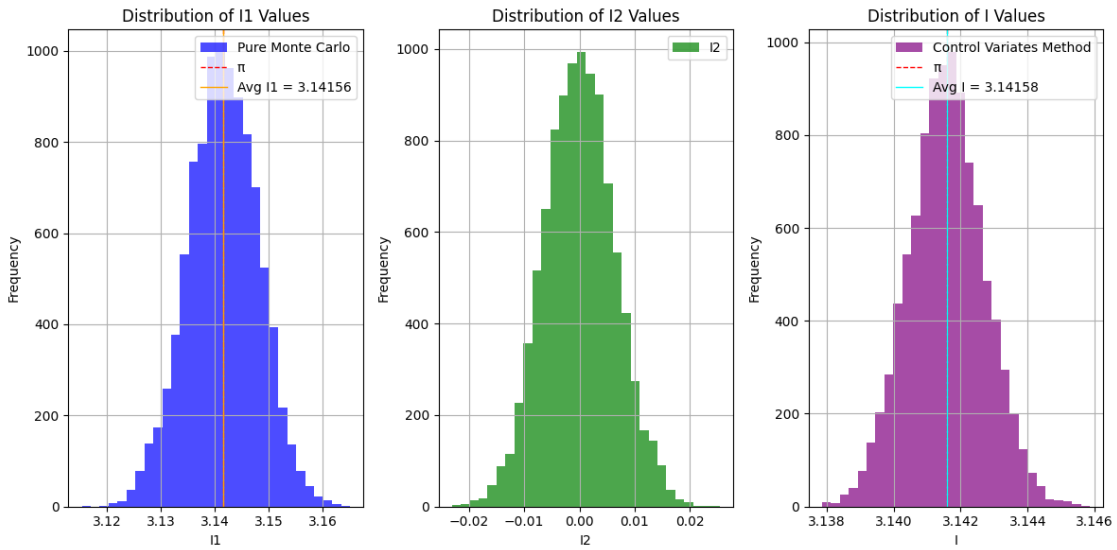
[7]:
```
average_error_I1 = np.mean(I1_errors)
average_error_I = np.mean(I_errors)
average_I1 = np.mean(I1_values)
average_I = np.mean(I_values)

print(f"Better estimations: {better_count}, Worse estimations: {worse_count},␣
 ↪Total: {n}")
print(f"Average error I1: {average_error_I1}")
print(f"Average error I: {average_error_I}")
print(f"Average value I1: {average_I1}")
print(f"Average value I: {average_I}")
```

```
Better estimations: 8962, Worse estimations: 1038, Total: 10000
Average error I1: 0.005211664126452378
Average error I: 0.0008958606497045818
Average value I1: 3.14155702365022
Average value I: 3.141580715597334
```

Wyświetlamy otrzymane wyniki. Wiadomo z analitycznych obliczeń, że poprawny wynik to π. Z wykresu można odczytać, że po skorzystaniu z metody zmiennej kontrolnej nasz wynik oscyluje dużo bliżej poprawnej odpowiedzi - nie jest aż tak rozrzucony jak w przypadku samego skorzystania z metody Monte Carlo.

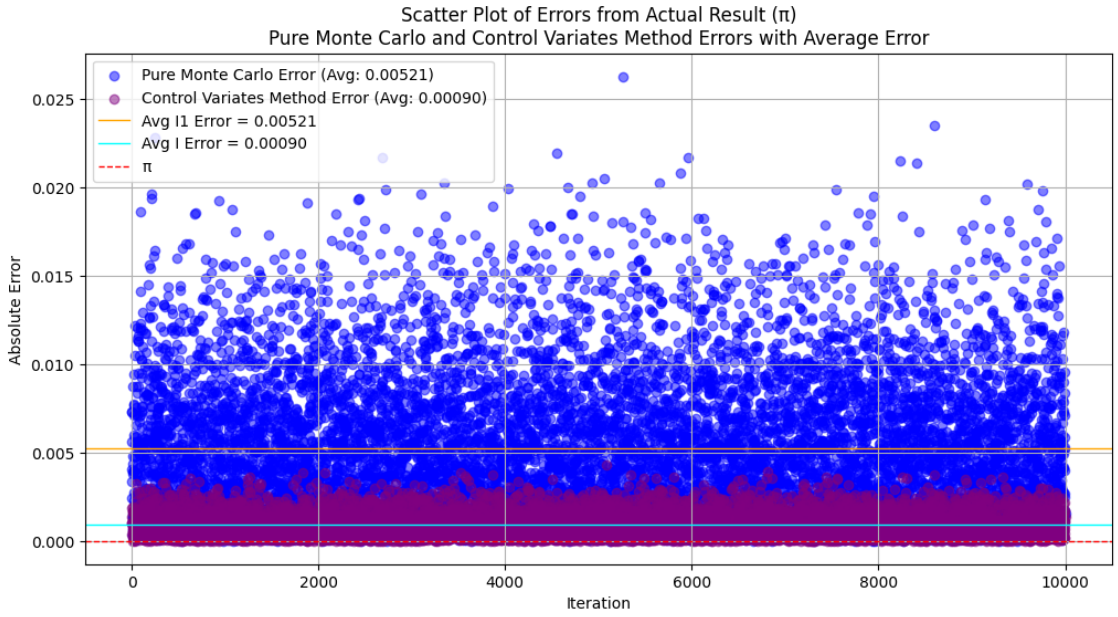
[8]:
```
# Visualization
plt.figure(figsize=(12, 6))

# Plot I1 values
plt.subplot(1, 3, 1)
plt.hist(I1_values, bins=30, color='blue', alpha=0.7, label='Pure Monte Carlo')
plt.axvline(np.pi, color='red', linestyle='--', linewidth=1, label=' ')
plt.axvline(average_I1, color='orange', linestyle='-', linewidth=1, label=f'Avg␣
 ↪I1 = {average_I1:.5f}')
plt.title('Distribution of I1 Values')
plt.xlabel('I1')
plt.ylabel('Frequency')
plt.legend()
plt.grid(True)

# Plot I2 values
plt.subplot(1, 3, 2)
```

```
plt.hist(I2_values, bins=30, color='green', alpha=0.7, label='I2')
plt.title('Distribution of I2 Values')
plt.xlabel('I2')
plt.ylabel('Frequency')
plt.legend()
plt.grid(True)

# Plot combined I values
plt.subplot(1, 3, 3)
plt.hist(I_values, bins=30, color='purple', alpha=0.7, label='Control Variates␣
 ↪Method')
plt.axvline(np.pi, color='red', linestyle='--', linewidth=1, label=' ')
plt.axvline(average_I, color='cyan', linestyle='-', linewidth=1, label=f'Avg I␣
 ↪= {average_I:.5f}')
plt.title('Distribution of I Values')
plt.xlabel('I')
plt.ylabel('Frequency')
plt.legend()
plt.grid(True)

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



Poniżej jest wykres, przedstawiający jak oddaleni od wyniku jesteśmy w zależności od tego, jakiej metody używaliśmy. Z tego wykresu również wynika, że metoda zmiennej kontrolnej daje nam dużo lepiej przybliżony wynik, z mniejszą wariancją.

[6]:
```
# Scatter plot of errors
plt.figure(figsize=(12, 6))
plt.scatter(range(n), I1_errors, color='blue', alpha=0.5, label=f'Pure Monte␣
 ↪Carlo Error (Avg: {average_error_I1:.5f})')
plt.scatter(range(n), I_errors, color='purple', alpha=0.5, label=f'Control␣
 ↪Variates Method Error (Avg: {average_error_I:.5f})')
plt.axhline(average_error_I1, color='orange', linestyle='-', linewidth=1,␣
 ↪label=f'Avg I1 Error = {average_error_I1:.5f}')
plt.axhline(average_error_I, color='cyan', linestyle='-', linewidth=1,␣
 ↪label=f'Avg I Error = {average_error_I:.5f}')
plt.axhline(0, color='red', linestyle='--', linewidth=1, label=' ')
plt.title('Scatter Plot of Errors from Actual Result ( )\nPure Monte Carlo and␣
 ↪Control Variates Method Errors with Average Error')
plt.xlabel('Iteration')
plt.ylabel('Absolute Error')
plt.legend()
plt.grid(True)
plt.show()
```



3.4 Wnioski

W przeprowadzonej analizie wykorzystano metodę zmiennej kontrolnej, aby zmniejszyć wariancję w estymacji całki metodą Monte Carlo. Metoda ta polega na użyciu dodatkowej zmiennej, która jest powiązana z funkcją, którą całkujemy. Dzięki temu można uzyskać bardziej dokładne wyniki przy mniejszej liczbie próbek. Symulacje pokazały, że użycie zmiennej kontrolnej znacząco zmniejszyło błąd estymacji w porównaniu do standardowej metody Monte Carlo. Średni błąd estymacji był

mniejszy, co potwierdza skuteczność tej metody. Zmniejszenie wariancji pozwala na bardziej efektywne obliczenia, co jest szczególnie ważne w praktycznych zastosowaniach, gdzie zasoby obliczeniowe są ograniczone.

4 zadanie 4

4.0.1 Warunkowa wartość oczekiwana

Wartość oczekiwana $f(X)$ zmiennej Y warunkowanej zmienną X ma bardzo ciekawą interpretację, mianowicie spełnia własność

$$\mathbb{E}(Y|X=x) = f(x) = \underset{g}{\operatorname{argmin}}\mathbb{E}\Big((Y-g(X))^2\Big)$$

Oznacza to, że $\mathbb{E}(Y|X)$ to najlepsze przybliżenie w sensie L^2 zmiennej Y korzystające z danych pochodzących ze zmiennej X. Sprawdź symulacyjnie następujące stwierdzenia.

4.1 Część pierwsza

- Jeśli X, Y są zmiennymi niezależnymi a $\mathbb{E}(Y)=0$, to dla $Z = XY + \sin X$ zachodzi $\mathbb{E}(Z|X) = \sin X$. Jako przykład możesz wziąć np. X, Y iid $\mathcal{N}(0,1)$. Zrób scatterplot $X \mapsto Z|X$ wysymulowanych wartości i zaznacz na nim wartości estymowane oraz teoretyczne $\mathbb{E}(Z|X)$.

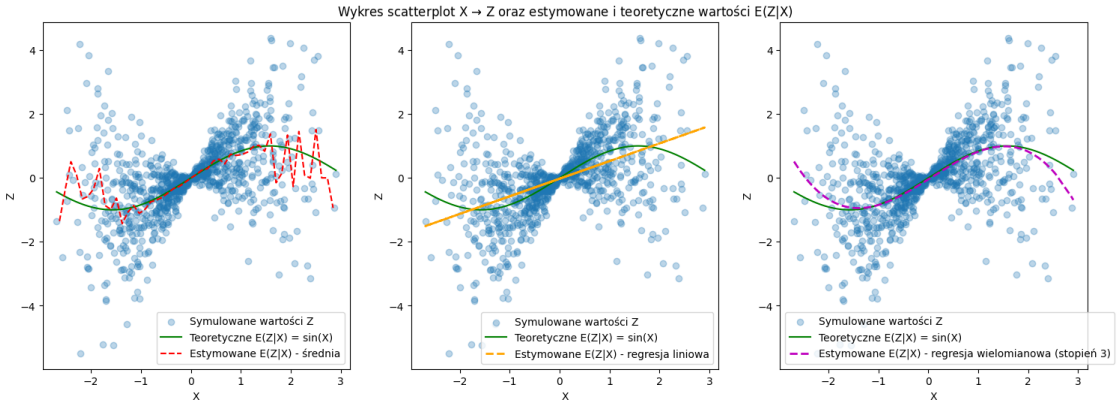
Do estymowania wartości oczekowanej jako funkcję g użyliśmy średniej, regresji liniowej oraz regresji wielomianowej trzeciego stopnia. Na wykresach widzimy porównanie tych metod. Jak widać, regresja wielomianowa oraz średnia (patrząc na przedział [-1,1]) daje bardzo dobrą estymację wartości $\mathbb{E}(Z|X)$. Regresja liniowa dla funkcji sinus nie daje zadawalającej estymacji, co wynika z właściwości obydwu funkcji.

[ ]:
```
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline
```

[ ]:
```
def E(Z, X):
  # Estymujemy wartość oczekiwaną dzieląc próbkę na przedziały
  # Liczba przedziałów
  num_bins = 50

