SPRAWOZDANIE Z PRZEDMIOTU PROGRAMOWANIE W JĘZYKU PYTHON

Autorzy:
Justyna Małysa
Maciej Sudoł

Akademia Górniczo-Hutnicza

Kraków 2021

SPIS TREŚCI

1. Temat zadania

Do obliczenia całki:

$$I = \int_{-\pi/2}^{\pi/2} [2 - \exp(\frac{2\ln(2)}{\pi} |x|)]\, dx$$

zastosowano podstawową metodę Monte Carlo, polegającą na losowaniu par liczb o rozkładzie równomiernym w prostokącie x ∈ [-π/2; π/2], y ∈ [0, 1]. Oszacować liczbę losowań N wystarczającą do zapewnienia wariancji rzędu 10^{-3}.

2. Metoda Monte Carlo

W przypadku omawianego projektu realizacja metody Monte Carlo sprowadzała się do obliczenia pola prostokąta, który ograniczał przedział zadanej do obliczenia całki. Następnie należało dla odpowiedniej liczby losowań wylosować pary wartości x i y, gdzie x ∈ [-π/2; π/2] oraz y ∈ [0; 1]. Kolejnym krokiem było obliczenie wartości funkcji podcałkowej dla wylosowanych x-ów i porównanie ich z wartościami wylosowanych y-ów. Jeśli wartość funkcji podcałkowej była większa od wartości y to dana próbka była uznawana za „trafioną". Ostatecznie wartość całki była obliczana ze wzoru:

$$P = P_k * \frac{k}{n}$$

Gdzie:

P – wartość całki

P_k - pole prostokąta ograniczającego przedział

k – liczba losowań trafionych

n – liczba wszystkich losowań

Implementacja omawianego algorytmu w kodzie:

```
def calculate_area(samples):
    lut = FunctionGivenLUT("lut_1e5.bin", -np.pi/2, 1e-5)
    k = 0
    rectangle_area = np.pi * 1
    px = np.random.uniform(low=-np.pi/2, high=np.pi/2, size=samples)
    py = np.random.uniform(low=0, high=1, size=samples)
    f = [lut.get(x) for x in px]
    for i in range(samples):
        if (f[i] >= py[i]):
            k = k+1; # samples hit
    area = rectangle_area*k/samples
    return area
```

Rys. 1. Kod w języku Python do obliczania całki metodą Monte Carlo

Funkcja *calculate_area()* odpowiada za obliczenie wartości całki, czyli de facto pola pod wykresem funkcji podcałkowej. Zmienna *rectangle_area* to wartość pola prostokąta ograniczającego przedział całki. *Px* i *py* to wektory, do których są dodawane wylosowane pary liczb x, y dla liczby losowań, którą reprezentuje zmienna *samples*. Podczas tworzenia projektu zmagano się z dużą ilością czasu potrzebnego do wykonania programu. Dlatego zdecydowano się na jego usprawnienie poprzez dołączenie pliku, w którym są już obliczone (z dokładnością do piątego miejsca po przecinku) wartości funkcji podcałkowej dla różnych x-ów. W tym celu stworzono klasę *FunctionGivenLUT.* Podczas tworzenia obiektu tej klasy wczytywany jest plik, którego zawartość umieszczana jest w tablicy. Takie rozwiązanie sprowadza obliczenie wartości zadanej funkcji do pobrania wartości z tablicy spod odpowiadającego tej wartości indeksu, co znacznie przyspiesza działanie programu.

3. Obliczanie wariancji

Z treści zadania wynika, że liczba losowań ma wystarczyć do zapewnienia wariancji rzędu 10^{-3}. W celu realizacji tego punktu napisano funkcję *calculate_var_worker()*, która zadaną liczbę razy (*count*) oblicza wartość całki w celu obliczenia wariancji (*variance*). W programie zastosowano podział wykonywanych obliczeń na większą liczbę rdzeni procesora w celu przyspieszenia ich wykonania.

Implementacja algorytmu obliczania wariancji w kodzie:

```
def calculate_var_worker(count, samples):
    if sys.platform == "win32":
        import win32api
        win32api.SetConsoleCtrlHandler(None, True)

    area_vec = []
    for i in range (count):
        area_vec.append(calculate_area(samples))
    variance = np.var(area_vec)
    return variance
```

Rys. 2 Kod w języku Python do obliczania wariancji

4. Metoda Bisekcji

Następnie posłużono się metodą Bisekcji w celu znalezienia odpowiedniej liczby losowań do zapewnienia wariancji rzędu 10^{-3}. Aby zrealizować to zadanie napisano funkcję *bisection()*, która jako argumenty przyjmuje zmienne *beginning_interval*, *end_interval*, które określają początek i koniec przedziału do metody Bisekcji oraz zmienną *count*, która tak jak poprzednio odpowiada za liczbę pól, jaką należy obliczyć w celu policzenia wariancji. Cała procedura polega na obliczeniu wariancji dla wartości próbek będącej dokładnie w połowie zadanego przedziału i przypisanie jej do zmiennej *samples*. Następnie sprawdzany jest warunek, czy wariancja jest większa od 10^{-3}. Jeśli warunek jest spełniony w następnym przejściu pętli *while* początek przedziału będzie znajdował się tam gdzie połowa poprzedniego natomiast koniec pozostanie taki sam. Gdy warunek nie zostanie spełniony początek przedziału się nie zmieni a jego koniec będzie tam gdzie połowa poprzedniego. Opisana procedura będzie powtarzać się do momentu aż przedział będzie już niepodzielny, czyli ilość losowań z poprzedniego przejścia pętli będzie równa aktualnej.

