RACHUNEK PRAWDOPODOBIEŃSTWA I STATYSTYKA – PROJEKT

Temat projektu: Generator liczb pseudolosowych

Wykonawca: Artur Górak

SPIS TREŚCI

1. OPIS PROJEKTU

Celem projektu jest zaimplementowanie generatora liczb pseudolosowych o rozkładzie równomiernym bez pomocy gotowych funkcji oraz bibliotek dla generatorów liczb pseudolosowych. Zabronione jest również korzystanie z gotowych źródeł licz pseudolosowych takich jak zegar systemowy. Należy sprawdzić czy generowane przez niego liczby spełniają postulat losowości próby oraz mają rozkład równomierny. W tym celu skorzystałem kolejno z testu serii oraz testu chi kwadrat. Następnie przy pomocy tego generatora stworzyć generator dla liczb z przedziału [0, 1], a następnie na jego podstawie kolejne dla rozkładów: Bernoulliego, dwumianowego, Poissona, wykładniczego oraz normalnego. Dane wygenerowane przez powyższe generatory również trzeba przetestować, czy na pewno tworzą odpowiedni rozkład. W tym celu również posłużyłem się wspomnianym wcześniej testem chi kwadrat. Jako język programowania zdecydowałem się na Pythona ze względu na ogromną ilość bibliotek z których skorzystam w późniejszej części projektu (przede wszystkim do rysowania wykresów, ściągania wartości krytycznej oraz całkowania).

2. GENERATORY

2.1 GENERATOR BLUM BLUM SHUB

Pierwszym zaimplementowanym przeze mnie generatorem jest generator Blum Blum Shub. Został on wymyślony przez Lenorę i Manuela Blumów oraz Micheal'a Shuba w 1968 roku. Ma on postać:

$$x_{n+1} = (x_n)^2 \, mod \, M,$$

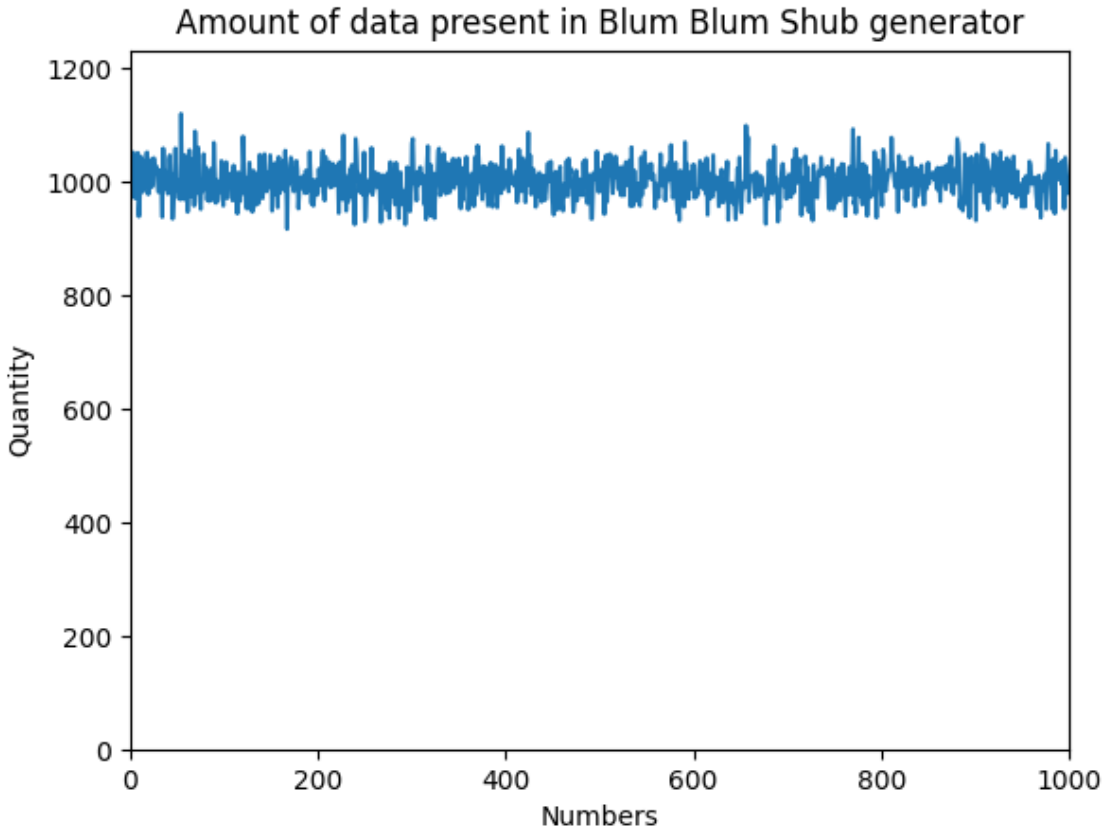
gdzie x_n to kolejne stany, a M to iloczyn dwóch liczb pierwszych p i q dających przy dzieleniu przez 4 resztę 3. W moim przypadku skorzystałem z liczb q = 30000000091, p =40000000003. Zaś jako ziarno zarówno w tym jak i każdym kolejnym generatorze przyjąłem 1619.
Moja implementacja:

```
class Bbs:
    def __init__(self, x0, q, p, quantity):
        self.x0 = x0
        self.random_numbers = []
        m = q * p
        self.random_numbers.insert(0, x0)
        for x in range(1, quantity):
            self.random_numbers.insert(x, (self.random_numbers[x - 1] *
self.random_numbers[x - 1]) % m)
```

Liczby z przedziału [0, 1] tworzymy przy pomocy modulo każdej z wygenerowanych poprzednio liczb przez 10^n gdzie n wybieramy w zależności jak wiele różnych liczb z przedziału chcemy uzyskać. Po czym otrzymane wyniki dzielimy wcześniejsze 10^n. Za te działania odpowiada u mnie metoda *generate_random_numbers*, która przyjmuje pustą tablicę do której chcemy wpisywać nasze liczby oraz dokładność naszych liczb (wspomniane wcześniej 10^n):

```
def generate_random_numbers(self, array, accuracy):
    for x in range(len(self.random_numbers)):
        array.insert(x, self.random_numbers[x] % accuracy / accuracy)
```