  # Przedziały
  bins = np.linspace(X.min(), X.max(), num_bins)
  bin_centers = 0.5 * (bins[1:] + bins[:-1])

  # Indeksy przedziałów dla X
  indices = np.digitize(X, bins)

  # Estymowane wartości oczekiwane E(Z|X)

  # Średnia
  E_ZX = [Z[indices == i].mean() if np.any(indices == i) else 0 for i in␣
 ↪range(1, num_bins)]
```

```
# Regresja liniowa
model = LinearRegression()
model.fit(X.reshape(-1,1), Z)

E_ZX1 = model.predict(X.reshape(-1,1))

# Regresja wielomianowa
degree = 3  # Stopień wielomianu
polynomial_model = make_pipeline(PolynomialFeatures(degree),␣
↪LinearRegression())
polynomial_model.fit(X.reshape(-1,1), Z)

E_ZX2 = polynomial_model.predict(X.reshape(-1,1))

# Porównanie estymowanych wartości oczekiwanych
plt.figure(figsize=(18, 6))
xs = np.linspace(X.min(), X.max(), n)
plt.subplot(1,3,1)
plt.scatter(X, Z, alpha=0.3, label='Symulowane wartości Z')
plt.plot(xs, np.sin(xs), color='green', label='Teoretyczne E(Z|X) = sin(X)')
plt.plot(bin_centers, E_ZX, color='red', linestyle='dashed',␣
↪label='Estymowane E(Z|X) - średnia')
plt.xlabel('X')
plt.ylabel('Z')
plt.legend()
plt.subplot(1,3,2)
plt.scatter(X, Z, alpha=0.3, label='Symulowane wartości Z')
plt.plot(xs, np.sin(xs), color='green', label='Teoretyczne E(Z|X) = sin(X)')
plt.plot(X, E_ZX1, color='orange', linestyle='dashed', label='Estymowane␣
↪E(Z|X) - regresja liniowa', linewidth=2)
plt.xlabel('X')
plt.ylabel('Z')
plt.title(r'Wykres scatterplot X $\to$ Z oraz estymowane i teoretyczne␣
↪wartości E(Z|X)')
plt.legend()
plt.subplot(1,3,3)
plt.scatter(X, Z, alpha=0.3, label='Symulowane wartości Z')
plt.plot(xs, np.sin(xs), color='green', label='Teoretyczne E(Z|X) = sin(X)')
plt.plot(np.sort(X, axis=0), E_ZX2[np.argsort(X, axis=0)], color='m',␣
↪linestyle='dashed', label=f'Estymowane E(Z|X) - regresja wielomianowa␣
↪(stopień {degree})', linewidth=2)
plt.xlabel('X')
plt.ylabel('Z')
plt.legend()
plt.show()
```

[ ]:
```
n = 1000
X = np.random.normal(size = n)
Y = np.random.normal(size = n)
Z = X*Y + np.sin(X)
E(Z, X)
```

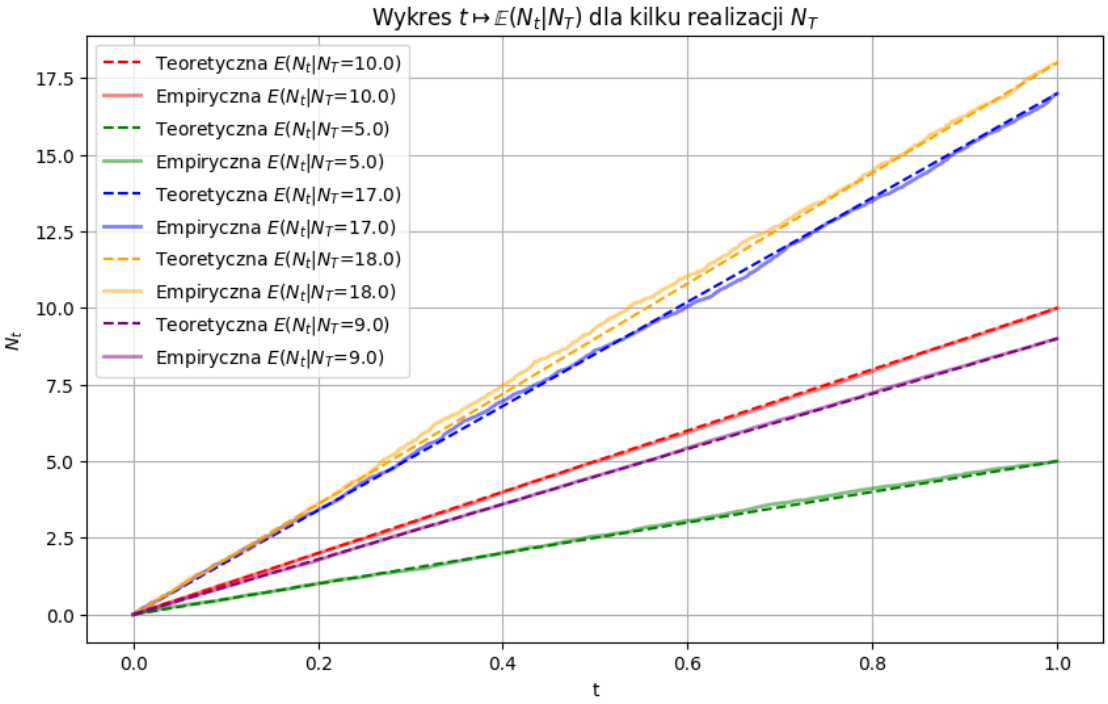


4.2 Część druga

- Gdy N jest procesem Poissona o intensywnosci , to dla $T \geq t \geq 0$ zachodzi $\mathbb{E}(N_t|N_T) = tN_T/T$. Oznacza to, że np. zakładając przybywanie klientów do sklepu zgodnie z procesem Poissona i mając dane na temat dotychczasowej ilości klientów w sklepie w chwili T (czyli N_T), to najlepszym przybliżeniem dotychczasowej chwili klientów w chwili $t < T$ równej N_t jest tN_T/T. Jako przykład możesz wziąć np. $T = 1, \lambda = 10$ (zależnie od metody, niekoniecznie potrzebujesz parametru intensywności). Zrób wykres $t \mapsto \mathbb{E}(N_t|N_T)$ dla kilku możliwych realizacji N_T i zaznacz na nim wartości estymowane.

[ ]:
```
# Parametry procesu Poissona
T = 1
lambda_ = 10
num_realizations = 10000
num_points = 1000

t_values = np.linspace(0, T, num_points)

N_T_values = np.zeros(num_realizations)
N_t_all = np.zeros((num_realizations, num_points))

# Symulacja procesu Poissona dla każdej realizacji
for i in range(num_realizations):
    inter_arrival_times = np.random.exponential(1/lambda_, int(lambda_ * T *␣
↪2))  # 2x większy rozmiar dla pewności
    arrival_times = np.cumsum(inter_arrival_times)
    arrival_times = arrival_times[arrival_times <= T]
```

```
    # Obliczanie N_t
    N_t = np.zeros(num_points)
    event_count = 0
    for j, t in enumerate(t_values):
        while event_count < len(arrival_times) and arrival_times[event_count]␣
 ↪<= t:
            event_count += 1
        N_t[j] = event_count

    N_t_all[i, :] = N_t
    N_T_values[i] = N_t[-1]  # wartość N_T

# Wybór kilku różnych wartości N_T
unique_N_T_values = np.unique(N_T_values)
chosen_N_T_values = np.random.choice(unique_N_T_values, 5, replace=False)

# Estymacja empiryczna
empirical_means_dict = {N_T: np.mean(N_t_all[N_T_values == N_T, :], axis=0) for␣
 ↪N_T in chosen_N_T_values}

# Obliczenie teoretycznej wartości oczekiwanej dla wybranych wartości N_T
theoretical_means_list = {N_T: t_values * N_T / T for N_T in chosen_N_T_values}