Implementacja algorytmu w kodzie:

```
def bisection(beginning_interval, end_interval, count):
  samples = 1
  samples_prev = 0
  while (samples_prev != samples):
    samples_prev = samples
    samples = int((beginning_interval + end_interval)/2)
    tstart = timer()
    variance = calculate_var(count, samples)
    tend = timer()
    print("Samples: {}  Variance: {}  Time: {}".format
        (samples, variance, tend - tstart))
    if (variance > 1e-3):
      beginning_interval = samples
    else:
      end_interval = samples
  return samples
```

Rys. 3 Kod w języku Python do obliczenia liczby losowań zapewniającej wariancję rzędu 10^{-3}

5. Przeprowadzone testy

Dla wyżej opisanego algorytmu przetestowano działanie programu dla przedziału startowego do metody Bisekcji należącego od 0 do 6000 oraz dla 14000 powtórzeń. Otrzymano trzy następujące wartości liczby koniecznych losowań do zapewnienia wariancji rzędu 10^{-3}:

Próba	Liczba losowań	Wariancja
1	2423	0.000993
2	2462	0.000981
3	2436	0.001011

Tabela nr 1 Zestawienie przeprowadzonych testów

Jak można zauważyć z otrzymanych wyników przeprowadzonych testów dla takiej samej liczby powtórzeń w każdym przejściu programu wyniki różnią się wartością 3 i 4 cyfry znaczącej. Wynika to z faktu, że metoda Monte Carlo jest metodą probabilistyczną więc każde uruchomienie programu daje inny wynik. Zwiększając liczbę losowań oraz liczbę prób można usprawnić deterministyczność ostatecznego wyniku. Niemniej jednak z charakteru metody wynika, że zawsze będzie obarczony on pewną niepewnością. Co więcej metoda Monte Carlo zakłada, że wykorzystywany generator liczb losowych będzie generatorem o rozkładzie równomiernym, czego w pełni nie zapewnia wykorzystywany generator pseudolosowy z biblioteki *numpy*.

6. Alternatywne rozwiązanie z zastosowaniem reguły trzech sigm

W celu ulepszenia działania programu postanowiono dla otrzymanej ilości losowań obliczyć kilkakrotnie wariancję w celu wyznaczenia odchylenia standardowego z rozkładu wariancji. Cała procedura ma na celu „przesunięcie" w rozkładzie Gaussa dla wariancji o wartość równą 3*sigma (odchylenie standardowe) w celu stabilizacji otrzymanych wyników. Aby to uczynić stworzono dodatkową funkcję *var_of_var(),* która zadaną liczbę razy (*count_var*) obliczy wariancję dla liczby losowań (samples) otrzymanej z poprzedniej wersji programu. Omawiana funkcja zwraca wartość *three_sigma*. Poszerzenie działa programu wymagało również zmodyfikowania funkcji *bisection()* omawianej powyżej tak, aby przyjmowała ona również jako jeden z argumentów wartość *three_sigma,* która następnie będzie odejmowana od liczby 10^{-3} w warunku *if ().* Dalsza procedura wyznaczania ilości losowań jest analogiczna jak w pierwotnej wersji programu.

Implementacja w kodzie funkcji *var_of_var ()*

```
def var_of_var (count, samples, count_var):
    var_vec = []
    tstart = timer()
    for i in range(count_var):
        var_vec.append(calculate_var(count, samples))
    var_var = np.var(var_vec)
    three_sigma = 3* (np.sqrt(var_var))
    tend = timer()
    print ("Three sigma: {}  Time: {}".format(three_sigma, tend - tstart))
    return three_sigma
```

Rys. 4 Kod w języku Python do obliczenia wariancji wariancji

Dla omawianej wyżej metody również przeprowadzono testy dla takiej samej liczby powtórzeń 14000 (*count*), przy liczbie obliczonych wariancji 100 (*count_var*). Otrzymano następujące wyniki:

Próba	Liczba losowań	Wariancja
1	2531	0.000959
2	2531	0.000966
3	2524	0.000973

Tabela nr 2 Zestawienie przeprowadzonych testów dla metody alternatywnej

Dzięki zaproponowanej modyfikacji uzyskano wyniki losowań zapewniają wariancję zawsze mniejszą od zadanego progu 10-3. Odbywa się to jednak kosztem czasu obliczeń, gdyż wyznaczenie odchylenia standardowego wariancji wymaga co najmniej kilkudziesięciu jej wartości dla danej liczby losowań. Co więcej obliczenia te należy powtórzyć, gdy zmieni się liczba losowań.

Dzięki zaproponowanej modyfikacji otrzymano większą zbieżność uzyskanych wyników oraz wariancję zawsze mniejszą od zadanego progu 10-3. Odbywa się to jednak kosztem czasu obliczeń, gdyż wyznaczenie odchylenia standardowego wariancji wymaga co najmniej kilkudziesięciu jej wartości dla danej liczby losowań.

Bibliografia

[1] https://pl.wikipedia.org/wiki/Metoda_Monte_Carlo

[2] https://pl.wikipedia.org/wiki/Odchylenie_standardowe

[3] https://pl.wikipedia.org/wiki/Metoda_r%C3%B3wnego_podzia%C5%82u