Rozkład liczb w generatorze:



W celu sprawdzenia czy dane mają rozkład równomierny oprócz wykresu posłużymy się również testem chi kwadrat. W tym celu zaimplementowałem metodę chi_square, która przyjmuje na wejściu tablicę losowymi liczbami z przedziału [0, 1]. W celu realizacji testu podzieliłem ten przedział na 7 równych boxów i do k-tego boxa „wrzucałem" liczbę jeśli znajdowała się w przedziale $\left(\frac{k}{7}, \frac{k+1}{7}\right)$. Następnie boxy, które skorzystałem ze wzoru:

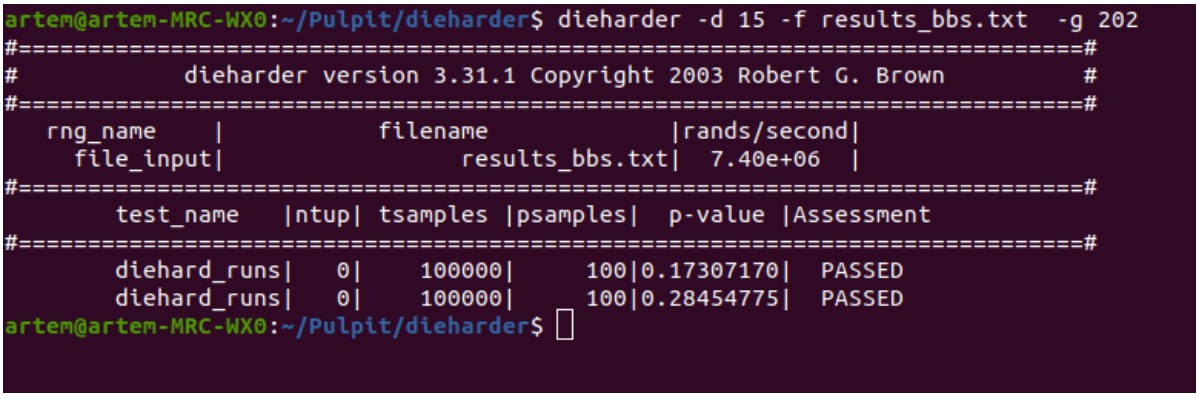
$$\chi^2 = \sum_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i}$$

gdzie O_i to ilość elementów, zaś E_i to oczekiwana ilość elementów w i-tym boxie. Jako wartości oczekiwane w każdym boxie brałem 1/7 * n, gdzie n to ilość wszystkich wygenerowanych liczb. Akceptowałem tylko te boxy do których wpadło więcej niż 5 liczb. Ilość tych szufladek równy jest stopniom swobody, zaś jako poziom istotności (alfa) przyjąłem 0.05. Następnie pobieram odpowiednią wartość z tablic rozkładu chi-kwadrat (przy pomocy biblioteki scipy) i przyrównujemy do wyliczonej przez nas wartości. Test uznajemy za zdany jeśli nasza wartość jest niższa od tej z tablic. Ten generator poprawnie zdaje ten test.

```
def chi_square(self, array):
    expected = []
    actual = []
    n = 7
    for x in range(n):
        expected.insert(x, int(len(array)/n))
        actual.insert(x, 0)

    for x in array:
        if x < 1/n:
            actual[0] += 1
        elif x < 2/n:
            actual[1] += 1
```

```
        elif x < 3/n:
            actual[2] += 1
        elif x < 4/n:
            actual[3] += 1
        elif x < 5/n:
            actual[4] += 1
        elif x < 6/n:
            actual[5] += 1
        else:
            actual[6] += 1

    chi = 0
    degrees = 0
    for x in range(n):
        if actual[x] > 5 and expected[x] > 5:
            chi += (actual[x] - expected[x]) ** 2 / expected[x]
            degrees += 1

    alfa = 0.05
    crit = stats.chi2.ppf(q=1 - alfa, df=degrees-1)
    if chi < crit:
        print("The distribution is consistent with the uniform distribution
 ")
    else:
        print("The distribution is not consistent with the uniform
distribution
 ")
```

W kolejce do sprawdzenia został tylko sprawdzenie postulatu losowości próby. Zrobiłem to na 2 sposoby. Pierwszym z nich jest wygenerowanie przez generator do pliku tekstowego 1000000000 liczb i skorzystanie z zaproponowanego w treści projektu programu dieharder. Powyższy generator pomyślnie zdał przygotowane przez ten przez program testy serii, czego jako dowód umieszczam poniżej screenshot.