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

# Wykres teoretycznych i empirycznych wartości oczekiwanych dla wybranych␣
 ↪wartości N_T
colors = ['red', 'green', 'blue', 'orange', 'purple']
for N_T, color in zip(chosen_N_T_values, colors):
    plt.plot(t_values, theoretical_means_list[N_T], linestyle='dashed',␣
 ↪color=color, label=r'Teoretyczna $E(N_t|N_T$='+f'{N_T})')
    plt.plot(t_values, empirical_means_dict[N_T], color=color,alpha=0.5,␣
 ↪linewidth=2, label=r'Empiryczna $E(N_t|N_T$='+f'{N_T})')

plt.xlabel('t')
plt.ylabel(r'$N_t$')
plt.title(r'Wykres $t \mapsto \mathbb{E}(N_t|N_T)$ dla kilku realizacji $N_T$')
plt.legend()
plt.grid(True)
plt.show()
```



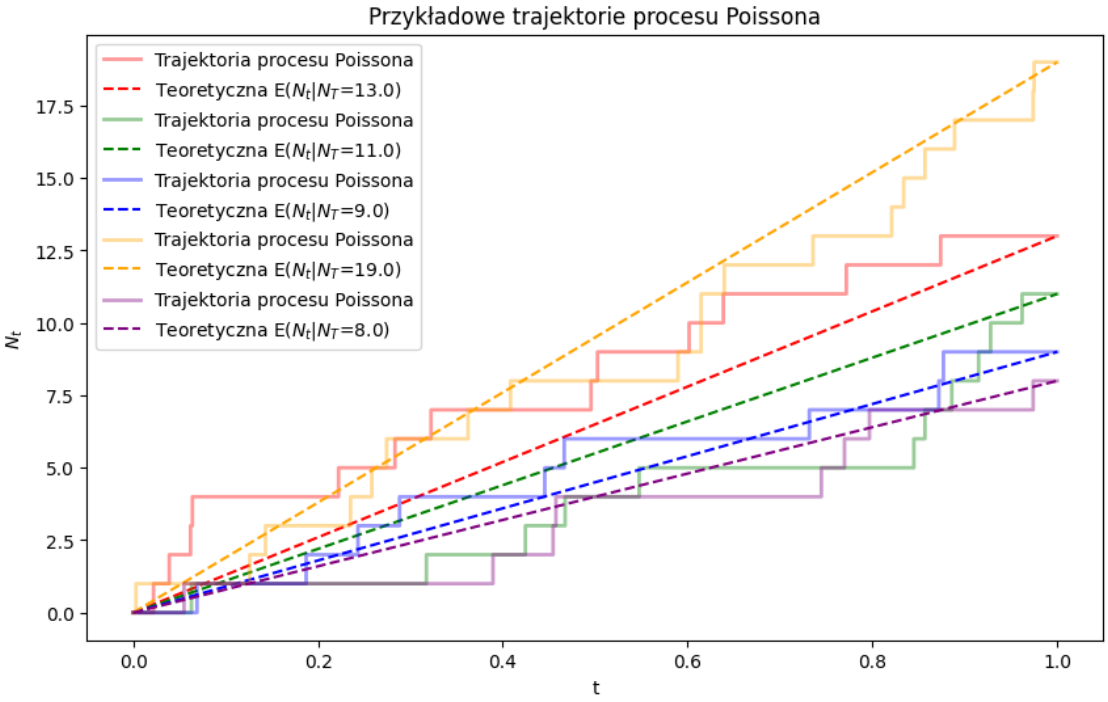
Wykres pokazuje, jak $\mathbb{E}(N_t|N_T)$ zmienia się wraz z t dla różnych realizacji N_T. Zgodnie z teoretycznym twierdzeniem, wartości oczekiwane $\mathbb{E}(N_t|N_T)$ rosną liniowo od 0 do N_T wg wzoru $\mathbb{E}(N_t|N_T) = \frac{tN_T}{T}$. Dla różnych realizacji N_T otrzymujemy różne linie na wykresie, co pokazuje, jak ilość klientów w sklepie w chwili t jest proporcjonalna do całkowitej ilości klientów N_T w chwili T.

Widzimy, że wartości teoretyczne oraz empiryczne są do siebie bardzo zbliżone, co potwierdza, że nasz sposób estymacji jest dobry.

Dodatkowo wysymulujmy kilka trajektorii. Na wykresie widać, że nasza estymacja jest zgodna z rzeczywistą trajektorią (wykresy trajektorii podążają mniej więcej tak jak $\frac{tN_T}{T}$).

[ ]:
```
plt.figure(figsize=(10, 6))
colors = ['red', 'green', 'blue', 'orange', 'purple']
for N_T, color, i in zip(N_t_all[0:6], colors, range(0,6)):
  plt.step(np.linspace(0,1,1000), N_T, linewidth=2, color=color, alpha=0.4,␣
 ↪label='Trajektoria procesu Poissona')
  plt.plot(np.linspace(0,1,1000),  np.linspace(0,1,1000) * N_T[-1] / T,␣
 ↪linestyle='dashed', color=color, label=r'Teoretyczna␣
 ↪E($N_t|N_T$='+f'{N_T[-1]})')

plt.xlabel('t')
plt.ylabel(r'$N_t$')
plt.title('Przykładowe trajektorie procesu Poissona')
plt.legend()
```

```
plt.show()
```

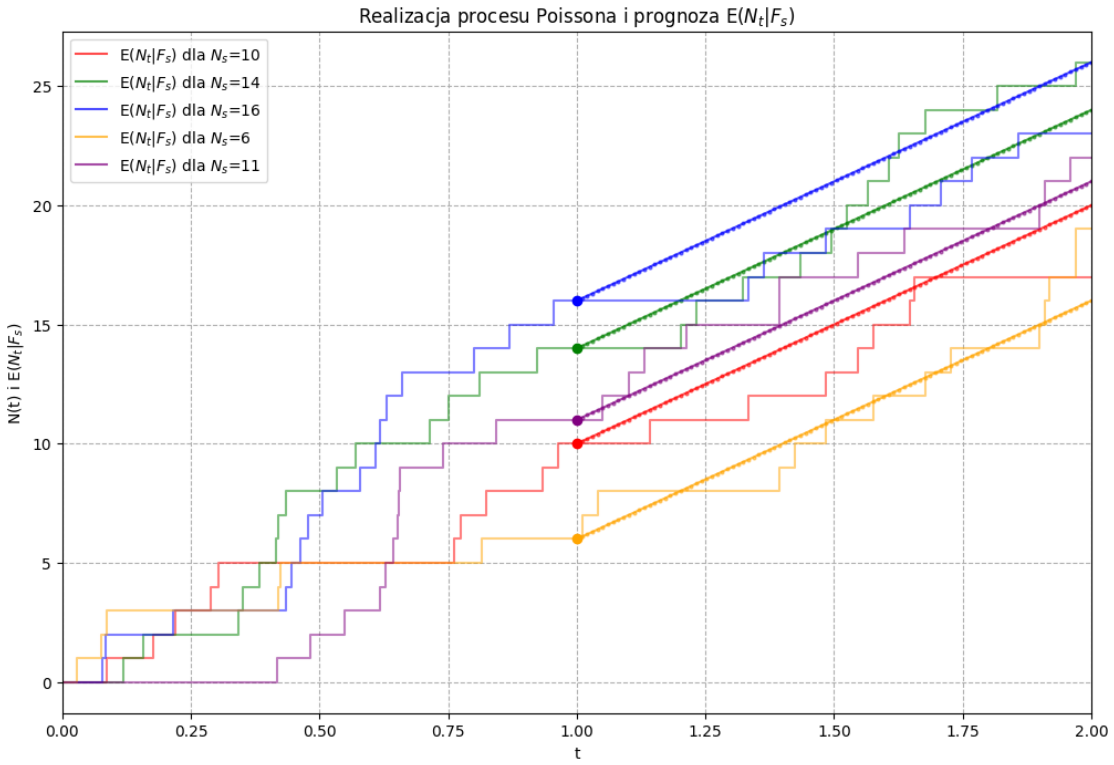


4.3 Część trzecia

- Gdy N jest procesem Poissona o intensywnosci , to dla $t \geq s \geq 0$ 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 np. zakładając przybywanie klientów do sklepu zgodnie z procesem Poissona i mając dane na temat dotychczasowej ilości klientów w sklepie w każdej chwili spełniającej $0 \leq \omega \leq s$ (czyli $\mathcal{F}_s$), to najlepszym przybliżeniem dotychczasowej chwili klientów w chwili $t \geq s$ jest $N_s + \lambda(t-s)$. Jako przykład możesz wziąć np. $s = 1, \lambda = 10$. Zrób wykres $t \mapsto \mathbb{E}(N_t|\mathcal{F}_s)$ dla kilku możliwych realizacji N_s, $s \leq t$, i zaznacz na nim wartości estymowane. Podpowiedź: Wygeneruj kilka trajektorii N_ω, $\omega \leq s$ i wybierz z nich te, które przybierają różne wartości N_s (w tym kroku nie potrzebowałeś użycia parametru intensywności). Następnie skorzystaj z własności Markowa i dla każdej z wybranych realizacji $N_\omega, \omega \leq s$ dosymuluj trajektorie N_ω dla $s < \omega \leq t$, za pomocą których wyestymuj $\mathbb{E}(N_t|\mathcal{F}_s)$.

[ ]:
```
def prediction(s=1, lambda_=10, num_realizations=5, T=4):
    NT_samples = []

    plt.figure(figsize=(12, 8))

    for i, color in zip(range(num_realizations), colors):
        NT_sample = np.random.poisson(lambda_ * s)
        while NT_sample in [sample[0] for sample in NT_samples]:
```

```
            NT_sample = np.random.poisson(lambda_ * s)
        NT_samples.append((NT_sample, color))
        # generowanie trajektorii do punktu s
        t_realization = np.append(0, np.sort(np.random.uniform(0, s,␣
↪NT_sample)))
        Nt_realization = np.arange(0, NT_sample + 1)

        # Obliczanie wartości oczekiwanej
        E_Nt_given_Fs = NT_sample + lambda_ * (t_values - s)

        # plotowanie części trajektorii
        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)

        # plotowanie predykcji
        plt.plot(t_values, E_Nt_given_Fs, linestyle="solid", alpha=0.7,␣
↪color=color, label=r'E($N_t | F_s$) dla $N_s$='+f'{NT_sample}')
        plt.plot(t_values, E_Nt_given_Fs, "o", markersize=2, alpha=0.5,␣
↪color=color)

