Metody Obliczeniowe w Nauce i Technice
Natalia Bratek, 08.06.2024
Laboratorium 11

Całkowanie Monte Carlo

1. Treści zadań

Tematem zadania będzie obliczanie metodą Monte Carlo całki funkcji: 1) x^2 + x + 1, 2) sqrt(1-x^2) oraz 3) 1/sqrt(*x*) w przedziale (0,1). Proszę dla tych funkcji:

1. Napisać funkcję liczącą całkę metodą "hit-and-miss". Czy będzie ona dobrze działać dla funkcji 1/sqrt(x)?
2. Policzyć całkę przy użyciu napisanej funkcji. Jak zmienia się błąd wraz ze wzrostem liczby prób?
3. Policzyć wartość całki korzystając ze wzoru prostokątów dla dokładności (1e-3, 1e-4, 1e-5 i 1e-6). Porównać czas obliczenia całki metodą Monte Carlo i przy pomocy wzoru prostokątów dla tej samej dokładności, narysować wykres. Zinterpretować wyniki.

2. Rozwiązania

Metoda "hit-and-miss" polega na losowym generowaniu punktów w obrębie określonego obszaru i liczeniu, ile z tych punktów "trafia" pod wykres funkcji, a ile "mija" się z nim.

Najpierw definiujemy prostokątny obszar, który obejmie wykres funkcji f(x) w danym przedziale całkowania [a,b]. Wysokość prostokąta ustala się na podstawie maksymalnej wartości funkcji M w przedziale. Następnie generujemy losowo dużą liczbę punktów wewnątrz tego prostokąta. Oznacza to losowanie wartości x z równomiernego rozkładu w przedziale [a,b] oraz wartości y z równomiernego rozkładu w przedziale [0,M]. Dla każdego wygenerowanego punktu (x,y) sprawdzamy, czy znajduje się on poniżej krzywej funkcji, czyli czy y≤f(x). Jeśli tak, punkt jest liczbony jako "trafienie". Stosunek liczby trafień do liczby wszystkich wygenerowanych punktów, pomnożony przez obszar prostokąta, daje przybliżoną wartość całki. Matematycznie, jeśli N to liczba wszystkich punktów, a H to liczba trafień, wtedy przybliżona wartość całki wynosi:

$$\frac{H}{N} * (b - a) * M$$

2.1 Zadanie pierwsze

```python
import numpy as np

def monte_carlo_hit_miss(f, a, b, M, num_points):
    hits = 0
    for _ in range(num_points):
        x = np.random.uniform(a, b)
        y = np.random.uniform(0, M)
        if y <= f(x):
            hits += 1
    return (hits / num_points) * (b - a) * M

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

def g(x):
    return np.sqrt(1 - x**2)

def h(x):
    return 1 / np.sqrt(x)

def modified_h(x):
    return 1/np.sqrt(x + 1e-10)

# Obliczanie maksimum funkcji na przedziale (0, 1) dla f i g
M_f = f(1)  # Bo funkcja f(x) jest rosnąca na (0,1)
M_g = 1     # sqrt(1 - x^2) ma maksimum w x=0
M_h = float('inf')  # Wartość nieskończona, co jest problemem dla metody hit-and-miss
# Maksymalna wartość funkcji na przedziale (0, 1)
M_h_mod = h(0)
```

Rys. 1 Implementacja hit-and-miss

Metoda "hit-and-miss" nie będzie skuteczna dla funkcji h(x), ponieważ nie jesteśmy w stanie określić odpowiedniego M - funkcja dąży do nieskończoności w x=0. W . W metodzie "hit-and-miss" potrzebujemy ustalić maksymalną wartość M dla y, aby zdefiniować prostokąt, w którym będą losowane punkty. Jednak w przypadku tej funkcji, nie można wyznaczyć skończonego M, co uniemożliwia prawidłowe zdefiniowanie obszaru do losowania punktów. Kiedy wartość M jest bardzo duża lub nieskończona, większość losowych punktów (x,y) znajduje się poza obszarem pod funkcją, co prowadzi do bardzo niskiej efektywności próbkowania.
Aby uniknąć osobliwości w x=0 zmieniamy funkcję h(x).