Drugim sposobem jest ręczne zaimplementowanie testu serii, który by sprawdził zgodność z danych z postulatem. U mnie odpowiada za to metoda *runs_test()*

```
def runs_test(self):
    tmp = []
    length = len(self.random_numbers)
    for x in range(length):
        tmp.insert(x, self.random_numbers[x])
```

```
    tmp.sort()
    median = 0
    if length % 2 == 0:
        median = 0.5 * (self.random numbers[int(length / 2)] +
self.random_numbers[int((length - 1) / 2)])
    else:
        median = self.random_numbers[int(length / 2)]

    runs = 0
    a_freq = 0
    b_freq = 0

    series_array = []
    for x in range(length):
        if self.random_numbers[x] > median:
            series_array.insert(x, 'a')
        elif self.random_numbers[x] < median:
            series_array.insert(x, 'b')
        else:
            series_array.insert(x, 'c')

    if series_array[0] == 'a':
        runs += 1
        a_freq += 1
    elif series_array[0] == 'b':
        runs += 1
        b_freq += 1

    for x in range(1, length):
        if series_array[x] == 'a' and series_array[x-1] == 'b':
            runs += 1
        elif series_array[x] == 'b' and series_array[x-1] == 'a':
            runs += 1
        elif series_array[x] == 'b' and series_array[x-1] == 'c' and
series_array[x-2] == 'a':
            runs += 1
        elif series_array[x] == 'a' and series_array[x-1] == 'c' and
series_array[x-2] == 'b':
            runs += 1

        if series_array[x] == 'a':
            a_freq += 1
        elif series_array[x] == 'b':
            b_freq += 1

    ek = 2*a_freq*b_freq/length + 1
    dk = math.sqrt((2*a_freq*b_freq*(2*a_freq*b_freq - length))/((length -
1) * length * length))

    z = (runs - ek) / dk
    z_for_005 = 1.96
    # alfa = 0.05
    # p_values_one = stats.norm.sf(abs(z))  # one-sided
    # p_values_two = stats.norm.sf(abs(z)) * 2  # twosided
    #

    if abs(z) > z_for_005:
        print("We reject the null hypothesis, i.e. the postulate of sample
randomness")
    else:
```

```
        print("We cannot reject the null hypothesis, i.e. the randomness of
the sample")
```

Polega ona na podziału naszych danych na 2 grupy. Zrobiłem to przy pomocy mediany. A mianowicie liczby mniejsze od niej należą do grupy a, większe zaś do b. Sama mediana to oddzielna grupa c, której później nie bierzmy pod uwagę w naszych działaniach. Następnie liczymy ile jest serii w naszych danych oraz ilość występowania danych w poszczególnych grupach. Jako że moich danych jest na pewno powyżej 40, więc aby przeprowadzić test musimy skorzystać ze wzorów na wartość średnią i wariancję:

$$\mathbb{E}(K) = \frac{2n_A n_B}{n} + 1,$$
$$D^2(K) = \frac{2n_A n_B(2n_A n_B - n)}{(n-1)n^2}.$$

Później musimy wyliczyć statystykę Z:

$$Z = \frac{K - E(K)}{D(K)}$$

Wiemy że ma ona rozkład normalny. Aby sprawdzić czy nasze Z znajduje się w regionie krytycznym, muszę je przyrównać do: $Z_{1-\alpha/2}$. Wiemy że dla α = 0.5 przyjmuje wartość 1.96. Nie ma podstaw do odrzucania postulatu jeśli $|Z| < Z_{1-\alpha/2}$, co też sprawdzam w powyższej metodzie. Wynik wyszedł analogiczny do tego z diehardera, czyli bezproblemowe zaliczenie testu.

2.2 GENERATOR MIESZANY

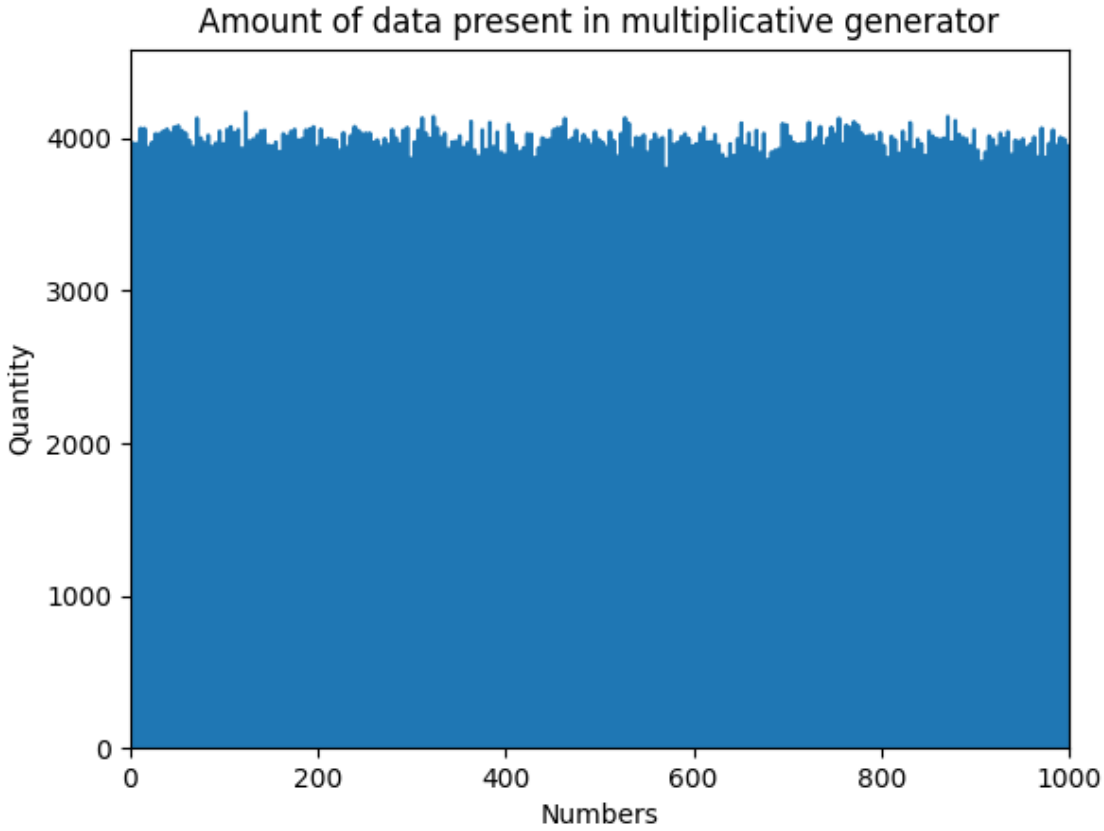
Kolejnym generatorem jest generator multiplikatywny dany wzorem:

$$x_{n+1} = (a \cdot x_n + c) \; mod \; m,$$

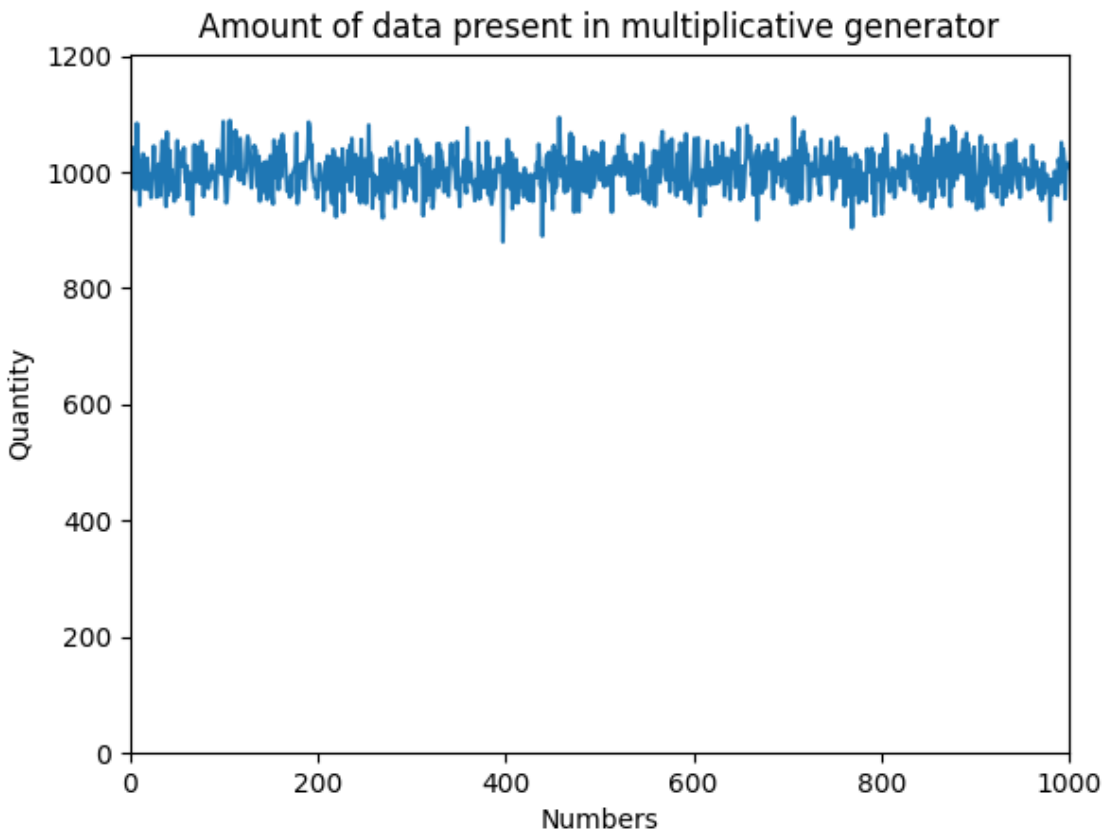
Gdzie my dobieramy odpowiednie a, c oraz m (w sytuacji gdy c = 0 generator nazywany jest multiplikatywnym). W moim przypadku zdecydowałem się przetestować 3 zestawy tych danych proponowane przez różne książki i firmy (Numerical Recipes, APPLE, Microsoft Visual). Poniżej implementacja:

```
class Multiplicative:
    def __init__(self, seed, quantity):
        self.random_numbers = []

        # # dane dla wersji Numerical Recipes
        # a = 1664525
        # m = 2**32
        # c = 1013904223

        # # dane dla wersji APPLE
        # a = 1220703125
        # m = 2 ** 35
        # c = 0

        # dane dla wersji Microsoft Visual
        a = 214013
        m = 2 ** 32
        c = 2531011

        n = seed

        self.random_numbers.insert(0, seed)
        for x in range(1, quantity):
            self.random_numbers.insert(x, (a * self.random_numbers[x - 1] +
c) % m)
```

Po testach najgorzej wypadł zestaw 2, ponieważ pojawiły się „dziury” w danych, czyli istnieją dane które w ogóle się nie pojawiły, co pokazuje poniższy wykres:



Ale przy wybraniu odpowiedniego a, c i m generator przechodzi test chi kwadrat oraz test serii. Analogiczny wykres dla a, c i m z Microsoft Visual:



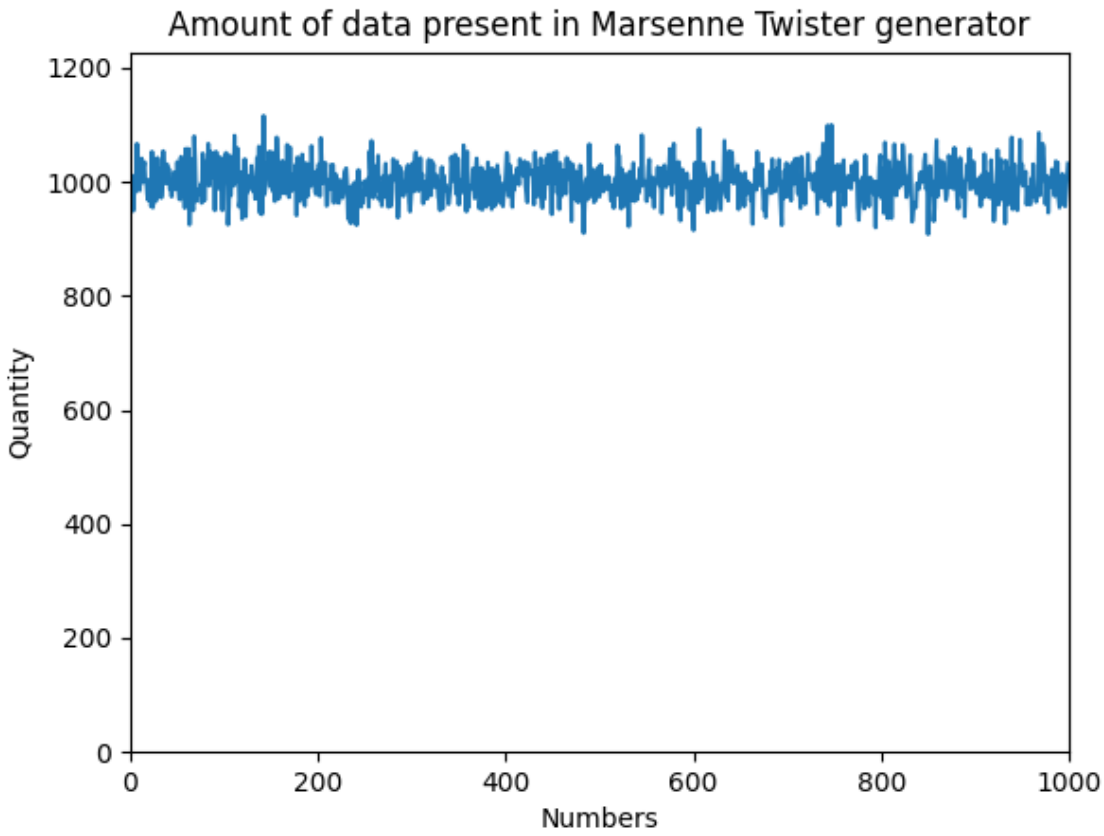
2.3 GENERATOR MARSENNE TWISTER

W celu porównania różnych generatorów zaimplementowałem zaproponowany w treści projektu generator Marsenne Twister. Został on wynaleziony przez Makoto Matsumoto i Takuji Nishimura w 1997 roku. Nazwa pochodzi od faktu wykorzystania w nim liczb pierwszych Marsenna, czyli liczb pierwszych, które można zapisać w postaci $2^n - 1$. Poniżej implementacja:

```
class Marsenne:
    def __init__(self, seed):
        self.mt = []
        self.index = 0
        self.mt.insert(0, seed)
        self.rand_num_array = []
        for i in range(1, 624):
            tmp = 1812433253 * (self.mt[i - 1] ^ (self.mt[i - 1] >> 30)) +
i
            mask = ~(~0 << 32)
            self.mt.insert(i, tmp & mask)

    def generate(self):
        mask = (1 << 31) - 1
        for i in range(624):
            tmp = mask & (self.mt[((i + 1) % 624)])

            y = (1 & (self.mt[i] >> (32 - 1))) + tmp
```

```
            self.mt[i] = self.mt[(i + 397) % 624] ^ (y >> 1)
            if y % 2 == 1:
                self.mt[i] = self.mt[i] ^ 2567483615

    def extract_numbers(self):
        if self.index == 0:
            self.generate()

        y = self.mt[self.index]
        y = y ^ (y >> 11)
        y = y ^ ((y << 7) & 2636928640)  # 0x9d2c5680
        y = y ^ ((y << 15) & 4022730752)  # 0xefc60000
        y = y ^ (y >> 18)

        self.index = (self.index + 1) % 624
        return y

    def random_numbers(self, array, quantity, limit):

        for x in range(quantity):
            self.rand_num_array.insert(x, self.extract_numbers() % limit)

        for x in range(len(self.rand_num_array)):
            array.insert(x, self.rand_num_array[x]/limit)
```

Generator tak jak oba powyższe generator przechodzi test chi kwadrat oraz test serii.