        # obliczanie i plotowanie reszty trajektorii
        inter_arrival_times = np.random.exponential(1 / lambda_, int(lambda_ *␣
↪(T - s) * 2))
        arrival_times = np.cumsum(inter_arrival_times)
        arrival_times = arrival_times[arrival_times <= T - s]

        N_t_extended = np.zeros_like(t_values)
        event_count = 0
        for j, t in enumerate(t_values):
            while event_count < len(arrival_times) and␣
↪arrival_times[event_count] <= t - s:
                event_count += 1
            N_t_extended[j] = NT_sample + event_count

        plt.step(t_values, N_t_extended, where="post", alpha=0.5, color=color)

    plt.xlabel("t")
    plt.ylabel(r"N(t) i E($N_t | F_s$)")
    plt.xlim(0, T)
    plt.title(r'Realizacja procesu Poissona i prognoza E($N_t | F_s$)')
    plt.grid(True, linestyle="--")
    plt.legend(loc="upper left")
    plt.show()

prediction(num_realizations=6, T=2)
```



Estymowane trajektorie są zgodne z przewidywaniami teoretycznymi.

4.4 Wnioski

Przeanalizowaliśmy warunkową wartość oczekiwaną oraz zachowanie procesu Poissona.

Obliczyliśmy zmienną Z zgodnie z podanym wzorem i wyestymowaliśmy jej wartość oczekiwaną, używając np. średniej. Na wykresie porównaliśmy wartości teoretyczne z estymowanymi.

Wyestymowaliśmy wartość oczekiwaną procesu Poissona i potwierdziliśmy na wykresie, że podany wzór daje dobre przybliżenie.

Również predykcja procesu od chwili s potwierdziła się z teoretycznym wzorem.

Udało nam się zaimplementować wskazane własności, co potwierdzono na wykresach - wartości estymowane są zgodne z teoretycznymi, z czego można wnioskować o poprawności użytych metod.

5 zadanie 5

6 Zadanie Ruiny

Niech X_t będzie procesem ruiny modelu Craméra-Lundberga:

$Xt = u + ct - \sum i = 0Nt\xi i$

gdzie $ t, c, u 0 $, $\xi_i \sim \text{Exp}(\eta)$, $ _i _j $ dla $ i j $, $ E(_i) = $, a $ N_t $ jest procesem Poissona o intensywności λ.

Czasem ruiny klasycznej nazywamy zmienną $\tau = \inf t > 0 \mid X_t < 0$. Prawdopodobieństwem ruiny w czasie nieskończonym nazywamy funkcję:

$\psi(u, c) = P(\tau < \infty)$

Wzór Pollaczka-Chinczyna mówi, że:

$\psi(u, c) = \eta\lambda ce - (1\eta - \lambda c)u$

Zweryfikuj symulacyjnie ten wynik

W tym celu skorzystaj z pomocniczego prawdopodobieństwa ruiny w czasie skończonym T, tj. $\Psi(u, c, T) = P(\tau < T)$, $T > 0$, dla odpowiednio dużego T. Sporządź wykresy funkcji $ u (u, c_0) $ dla ustalonych c_0 oraz $ c (u_0, c) $ dla ustalonych u_0 (kilka trajektorii na jednym wykresie) i porównaj je z wartościami wyestymowanymi. Analiza funkcji odwrotnej

Powtórz analizę dla funkcji odwrotnej:

$c(u, \psi) = \lambda uW0(u\psi eu\eta\eta)$

gdzie $ W_0 $ to gałąź funkcji W Lamberta zdefiniowana poprzez równanie $ W_0(x e^x) = x $ dla $ x 0 $.

Ta analiza odpowiada szukaniu wymaganej wartości wpłat przy danym kapitale początkowym w celu osiągnięcia wymaganego prawdopodobieństwa ruiny. Zweryfikuj symulacyjnie ten wynik. Tym razem sporządź wykresy funkcji $ u c(u, _0) $ dla ustalonych ψ_0 (kilka trajektorii na jednym wykresie) i porównaj je z wartościami wyestymowanymi. Pamiętaj

Gdy $ c $ to zachodzi $\psi = 1$

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import expon
import scipy.stats
import matplotlib.pyplot as plt
from scipy.special import lambertw
```

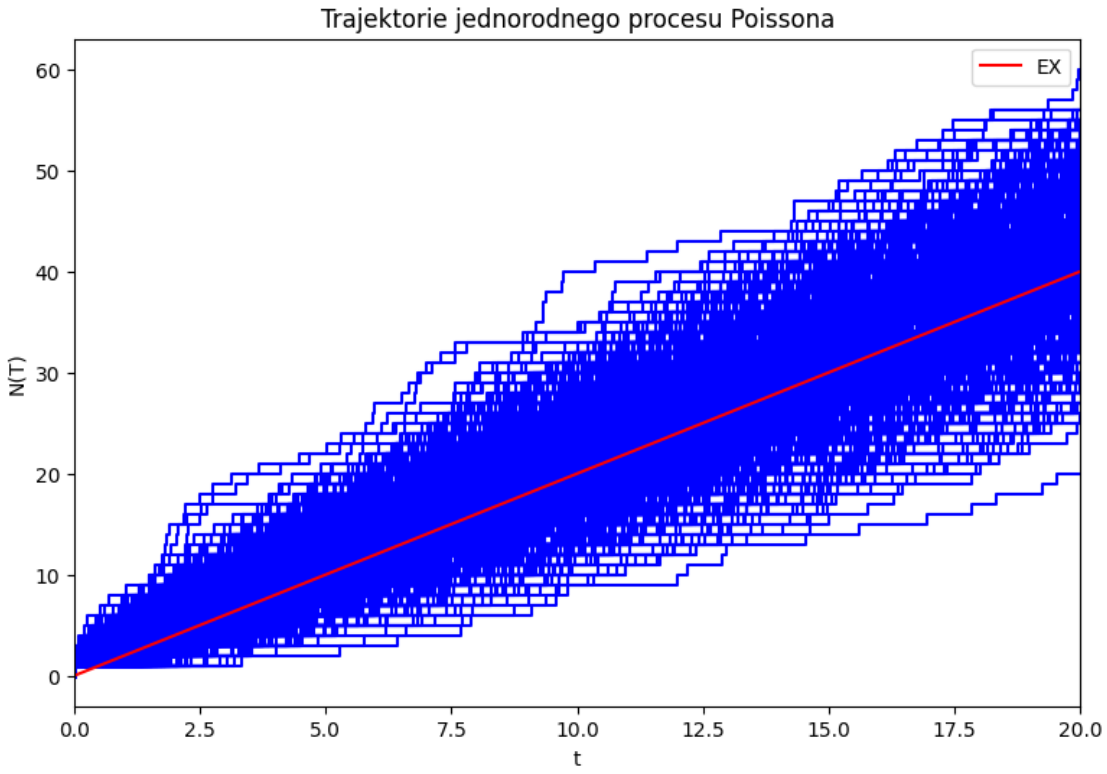
1.Proces Ryzyka Jednorodny proces Poissona to proces stochastyczny $N(t)$ spełniający następujące warunki: - $N(0) = 0$; - $N(t)$ ma niezależne przyrosty; - $N(t)$ ma stacjonarne przyrosty; - $N(t) \sim Poiss(\lambda t)$

Algorytm:

1. Wstaw $I = 0$, $t = 0$.
2. Generuj $U \sim U(0, 1)$.
3. Wstaw $t = t - \frac{1}{\lambda} \log U$. Jeżeli $t > T$, STOP.
4. Wstaw $I = I + 1$, $S_I = t$.
5. Wróć do punktu 2.

Poniższy kod implementuje powyższy algorytm oraz rysuje trajektorie jednorodnego procesu Poissona:

[ ]:
```
def t_poisson(lambda1, T):
    I = 0
    t = 0
    t_list = []
    t_list.append(t)
    while t < T:
        U = np.random.uniform()
        t = t - (1/lambda1)*np.log(U)
        t_list.append(t)
        I += 1
    return t_list

steps = []

plt.figure(figsize=(9,6))
for i in range(1000):
    s = t_poisson(2, 20)
    steps.append(len(s))
    plt.step(s, list(range(len(s))), color='blue')

x = np.linspace(0, 20, len(s))
plt.plot(x, 2*x, color='red', label='EX')
plt.xlabel('t')
plt.ylabel('N(T)')
plt.xlim(0, 20)
plt.title("Trajektorie jednorodnego procesu Poissona")
plt.legend()
plt.show()
```



Proces Ryzyka to proces stochastyczny opisujący kapitał firmy ubezpieczeniowej. Wyraża się następującym wzorem:

$$R(t) = u + c(t) - \sum_{i=1}^{N(t)} X_i, t \in [0, T]$$

gdzie:

u>0to kapitał początkowy,

c(t) to funkcja deterministyczna oznaczająca premię,

Xi to niezależne zmienne losowe reprezentujące wysokość wypłacanych odszkodowań,

N(t) to jednorodny proces Poissona z intensywnością >0 >0.

W klasycznym procesie ryzyka mamy:

$$c(t) = (1 + \theta)\lambda t$$

,gdzie $\mu = EX_i$ i jest parametrem odpowiedzialnym za wysokość premii.