2.2 Zadanie drugie

Dokładne wartości całek:

Dla f(x): $\int_0^1 x^2 + x + 1 \, dx = 1.8333$
Dla g(x): $\int_0^1 \sqrt{1 - x^2} \, dx = \frac{\pi}{4} \approx 0.7854$
Dla h(x): $\int_0^1 \frac{1}{\sqrt{x}} \, dx = 2$

```
points = [100, 1000, 10000, 100000, 1000000]

results_f = [monte_carlo_hit_miss(f, 0, 1, M_f, n) for n in points]
results_g = [monte_carlo_hit_miss(g, 0, 1, M_g, n) for n in points]
results_h = [monte_carlo_hit_miss(modified_h, 0, 1, M_h_modified, n) for n in points]

exact_f = 1.8333
exact_g = np.pi / 4
exact_h = 2
errors_f = [abs(result - exact_f) for result in results_f]
errors_g = [abs(result - exact_g) for result in results_g]
errors_h = [abs(result - exact_h) for result in results_h]
errors_f, errors_g, errors_h
```

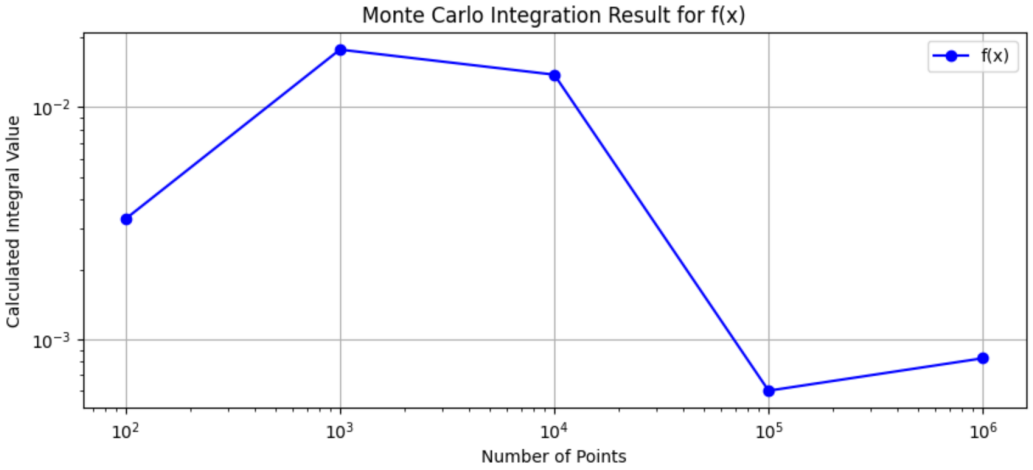
Rys. 2 Błędy

L.punktów	Całka f	Całka g	Całka h
100	0.00329999999999998586	0.0553981633974483	2.0
1000	0.0177000000000005	0.0026018366025517548	2.0
10000	0.0138000000000000257	0.0010981633974482818	2.0
100000	0.0005999999999999339	0.0012581633974483308	1.0
1000000	0.0008280000000000273	0.0003658366025517739	0.10000000000000009