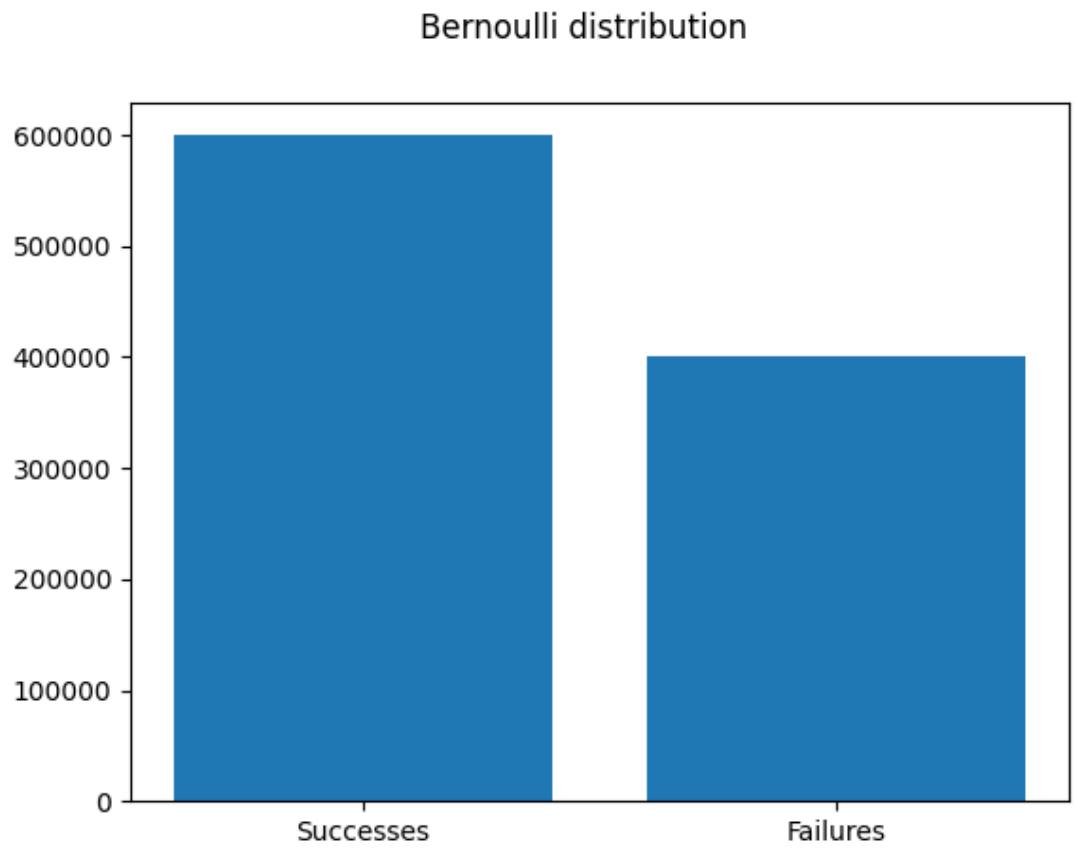


3. ROZKŁAD BERNOULLIEGO

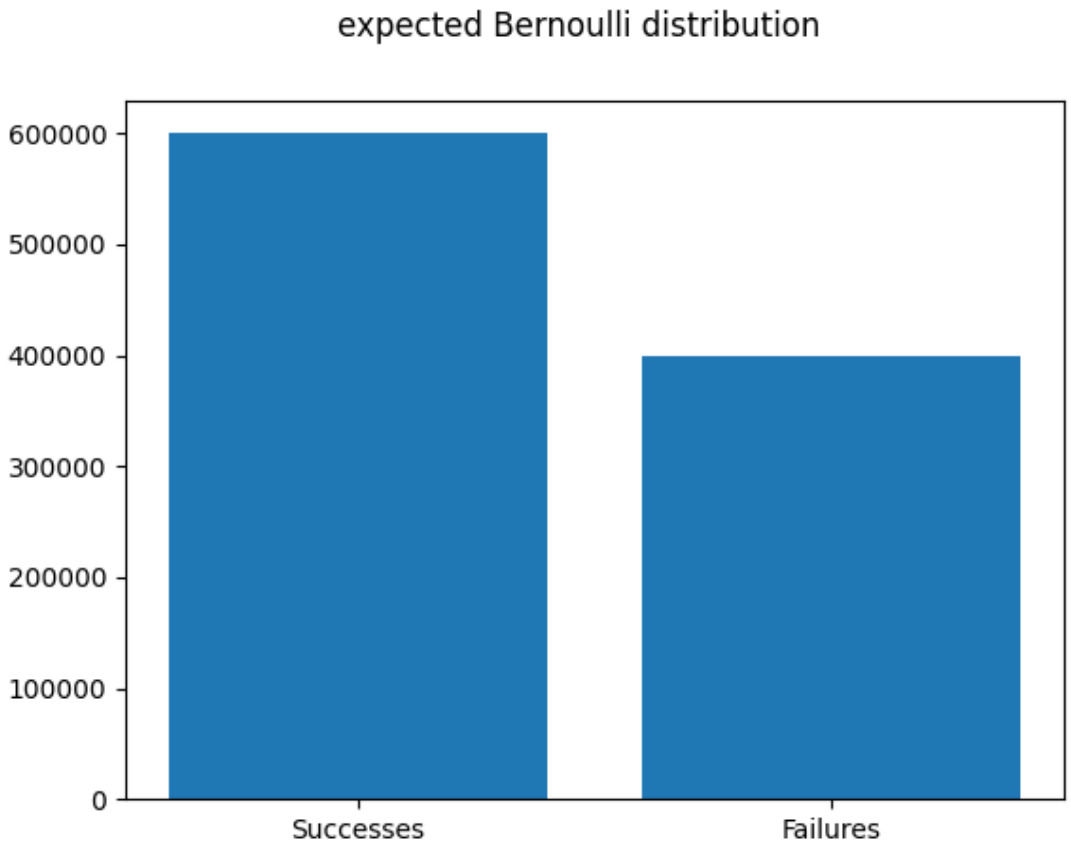
Aby stworzyć rozkład Bernoulliego losuję liczbę z przedziału [0, 1], jeśli jest mniejsza bądź równa danemu prawdopodobieństwu p to odnotowujemy ją jako sukces, w przeciwnym wypadku jako porażkę. Implementacja:

```
class Bernoulli:
    def __init__(self, p, quantity, random_numbers):
        self.p = p
        self.n = quantity
        self.distribution = []
        self.distribution.insert(0, 0)
        self.distribution.insert(1, 0)
        for x in random_numbers:
            if x <= p:
                self.distribution[0] += 1
            else:
                self.distribution[1] += 1
```

Wykres przedstawiający powyższy rozkład dla p = 0.6 oraz n = 1000000:



W celu wykonania testu chi kwadrat wyznaczam dwa koszyki, jeden na sukcesy, drugi na porażki. Oznaczam wartość oczekiwaną sukcesów jako p * n oraz porażek jako (1 – p) * n. Wartości oczekiwane prezentuje poniższy histogram:



Następnie korzystam z przedstawionego już wcześniej wzoru, pobieram z tablic wartość krytyczną i porównuję. Wygenerowany przeze mnie rozkład zdaje na test na rozkład Bernoulliego.