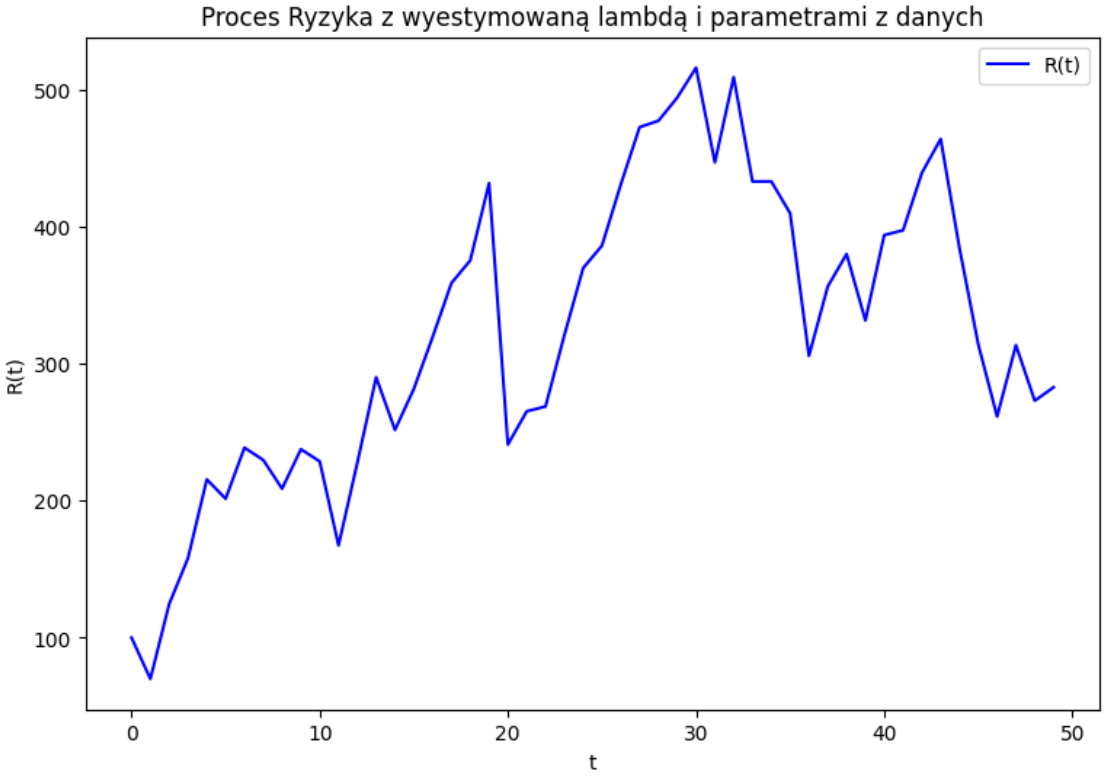
Poniższy kod implementuje powyższy algorytm i rysuje trajektorie procesu ryzyka:

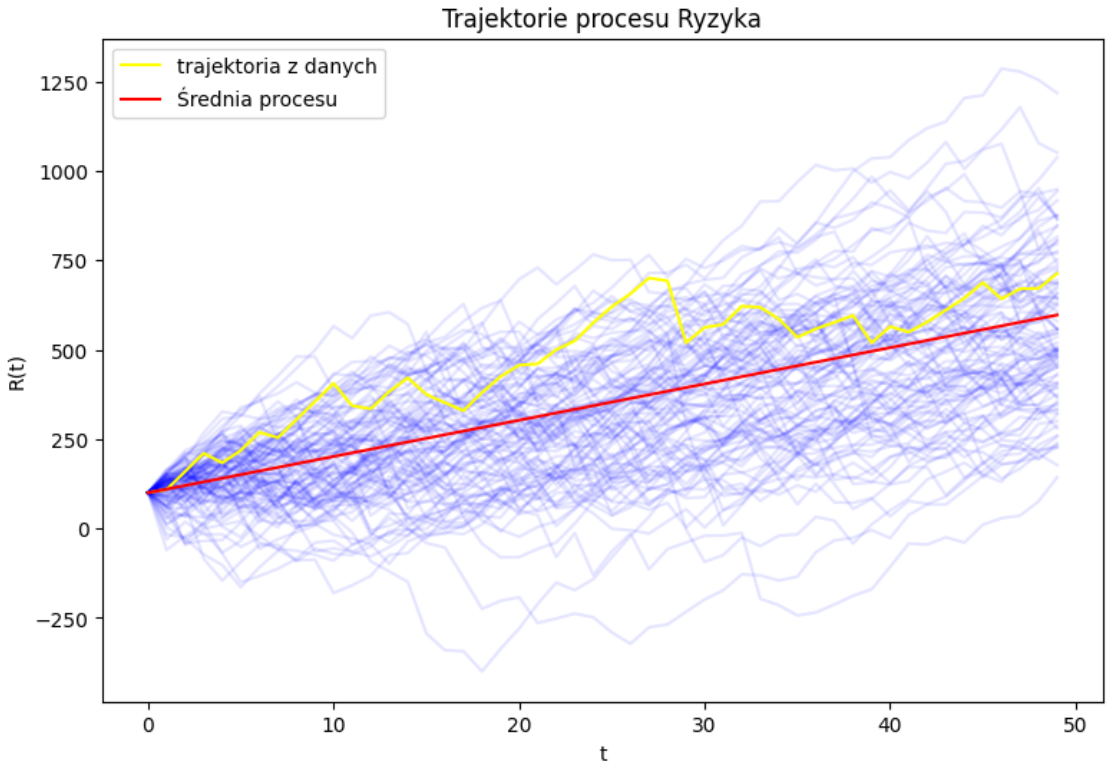
[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import expon
```

```
u_values = 100
c_values = 50
lambda_ = 10 / 4
eta = 1 / 20
T = 50
num_simulations = 1000
theta = 0.2
h = 1


def t_poisson(lmbda, T):
    t = 0
    events = []
    while t < T:
        t += np.random.exponential(1 / lmbda)
        if t < T:
            events.append(t)
    return np.array(events)


def proces_ryzyka(u, T, h, theta, lambda1):
    t = np.arange(0, T, h)
    p = t_poisson(lambda1, T)
    R = np.zeros(len(t))
    x = expon.rvs(scale=1 / eta, size=len(p))
    c = lambda t: (1 + theta) * lambda1 * t * np.mean(x)
    for i in range(len(t)):
        claims_up_to_t = x[p <= t[i]]
        R[i] = u + c(t[i]) - np.sum(claims_up_to_t)
    return R, t

# Function to calculate ruin
def ruina(start_kapital, przychod, T, lamb, lamb_skok):
    trajektoria = t_poisson(lamb, T)
    skoki = expon.rvs(scale=1 / lamb_skok, size=len(trajektoria))
    kapital_list = [start_kapital]
    spadek = 0

    for i in range(1, len(trajektoria)):
        t = trajektoria[i]
        spadek += skoki[i - 1]
        kapital = start_kapital + przychod * t - spadek
        if kapital <= 0:
            kapital_list.append(kapital)
            return True
        kapital_list.append(kapital)
```

```
    return False

def prawdopodobienstwo_ruiny(start_kapital, przychod, T, lamb, lamb_skok,␣
 ↪liczba_symulacji=100):
    ruiny = 0
    for _ in range(liczba_symulacji):
        if ruina(start_kapital, przychod, T, lamb, lamb_skok):
            ruiny += 1
    return ruiny / liczba_symulacji

u = u_values
R, t = proces_ryzyka(u, T, h, theta, lambda_)


plt.figure(figsize=(9, 6))
plt.plot(t, R, color='blue')
plt.xlabel('t')
plt.ylabel('R(t)')
plt.title("Proces Ryzyka z wyestymowaną lambdą i parametrami z danych")
plt.legend(['R(t)'])
plt.show()


plt.figure(figsize=(9, 6))

for _ in range(100):
    R, t = proces_ryzyka(u, T, h, theta, lambda_)
    plt.plot(t, R, color='blue', alpha=0.1)

trajectory, _ = proces_ryzyka(u, T, h, theta, lambda_)
t1 = np.arange(0, T, h)
plt.plot(t1, trajectory, label="trajektoria z danych", color='yellow')

xi = expon.rvs(scale=1 / eta, size=1000)
plt.plot(t1, u + theta * np.mean(xi) * lambda_ * t1, color='red',␣
 ↪label="Średnia procesu")

plt.xlabel('t')
plt.ylabel('R(t)')
plt.title('Trajektorie procesu Ryzyka')
plt.legend()
plt.show()


start_kapital = 100
przychod = 50
lamb = 10 / 4
```

```
lamb_skok = 1 / 20
T = 50
liczba_symulacji = 1000
```





CZAS RUINY

[ ]:
```python
def simulate_poisson(lamb, T):
    """
    Symuluje proces Poissona o intensywności lamb do czasu T.

    Parametry:
    lamb : float
        Intensywność procesu Poissona.
    T : float
        Czas maksymalny, do którego trwa symulacja.

    Zwraca:
    list
        Lista czasów wystąpienia zdarzeń w procesie Poissona do czasu T.
    """
    t = 0
    trajectory = [t]
    while t <= T:
        delta_t = np.random.exponential(1 / lamb)
        t += delta_t
        if t > T:
            break
```

```
        trajectory.append(t)
    return trajectory


def simulate_compound_poisson(lamb, lamb_jump, T):
    """
    Symuluje złożony proces Poissona z intensywnością lamb oraz skokami o␣
 ↪intensywności lamb_jump do czasu T.

    Parametry:
    lamb : float
        Intensywność procesu Poissona.
    lamb_jump : float
        Intensywność skoków w procesie.
    T : float
        Czas maksymalny, do którego trwa symulacja.

    Zwraca:
    tuple
        Krotka zawierająca listę czasów zdarzeń w procesie Poissona oraz listę␣
 ↪wartości skoków.
    """
    trajectory = simulate_poisson(lamb, T)
    n = len(trajectory)
    jumps = np.random.exponential(1 / lamb_jump, size=n)
    return trajectory, jumps


def simulate_risk_process(initial_capital, income_rate, T, lamb, lamb_jump):
    """
    Symuluje proces ryzyka, uwzględniając proces złożony Poissona oraz␣
 ↪kumulacyjne straty, sprawdzając warunek ruiny.

    Parametry:
    initial_capital : float
        Początkowy kapitał.
    income_rate : float
        Stopa przychodów na jednostkę czasu.
    T : float
        Czas maksymalny, do którego trwa symulacja.
    lamb : float
        Intensywność procesu Poissona.
    lamb_jump : float
        Intensywność skoków w procesie Poissona.

    Zwraca:
    tuple
```

```
        Krotka zawierająca wartość logiczną informującą o ruinie, listę␣
↪kapitałów w kolejnych momentach czasowych oraz listę czasów zdarzeń w␣
↪procesie Poissona do chwili ruiny lub końca symulacji.
    """
    trajectory, jumps = simulate_compound_poisson(lamb, lamb_jump, T)
    capital_list = [initial_capital]
    cumulative_loss = 0

    for i in range(1, len(trajectory)):
        t = trajectory[i]
        cumulative_loss += jumps[i-1]
        capital = initial_capital + income_rate * t - cumulative_loss
        if capital <= 0:
            capital_list.append(capital)
            return True, capital_list, trajectory[:len(capital_list)]
        capital_list.append(capital)

    return False, capital_list, trajectory[:len(capital_list)]


def calculate_ruin_probability(initial_capital, income_rate, T, lamb,␣
↪lamb_jump, simulations=1000):
    """
    Oblicza empiryczne prawdopodobieństwo ruiny kapitału przy użyciu wielu␣
↪symulacji procesu ryzyka.