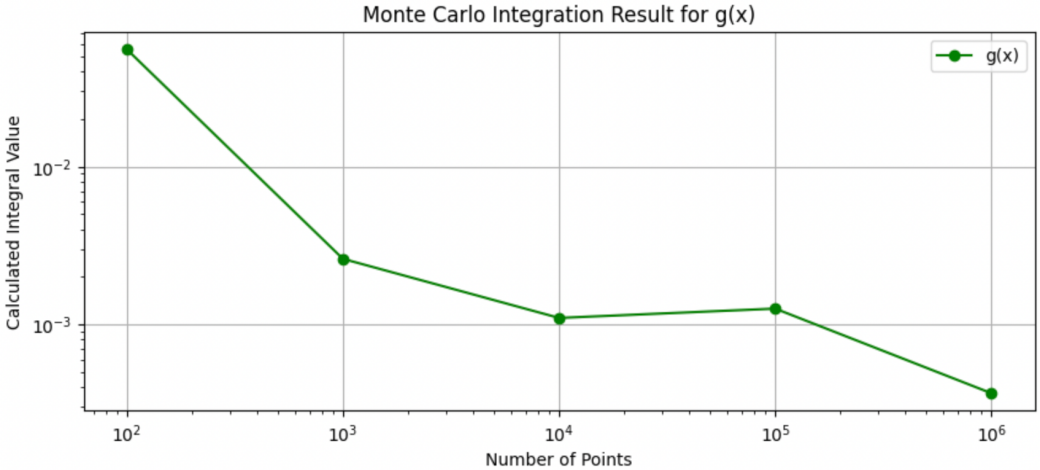
Tab.1 Wyniki

```
# Tworzenie wykresu dla f(x)
plt.figure(figsize=(10, 4))
plt.plot(points, results_f, label='f(x)', marker='o', color='blue')
plt.xlabel('Number of Points')
plt.ylabel('Calculated Integral Value')
plt.title('Monte Carlo Integration Result for f(x)')
plt.xscale('log')
plt.yscale('log')
plt.grid(True)
plt.legend()
plt.show()

# Tworzenie wykresu dla g(x)
plt.figure(figsize=(10, 4))
plt.plot(points, results_g, label='g(x)', marker='o', color='green')
plt.xlabel('Number of Points')
plt.ylabel('Calculated Integral Value')
plt.title('Monte Carlo Integration Result for g(x)')
plt.xscale('log')
plt.yscale('log')
plt.grid(True)
plt.legend()
plt.show()

# Tworzenie wykresu dla h(x)
plt.figure(figsize=(10, 4))
plt.plot(points, results_h, label='h(x)', marker='o', color='red')
plt.xlabel('Number of Points')
plt.ylabel('Calculated Integral Value')
plt.title('Monte Carlo Integration Result for h(x)')
plt.xscale('log')
plt.yscale('log')
plt.grid(True)
plt.legend()
plt.show()
```

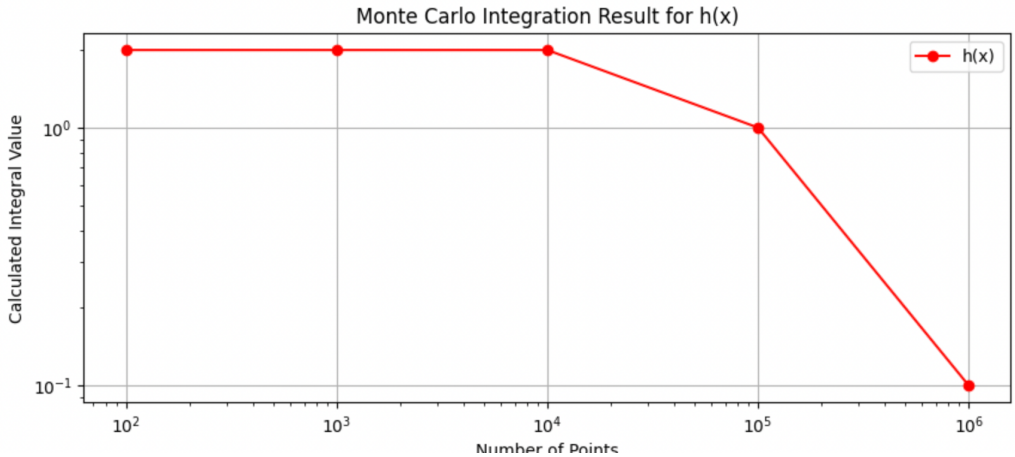
Rys. 3 Wykresy



Rys.4 Wykres dla f(x)



Rys.5 Wykres dla g(x)



Rys.6 Wykres dla h(x)

Wyniki dla f(x) wykazują nieregularności w wynikach. Zmniejszenie błędu może nie być jednolite, co wynika z losowości próbkowania. Błąd najpierw rośnie, a później maleje. Dla g(x), błąd również generalnie maleje, choć z pewnymi nieoczekiwanymi wzrostami, co może wynikać z charakterystyki funkcji lub specyfiki próbkowania. Dla h(x), wyniki są najbardziej niestabilne, co może wynikać z wyzwań w próbkowaniu funkcji mającej wyższe wartości w okolicy x=0 (lub ogólnie problemów z funkcją posiadającą osobliwości).