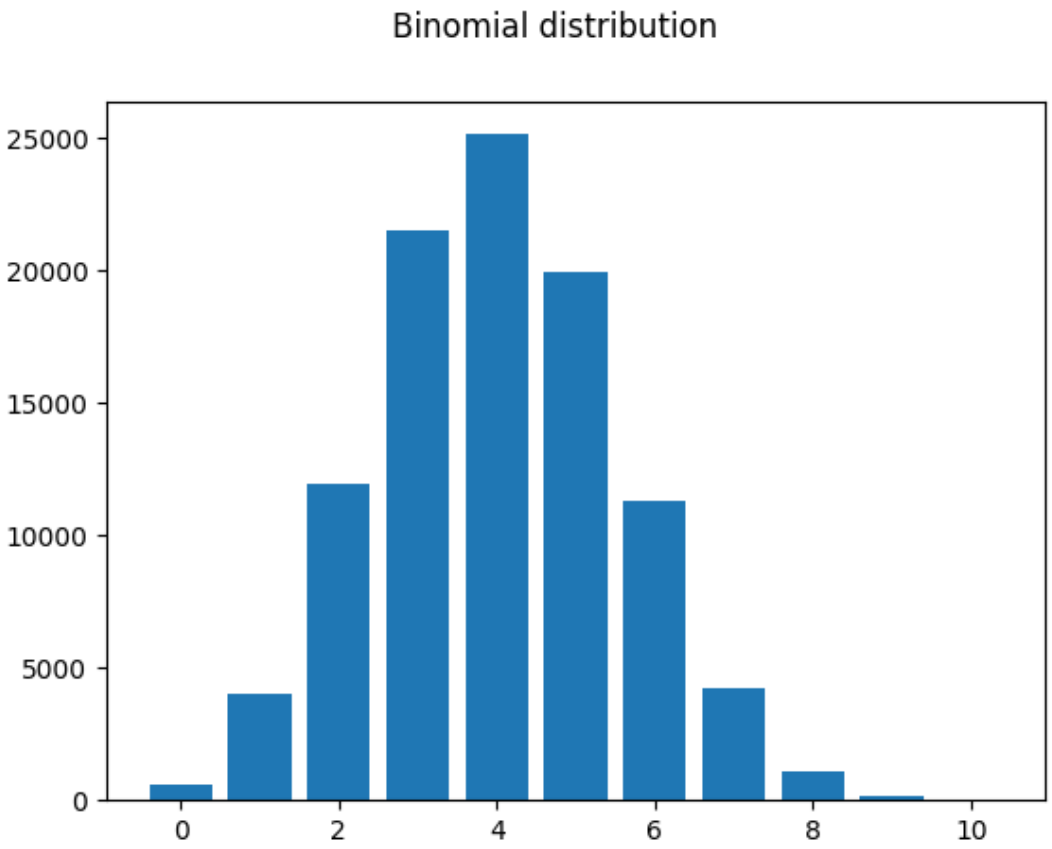
4. ROZKŁAD DWUMIANOWY

Rozkład bezpośrednio związany z rozkładem Bernoulliego. Aby go utworzyć wykonuję na n liczbach próbę Bernoulliego z prawdopodobieństwem p i zliczam sukcesy. Moja implementacja, gdzie p – prawdopodobieństwo, size – nasze wcześniejsze n, random_numbers – tablica losowych liczb z przedziału [0,1]:

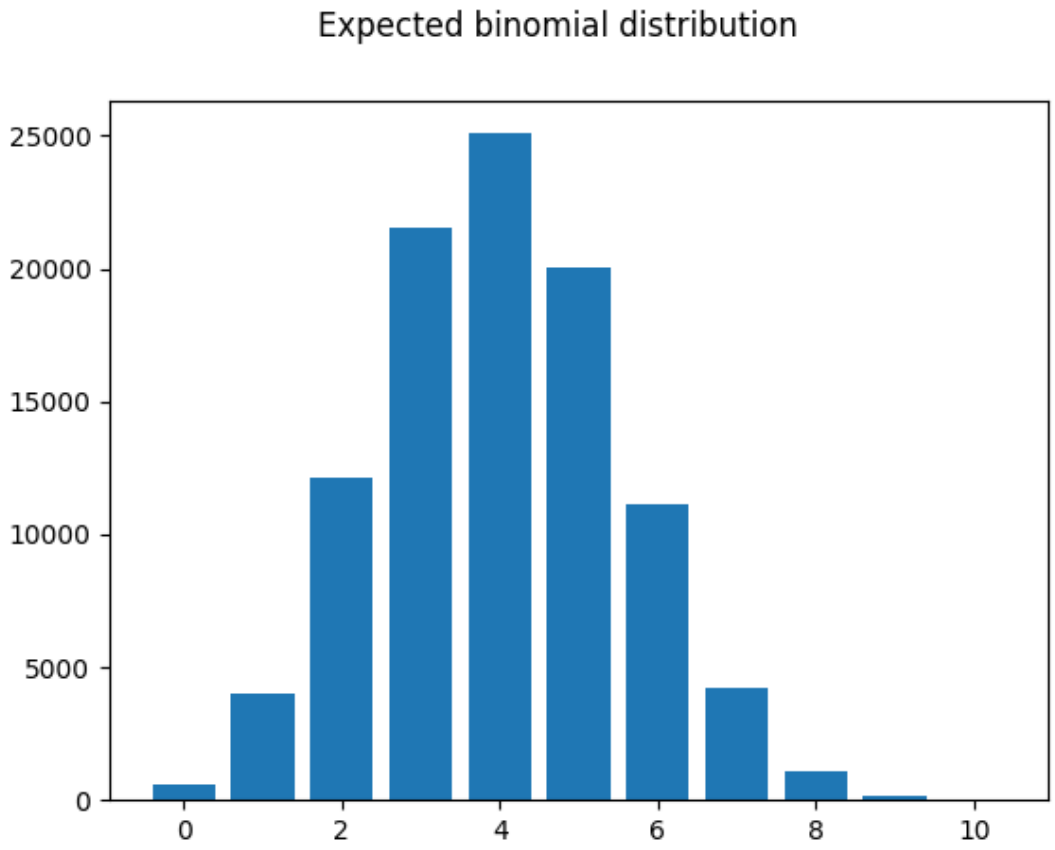
```
class Binomial:
    def __init__(self, p, size, random_numbers):
        self.p = p
        self.size = size
        self.results = []
        self.frequency = []
        self.count = []
        self.rn_size = len(random_numbers)
        iterator = 0
        for x in range(int(self.rn_size / size)):
            successes = 0
            for i in range(size):
                if random_numbers[iterator] < p:
                    successes += 1
                iterator += 1

            self.results.insert(x, successes)

        for x in range(size + 1):
            self.frequency.insert(x, 0)
```

```
            self.count.insert(x, x)

        for x in self.results:
            self.frequency[x] += 1
```

Histogram rozkładu dwumianowego dla p = 0.4, size = 10 oraz 1000000 liczbach.