    Parametry:
    initial_capital : float
        Początkowy kapitał.
    income_rate : float
        Stopa przychodów na jednostkę czasu.
    T : float
        Czas maksymalny, do którego trwa symulacja.
    lamb : float
        Intensywność procesu Poissona.
    lamb_jump : float
        Intensywność skoków w procesie Poissona.
    simulations : int, opcjonalny
        Liczba symulacji do wykonania, domyślnie 1000.

    Zwraca:
    float
        Empirycznie obliczone prawdopodobieństwo ruiny kapitału.
    """
    ruin_count = 0
    for _ in range(simulations):
```

```
        is_ruin, _, _ = simulate_risk_process(initial_capital, income_rate, T,␣
↪lamb, lamb_jump)
        if is_ruin:
            ruin_count += 1
    return ruin_count / simulations
```

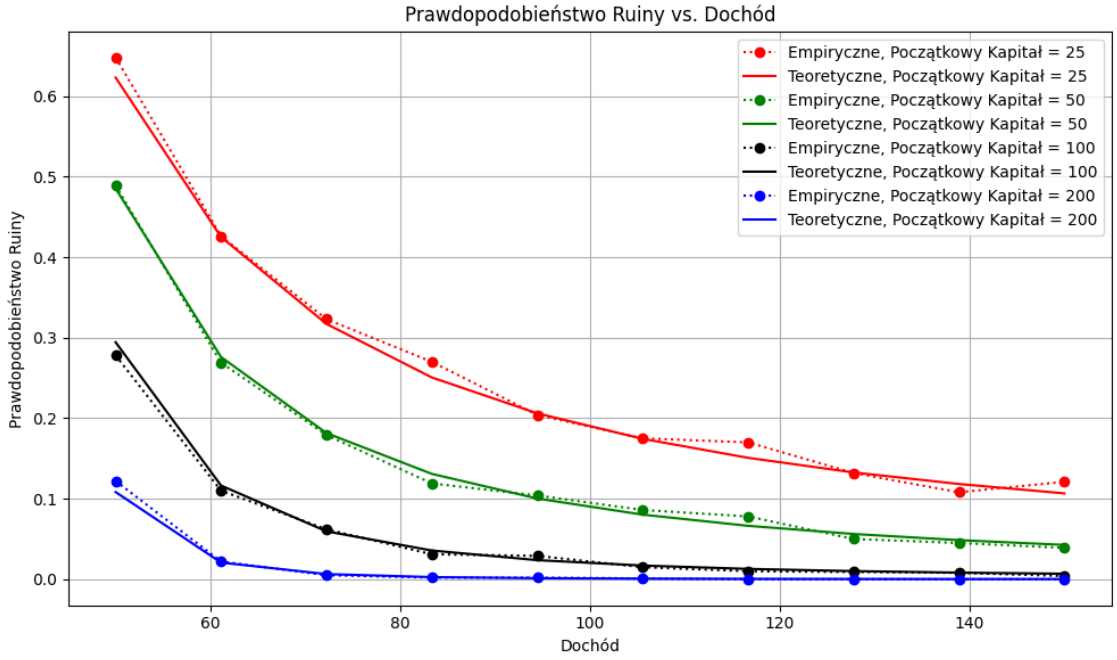
Ten kod jest odpowiedzialny za symulację procesów ryzyka oraz generowanie wykresów porównujących empiryczne i teoretyczne prawdopodobieństwa ruiny w zależności od dochodu oraz początkowego kapitału.

6.1 Wykres Prawdopodobieństwa Ruiny w Zależności od Przychodu

Wykres porównuje teoretyczne i empiryczne prawdopodobieństwo ruiny w zależności od przychodu przy ustalonym początkowym kapitale (25, 50, 100, 200)

[ ]:
```
T = 100
lambda_ = 2
lambda_jump = 1 / 20


initial_capital_values = np.array([25, 50, 100, 200])
income_values = np.linspace(50, 150, 10)

eta = 1 / lambda_jump
start_capital_matrix, income_matrix = np.meshgrid(initial_capital_values,␣
↪income_values, indexing='ij')
probabilities_matrix = np.exp(-(1 / eta - lambda_ / income_matrix) *␣
↪start_capital_matrix) * (eta * lambda_ / income_matrix)
theoretical_ruin_probabilities = probabilities_matrix.tolist()

empirical_ruin_probabilities = np.
↪array([[calculate_ruin_probability(start_capital, income, T, lambda_,␣
↪lambda_jump, simulations=1000)
                                          for income in income_values]
                                         for start_capital in␣
↪initial_capital_values])

plt.figure(figsize=(10, 6))
colors = ['red', 'green', 'black', 'blue']
labels = ['Początkowy Kapitał = 25', 'Początkowy Kapitał = 50', 'Początkowy␣
↪Kapitał = 100', 'Początkowy Kapitał = 200']

for i in range(len(initial_capital_values)):
    plt.plot(income_values, empirical_ruin_probabilities[i], "o:",␣
↪color=colors[i], label=f"Empiryczne, {labels[i]}")
    plt.plot(income_values, theoretical_ruin_probabilities[i], color=colors[i],␣
↪linestyle='-', label=f"Teoretyczne, {labels[i]}")
```

```
plt.xlabel("Dochód")
plt.ylabel("Prawdopodobieństwo Ruiny")
plt.title("Prawdopodobieństwo Ruiny vs. Dochód")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```



Prawdopodobieństwo Ruiny vs. Dochód

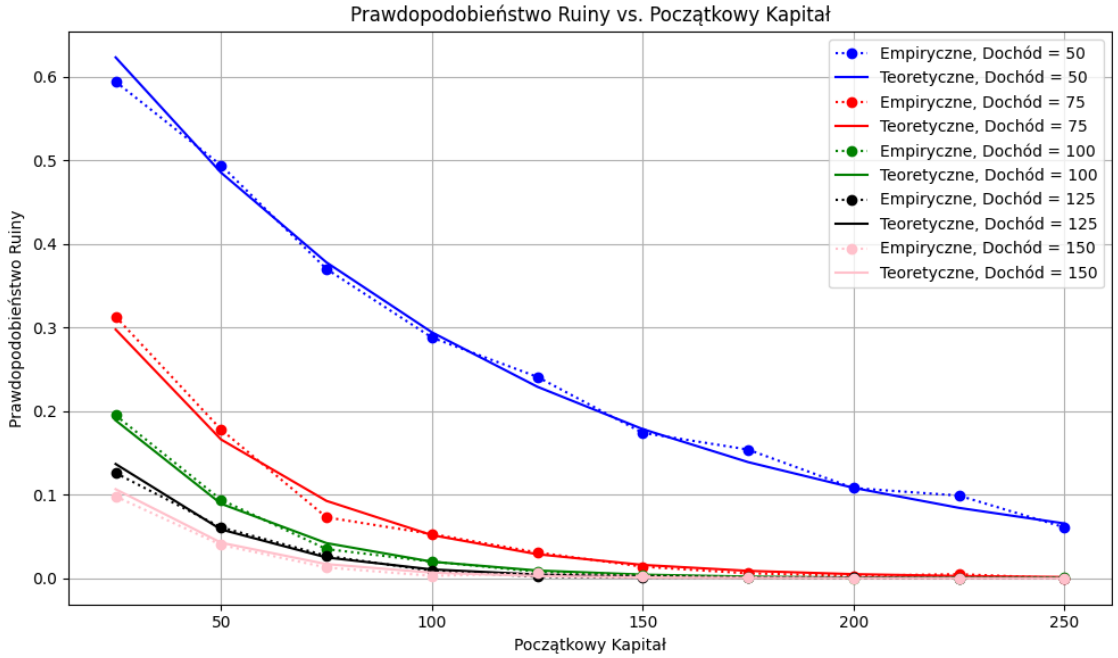
Wykres jasno pokazuje, że zwiększenie dochodu oraz początkowego kapitału znacząco redukuje prawdopodobieństwo ruiny. Wyniki empiryczne są w dobrej zgodności z wynikami teoretycznymi, co potwierdza, że użyty model matematyczny jest trafny i może być używany do przewidywania ryzyka ruiny w praktycznych scenariuszach.

6.2 Wykres Prawdopodobieństwa Ruiny w Zależności od Początkowego Kapitału

Na tym wykresie przedstawiono porównanie teoretycznego i empirycznego prawdopodobieństwa ruiny dla różnych wartości przychodów. Dla każdej wartości przychodu (50, 75, 100, 125, 150)