2.3 Zadanie trzecie

Policzyłam wartości całek korzystając ze wzoru prostokątów dla różnych dokładności.

```
def f(x):
    return x**2 + x + 1

def g(x):
    return np.sqrt(1 - x**2)

def h(x):
    return 1 / np.sqrt(x + 1e-10)

def rectangle_rule(func, a, b, precision):
    n = 1
    current_area = 0
    while True:
        dx = (b - a) / n
        x_midpoints = a + dx / 2 + np.arange(n) * dx
        area = np.sum(func(x_midpoints) * dx)
        if np.abs(area - current_area) < precision:
            break
        current_area = area
        n *= 2
    return area, n

a, b = 0, 1
precisions = [1e-3, 1e-4, 1e-5, 1e-6]
results = {}
for precision in precisions:
    results[f'f, precision {precision}'] = rectangle_rule(f, a, b, precision)
    results[f'g, precision {precision}'] = rectangle_rule(g, a, b, precision)
    results[f'h, precision {precision}'] = rectangle_rule(h, a, b, precision)
results
```

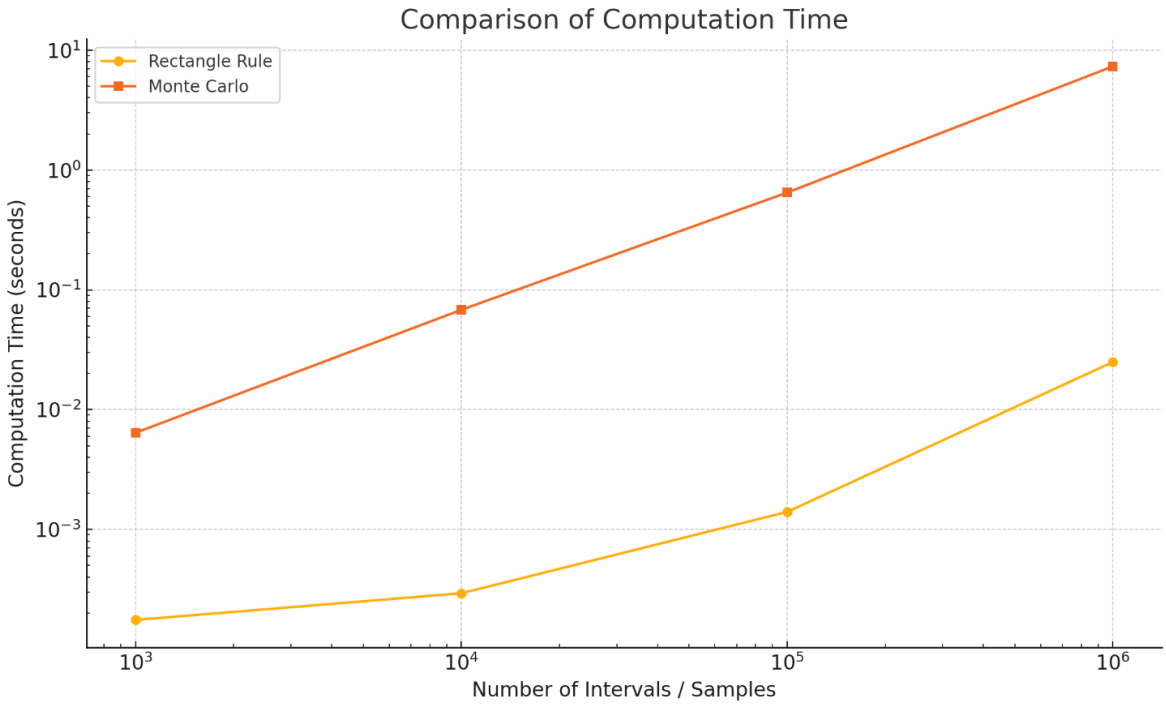
Rys. 7 Metoda prostokątów

Następnie porównałam czas obliczania całek metodą Monte Carlo oraz przy pomocy wzoru prostokątów.