[ ]:
```
T = 100
lamb = 2
lamb_jump = 1 / 20
eta = 1 / lamb_jump
```

```
income_values = np.arange(50, 151, 25)
initial_capital_values = np.linspace(25, 250, 10)

initial_capital_matrix, income_matrix = np.meshgrid(initial_capital_values,␣
 ↪income_values, indexing='ij')
probabilities_matrix = np.exp(-(1 / eta - lamb / income_matrix) *␣
 ↪initial_capital_matrix) * (eta * lamb / income_matrix)
theoretical_ruin_probabilities = probabilities_matrix.T.tolist()


empirical_ruin_probabilities = np.
 ↪array([[calculate_ruin_probability(initial_capital, income, T, lamb,␣
 ↪lamb_jump, simulations=1000)
                                          for initial_capital in␣
 ↪initial_capital_values]
                                         for income in income_values])


plt.figure(figsize=(10, 6))
colors = ['blue','red', 'green', 'black', 'pink']
labels = ['Dochód = 50', 'Dochód = 75', 'Dochód = 100', 'Dochód = 125', 'Dochód␣
 ↪= 150']

for i in range(len(income_values)):
    plt.plot(initial_capital_values, empirical_ruin_probabilities[i], "o:",␣
 ↪color=colors[i], label=f"Empiryczne, {labels[i]}")
    plt.plot(initial_capital_values, theoretical_ruin_probabilities[i],␣
 ↪color=colors[i], linestyle='-', label=f"Teoretyczne, {labels[i]}")

plt.xlabel("Początkowy Kapitał")
plt.ylabel("Prawdopodobieństwo Ruiny")
plt.title("Prawdopodobieństwo Ruiny vs. Początkowy Kapitał")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```



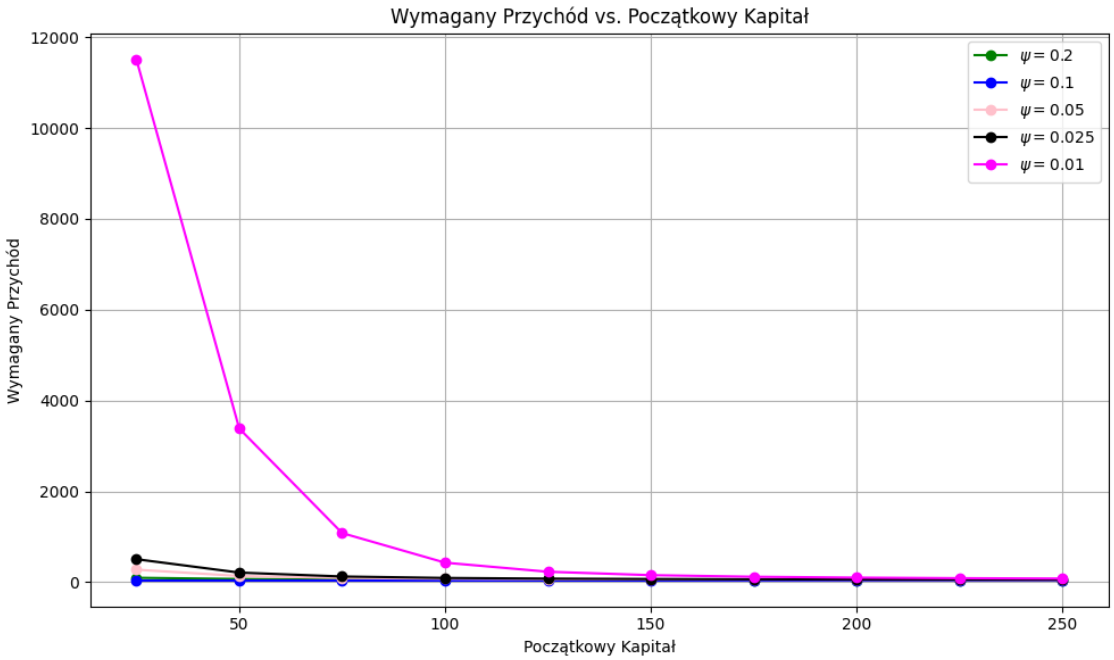
Wykres Prawdopodobieństwa Ruiny w Zależności od Początkowego Kapitału

Wykres skutecznie pokazuje, że wyższy początkowy kapitał i wyższe poziomy dochodów zmniejszają prawdopodobieństwo finansowej ruiny. Zarówno dane empiryczne, jak i teoretyczne są ze sobą zgodne, co potwierdza użyte modele teoretyczne w przewidywaniu prawdopodobieństwa ruiny finansowej.

6.3 Wykres Zależności Potrzebnego Przychodu od Początkowego Kapitału

[ ]:
```
def przychod_odw(start_kapital, lamb, psi, eta):
    return lamb * start_kapital / (lambertw((start_kapital * psi * np.
 ↪exp(start_kapital / eta)) / eta)).real


lamb = 2
eta = 1 / (1 / 20)
psis = np.array([0.2, 01., 0.05, 0.025, 0.001])
start_kapital_values = np.linspace(25, 250, 10)


przychod_odw_list = [[przychod_odw(start_kapital, lamb, psi, eta) for␣
 ↪start_kapital in start_kapital_values] for psi in psis]


plt.figure(figsize=(10, 6))
colors = ['green', 'blue', 'pink', 'black', 'magenta']
```

```
labels = ['$\\psi = 0.2$', '$\\psi = 0.1$', '$\\psi = 0.05$', '$\\psi = 0.
 ↪025$', '$\\psi = 0.01$']

for i in range(len(psis)):
    plt.plot(start_kapital_values, przychod_odw_list[i], "o-", color=colors[i],␣
 ↪label=labels[i])

plt.xlabel("Początkowy Kapitał")
plt.ylabel("Wymagany Przychód")
plt.title("Wymagany Przychód vs. Początkowy Kapitał")
plt.legend()
plt.grid(True)
plt.tight_layout()
```



Wykres pokazuje, że im wyższy kapitał początkowy, tym mniejszy wymagany przychód niezależnie od wartości parametru ψ. Różne wartości $ $ wpływają na wysokość wymaganego przychodu, zwłaszcza przy niskim kapitale początkowym. Dla wyższych wartości kapitału początkowego, wymagany przychód stabilizuje się i różnice między różnymi stają się mniej znaczące.

PODSUMOWANIE Analiza procesu ruiny w modelu Craméra-Lundberga, jak przedstawiono w tym projekcie, dostarcza solidnych podstaw do rozumienia i zarządzania ryzykiem finansowym. Zastosowanie teoretycznych wzorów oraz symulacji komputerowych stanowi wartościowe narzędzie dla praktyków i badaczy zajmujących się analizą ryzyka w różnych dziedzinach, w tym w ubezpieczeniach i finansach.

7 zadanie 6

7.1 Prawa arcusa sinusa

Ciągła zmienna losowa X ma rozkład arcusa sinusa ($X \sim$ Arcsine), gdy jej funkcja gęstości prawdopodobieństwa p_X ma postać

$$p_X(x) = \frac{1}{\pi\sqrt{x(1-x)}} \quad x \in (0,1),$$

Dystrybuanta tej zmiennej jest wtedy równa

$$F_X(x) = \begin{cases} 0, & x < 0, \\ \frac{2}{\pi}\arcsin\sqrt{x}, & x \in [0,1], \\ 1, & x > 1. \end{cases}$$

Prawa arcusa sinusa dla procesu Wienera W_t brzmią następująco.

Pierwsze prawo arcusa sinusa

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

gdzie λ to miara Lebesgue'a. Oznacza to, że czas spędzony przez proces Wienera powyżej osi OX na odcinku $[0,1]$ ma rozkład arcusa sinusa.

Drugie prawo arcusa sinusa

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

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

Trzecie prawo arcusa sinusa Niech M będzie liczbą spełniającą

$$W_M = \sup\{W_t | t \in [0,1]\}.$$

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

Zweryfikuj symulacyjnie prawa arcusa sinusa dla procesu Wienera. W tym celu użyj porównania histogramów oraz dystrybuant empirycznych wysymulowanych próbek z teoretycznymi wartościami.

Zaczynamy od deklaracji wszystkich parametrów do symulacji oraz funkcji związanych z rozkładem arcusa sinusa.

[12]:
```
from scipy.stats import gaussian_kde
import seaborn as sns
```

[13]:
```
# Define the PDF of the arcsine distribution
def arcsine_pdf(x):
```

```
    return 1 / (np.pi * np.sqrt(x * (1 - x)))

# Define the CDF of the arcsine distribution
def arcsine_cdf(x):
    return (2 / np.pi) * np.arcsin(np.sqrt(x))

# Number of samples
n = 10000

# Simulation parameters
num_steps = 1000
T = 1.0
dt = T / num_steps
time = np.linspace(0, T, num_steps + 1)

# Arrays to store simulation results
arcssim1 = np.zeros(n)
arcssim2 = np.zeros(n)
arcssim3 = np.zeros(n)
```

Wywołujemy proces Wienera, oraz zapisujemy wszystkie dane potrzebne do zweryfikowania symulacyjnie trzech praw arcusa sinusa. Informacje jakie chcemy uzyskać z naszej symulacji to: Czas spędzony nad osią X, Ostatni moment uderzenia w oś OX, Ostatni moment osiągnięcia supremum. Dane te zapisujemy do list, aby później zweryfikować ich rozkład.

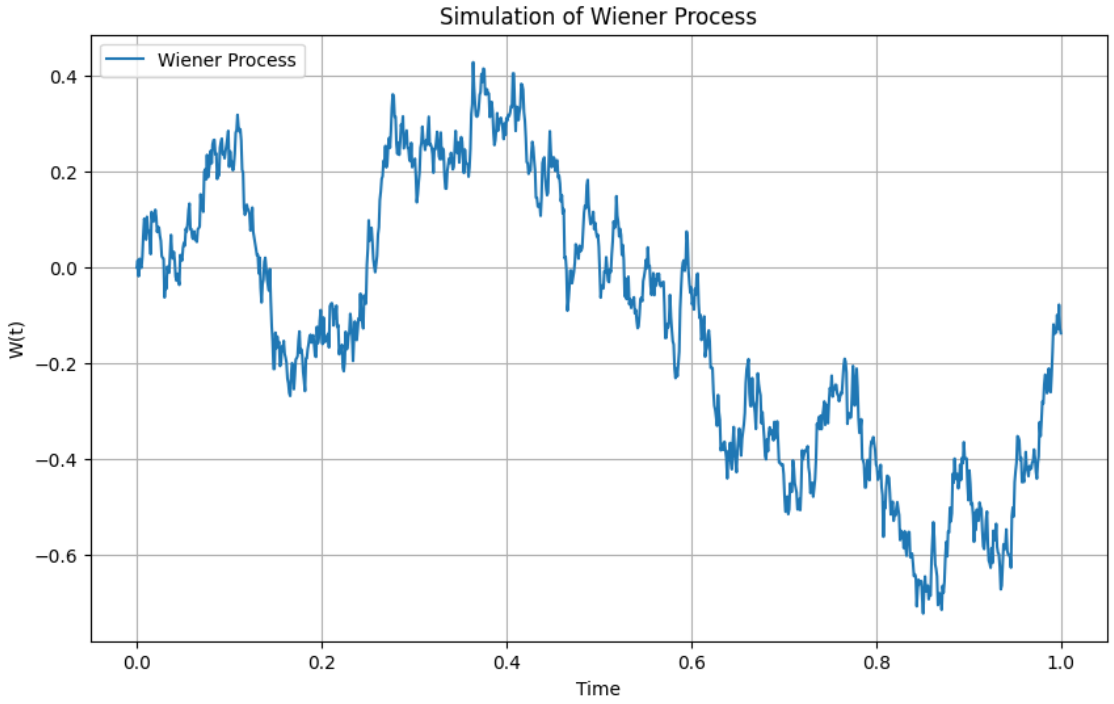
[14]:
```
for i in range(n):
    W = np.zeros(num_steps + 1)
    last_sign_change_time = 0

    for j in range(1, num_steps + 1):
        W[j] = W[j-1] + np.sqrt(dt) * np.random.normal()

    for j in range(num_steps, 0, -1):
        if (W[j-1] > 0 and W[j] < 0) or (W[j-1] < 0 and W[j] > 0):
            last_sign_change_time = time[j]
            break

    time_of_max = time[np.argmax(W)]

    count = np.sum(W > 0)
    arcssim1[i] = count / len(W)
    arcssim2[i] = last_sign_change_time
    arcssim3[i] = time_of_max
```

Reszta kodu odpowiada za poprawne opisanie, wyświetlenie i wywołanie wykresów. Za dziedzine funkcji gęstości oraz dystrybuanty przyjęto wektor x, aby uniknąć dzielenia przez 0. Warto zaznaczyć, że nasza funkcja gęstości ma dziedzine D = (0, 1), zatem jądrowy estymator gęstości również powinien mieć taką dziedzine.

[15]:
```
# Theoretical Arcsine Distribution
x = np.linspace(0.0001, 0.9999, 1000)
arc_pdf = arcsine_pdf(x)
arc_cdf = arcsine_cdf(x)