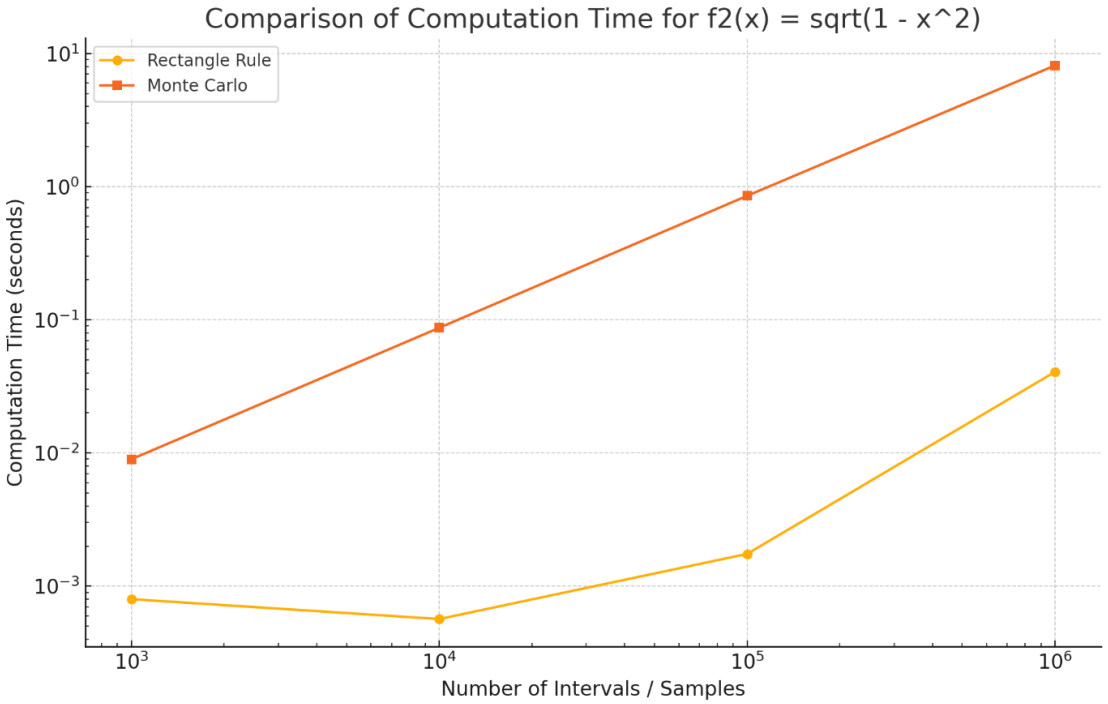
```
def f(x):
    return x**2 + x + 1
def g(x):
    return np.sqrt(1 - x**2)
def h(x):
    return 1 / np.sqrt(x + 1e-10)
def monte_carlo_integration(f, a, b, num_samples):
    x_vals = np.linspace(a, b, num_samples)
    f_vals = f(x_vals)
    max_f = max(f_vals)
    hits = 0
    for _ in range(num_samples):
        x_random = np.random.uniform(a, b)
        y_random = np.random.uniform(0, max_f)
        if y_random <= f(x_random):
            hits += 1
    area = (b - a) * max_f
    integral_estimate = area * (hits / num_samples)
    return integral_estimate
```

```
def rectangle_rule_integration(f, a, b, num_intervals):
    width = (b - a) / num_intervals
    x_vals = np.linspace(a + width/2, b - width/2, num_intervals)
    f_vals = f(x_vals)
    integral_estimate = width * np.sum(f_vals)
    return integral_estimate
num_intervals = [1000, 10000, 100000, 1000000]
num_samples_mc = [1000, 10000, 100000, 1000000]
mc_times = []
rect_times = []
for ni, ns in zip(num_intervals, num_samples_mc):
    start_time = time.time()
    rectangle_rule_integration(f, 0, 1, ni)
    rect_times.append(time.time() - start_time)
    start_time = time.time()
    monte_carlo_integration(f, 0, 1, ns)
    mc_times.append(time.time() - start_time)
plt.figure(figsize=(10, 5))
plt.plot(num_intervals, rect_times, 'o-', label='Rectangle Rule')
plt.plot(num_samples_mc, mc_times, 's-', label='Monte Carlo')
plt.xscale('log')
plt.yscale('log')
plt.xlabel('Number of Intervals / Samples')
plt.ylabel('Computation Time (seconds)')
plt.title('Comparison of Computation Time for f(x)')
plt.legend()
plt.show()
```

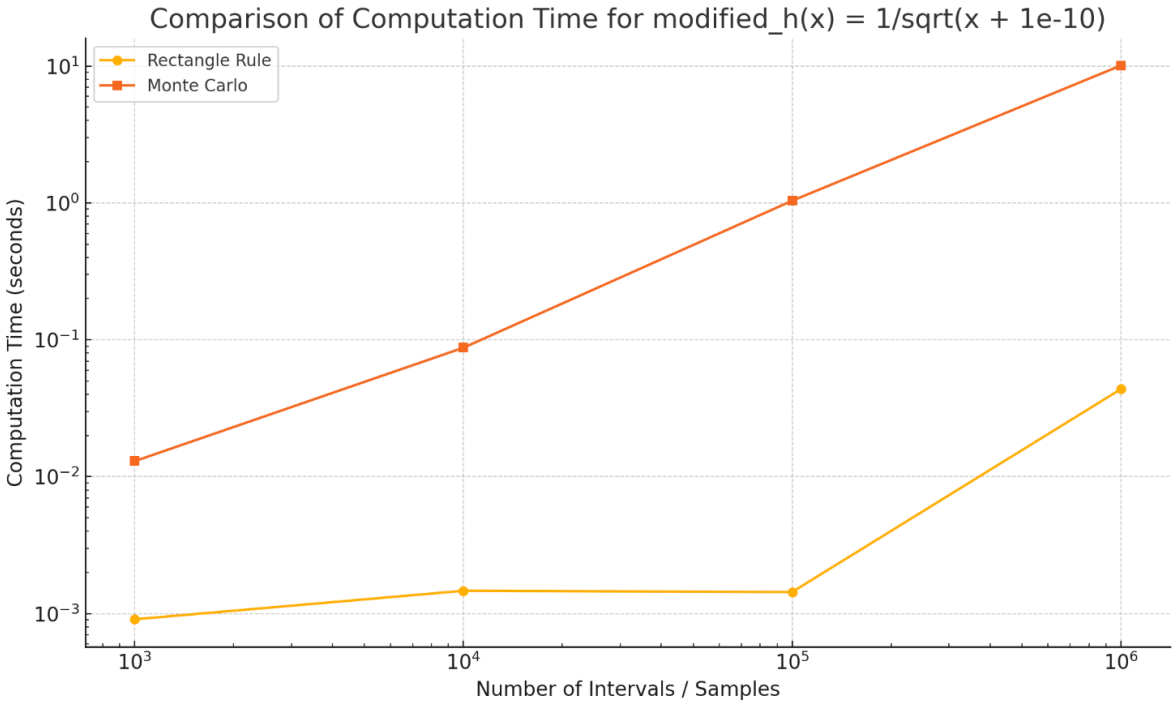
Rys. 8 Obliczanie czasów i rysowanie wykresów



Rys. 9 Wykres dla f(x)



Rys. 10 Wykres dla g(x)



Rys. 11 Wykres dla h(x)

Wnioski:

- Dla f(x) Metoda prostokątów pokazuje znacznie krótsze czasy obliczeń dla mniejszych liczby przedziałów/samples w porównaniu do metody Monte Carlo. W miarę zwiększania liczby prób/interwałów czas obliczeń obu metod rośnie, ale metoda Monte Carlo wydaje się rosnąć szybciej, zwłaszcza przy największej liczbie prób.
- Dla g(x) Podobnie jak w pierwszym przypadku, metoda prostokątów jest szybsza przy mniejszej liczbie prób, ale różnica między metodami zmniejsza się wraz ze wzrostem liczby prób. Obie metody pokazują liniowy wzrost czasu obliczeń, jednak wzrost metody Monte Carlo jest znacznie bardziej wyraźny.
- Dla h(x) Metoda prostokątów również wykazuje szybszy czas obliczeń przy mniejszych liczbach interwałów, ale metoda Monte Carlo ma nieco bardziej stromy wzrost czasu obliczeń w porównaniu do metody prostokątów. Różnica w czasie obliczeń między metodami jest najbardziej widoczna przy wyższych liczbach prób, co wskazuje na to, że dla funkcji z osobliwościami, takimi jak h(x), metoda Monte Carlo może być mniej efektywna.
- Metoda prostokątów zazwyczaj wykazuje lepszą efektywność czasową przy mniejszych liczbach interwałów. Jest to metoda bardziej stabilna i przewidywalna, ponieważ polega na równomiernym podziale przedziału i sumowaniu wartości funkcji.
- Metoda Monte Carlo, która opiera się na losowaniu próbek, wykazuje wolniejsze tempo przyrostu efektywności w miarę zwiększania liczby prób, szczególnie dla funkcji z osobliwościami.

3. Bibliografia

- https://mathworld.wolfram.com/MonteCarloMethod.html
- https://www.taygeta.com/rwalks/node3.html
- https://en.wikipedia.org/wiki/Monte_Carlo_integration