# Function to compute boundary-corrected KDE
def boundary_corrected_kde(data, x_grid, bandwidth=0.05):
    kde = gaussian_kde(data, bw_method=bandwidth)
    kde_values = kde(x_grid)
    kde_values[x_grid < 0] = 0
    kde_values[x_grid > 1] = 0
    return kde_values

# Grid for KDE
x_grid = np.linspace(0, 1, 1000)

# Compute boundary-corrected KDEs
kde_sim1 = boundary_corrected_kde(arcssim1, x_grid)
kde_sim2 = boundary_corrected_kde(arcssim2, x_grid)
kde_sim3 = boundary_corrected_kde(arcssim3, x_grid)
```

Na poniższym wykresie przedstawiono przykład procesu Weinera. (poniżej pierwszy wykres)

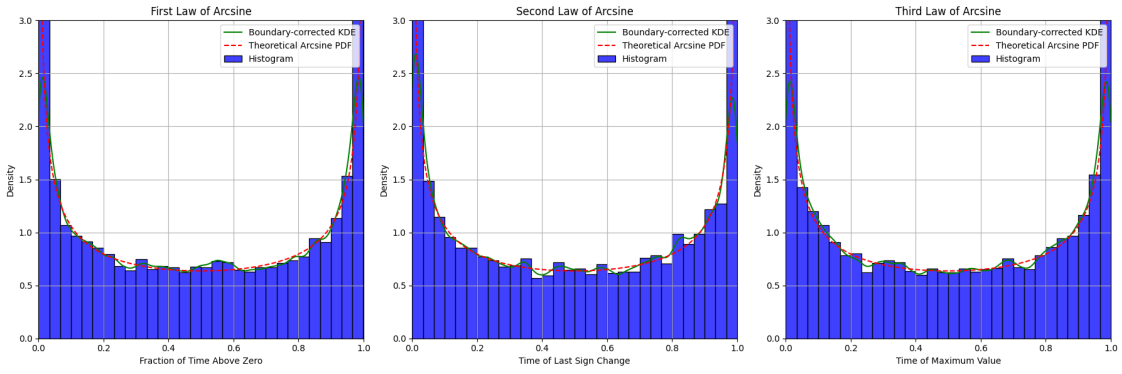
[16]:
```
# Plot the Wiener process (last simulated)
plt.figure(figsize=(10, 6))
plt.plot(time, W, label='Wiener Process')
plt.title('Simulation of Wiener Process')
plt.xlabel('Time')
plt.ylabel('W(t)')
plt.legend(loc='upper left')
plt.grid(True)
plt.show()
```



Na poniższych wykresach widać histogramy, jądrowe estymatory gęstości oraz teoretyczne gęstości. Można zauważyć, że wykresy się pokrywają, co potwierdza prawdziwość trzech praw arcusa sinusa (tutaj wrzuć te 3 wykresy obok siebie co wszystkie wyglądają jak U)

[17]:
```
# Plot the PDFs
plt.figure(figsize=(18, 6))

plt.subplot(1, 3, 1)
sns.histplot(arcssim1, bins=30, kde=False, stat='density', label='Histogram',␣
 ↪color='blue')
plt.plot(x_grid, kde_sim1, label='Boundary-corrected KDE', linestyle='-',␣
 ↪color='green')
plt.plot(x, arc_pdf, label='Theoretical Arcsine PDF', linestyle='--',␣
 ↪color='red')
plt.title('First Law of Arcsine')
plt.xlabel('Fraction of Time Above Zero')
plt.ylabel('Density')
plt.xlim(0, 1)
plt.ylim(0, 3)
plt.legend(loc='upper right')
plt.grid(True)

plt.subplot(1, 3, 2)
```

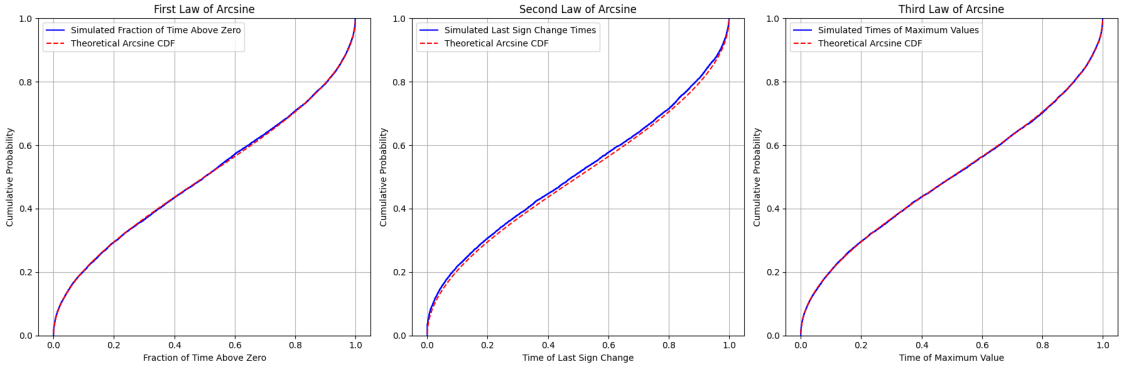
```
sns.histplot(arcssim2, bins=30, kde=False, stat='density', label='Histogram',␣
 ↪color='blue')
plt.plot(x_grid, kde_sim2, label='Boundary-corrected KDE', linestyle='-',␣
 ↪color='green')
plt.plot(x, arc_pdf, label='Theoretical Arcsine PDF', linestyle='--',␣
 ↪color='red')
plt.title('Second Law of Arcsine')
plt.xlabel('Time of Last Sign Change')
plt.ylabel('Density')
plt.xlim(0, 1)
plt.ylim(0, 3)
plt.legend(loc='upper right')
plt.grid(True)

plt.subplot(1, 3, 3)
sns.histplot(arcssim3, bins=30, kde=False, stat='density', label='Histogram',␣
 ↪color='blue')
plt.plot(x_grid, kde_sim3, label='Boundary-corrected KDE', linestyle='-',␣
 ↪color='green')
plt.plot(x, arc_pdf, label='Theoretical Arcsine PDF', linestyle='--',␣
 ↪color='red')
plt.title('Third Law of Arcsine')
plt.xlabel('Time of Maximum Value')
plt.ylabel('Density')
plt.xlim(0, 1)
plt.ylim(0, 3)
plt.legend(loc='upper right')
plt.grid(True)

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



Podobnie jak wyżej, przedstawiono tutaj dystrybuanty empiryczne, oraz teoretyczne dla każdego z praw arcusa sinusa. Tutaj również nasze wykresy się pokrywają, z czego można wywnioskować

prawdziwość praw. (tutaj te 3 dystrybuanty)

[18]:
```
# Plot the CDFs
plt.figure(figsize=(18, 6))

plt.subplot(1, 3, 1)
sns.ecdfplot(arcssim1, label='Simulated Fraction of Time Above Zero',␣
 ↪color='blue')
plt.plot(x, arc_cdf, label='Theoretical Arcsine CDF', linestyle='--',␣
 ↪color='red')
plt.title('First Law of Arcsine')
plt.xlabel('Fraction of Time Above Zero')
plt.ylabel('Cumulative Probability')
plt.legend(loc='upper left')
plt.grid(True)

plt.subplot(1, 3, 2)
sns.ecdfplot(arcssim2, label='Simulated Last Sign Change Times', color='blue')
plt.plot(x * T, arc_cdf, label='Theoretical Arcsine CDF', linestyle='--',␣
 ↪color='red')
plt.title('Second Law of Arcsine')
plt.xlabel('Time of Last Sign Change')
plt.ylabel('Cumulative Probability')
plt.legend(loc='upper left')
plt.grid(True)

plt.subplot(1, 3, 3)
sns.ecdfplot(arcssim3, label='Simulated Times of Maximum Values', color='blue')
plt.plot(x * T, arc_cdf, label='Theoretical Arcsine CDF', linestyle='--',␣
 ↪color='red')
plt.title('Third Law of Arcsine')
plt.xlabel('Time of Maximum Value')
plt.ylabel('Cumulative Probability')
plt.legend(loc='upper left')
plt.grid(True)

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



7.2 Wnioski

W symulacji procesu Wienera zastosowaliśmy pierwsze, drugie i trzecie prawo arcusa sinusa, aby analizować wyniki. Metoda estymacji gęstości jądrowej pozwoliła nam na uzyskanie gładkich wykresów rozkładu danych, co ułatwiło porównanie z teoretycznym rozkładem arcusa sinusa. Wyniki symulacji wykazały zgodność z teoretycznymi rozkładami, co potwierdza prawidłowość przeprowadzonych symulacji. Pierwsze prawo arcusa sinusa pokazało, że ułamek czasu, w którym proces Wienera jest powyżej zera, jest zgodny z przewidywaniami teoretycznymi. Drugie prawo wskazało, że czas ostatniej zmiany znaku również odpowiada teoretycznemu rozkładowi. Trzecie prawo arcusa sinusa potwierdziło, że czas osiągnięcia maksymalnej wartości przez proces Wienera jest zgodny z oczekiwaniami teoretycznymi.