Oczekiwany rozkład dwumianowy tworzymy korzystając ze wzoru funkcji rozkładu prawdopodobieństwa, podstawiając do niego kolejne wartości i mnożąc otrzymaną liczbą przez ilość wykonanych serii prób Bernoulliego. Poniżej przedstawiam histogram wartości oczekiwanych dla tego rozkładu.



W teście chi kwadrat boxami są ilości sukcesów w k próbach (na histogramie oś OX). Dalsze działania analogiczne jak w poprzednich przypadkach. Implementacja tego testu dla rozkładu dwumianowego:

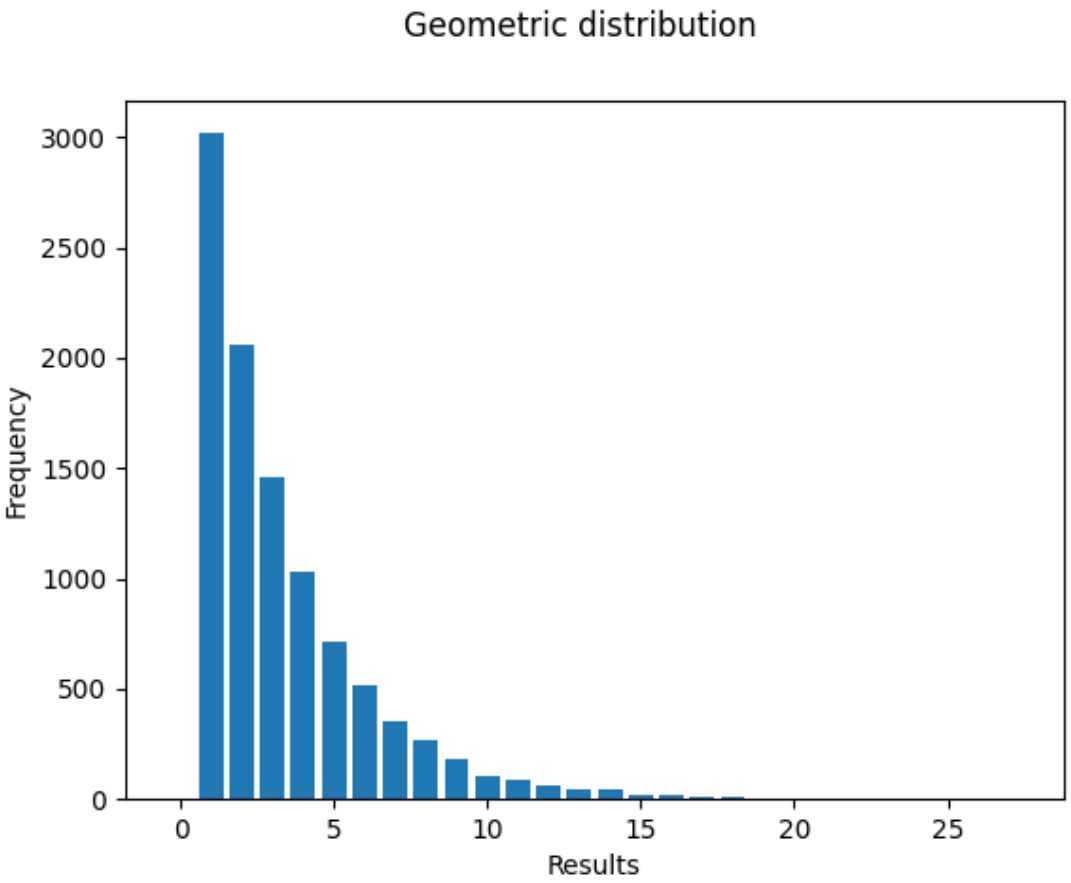
```
def chi_square(self):
    expected = []
    for x in range(self.size + 1):
        expected.insert(x,
(math.factorial(self.size)/(math.factorial(self.size - x) *
math.factorial(x)) * math.pow(self.p, x) * math.pow(1 - self.p, self.size -
x))*(self.rn_size / self.size))

    plt.bar(self.count, expected)
    plt.suptitle('Expected binomial distribution ')
    plt.show()

    chi = 0
    degrees = 0
    for i in range(self.size + 1):
        if self.frequency[i] > 5 and expected[i] > 5:
            chi += (self.frequency[i] - expected[i]) ** 2 / expected[i]
            degrees += 1

    alfa = 0.05
    crit = stats.chi2.ppf(q=1 - alfa, df=degrees - 1)
    if chi < crit:
        print('The distribution is consistent with the binomial
distribution')
    else:
        print('The distribution is not consistent with the binomial
distribution')
```

5. ROZKŁAD GEOMETRYCZNY

Rozkład jest podobny do rozkładu dwumianowego z tą różnicą, że zamiast ilość sukcesów w n próbach, badamy w którym teście Bernoulliego będzie pierwszy sukces. Implementacja:

```
class Geometric:
    def __init__(self, p, n, quantity, random_numbers):
        self.p = p
        self.n = n
        self.quantity = quantity
        results = []
        results_frequency = []
        self.max_result_freq = 0
        iterator = 0

        for x in range(quantity):
            results_frequency.insert(x, 0)

        for x in range(n):
            X = 1

            if iterator == quantity:
                break

            u = random_numbers[iterator]
            iterator += 1

            while u > p:
                X += 1
                if iterator == quantity:
                    break
                u = random_numbers[iterator]
                iterator += 1

            results.insert(x, X)
            results_frequency[X] += 1

            if self.max_result_freq < X:
                self.max_result_freq = X

        self.results_frequency_without_tail = []
        self.count = []

        for x in range(self.max_result_freq):  # delete a tail of zeros
            self.count.insert(x, x)
            self.results_frequency_without_tail.insert(x,
results_frequency[x])
```

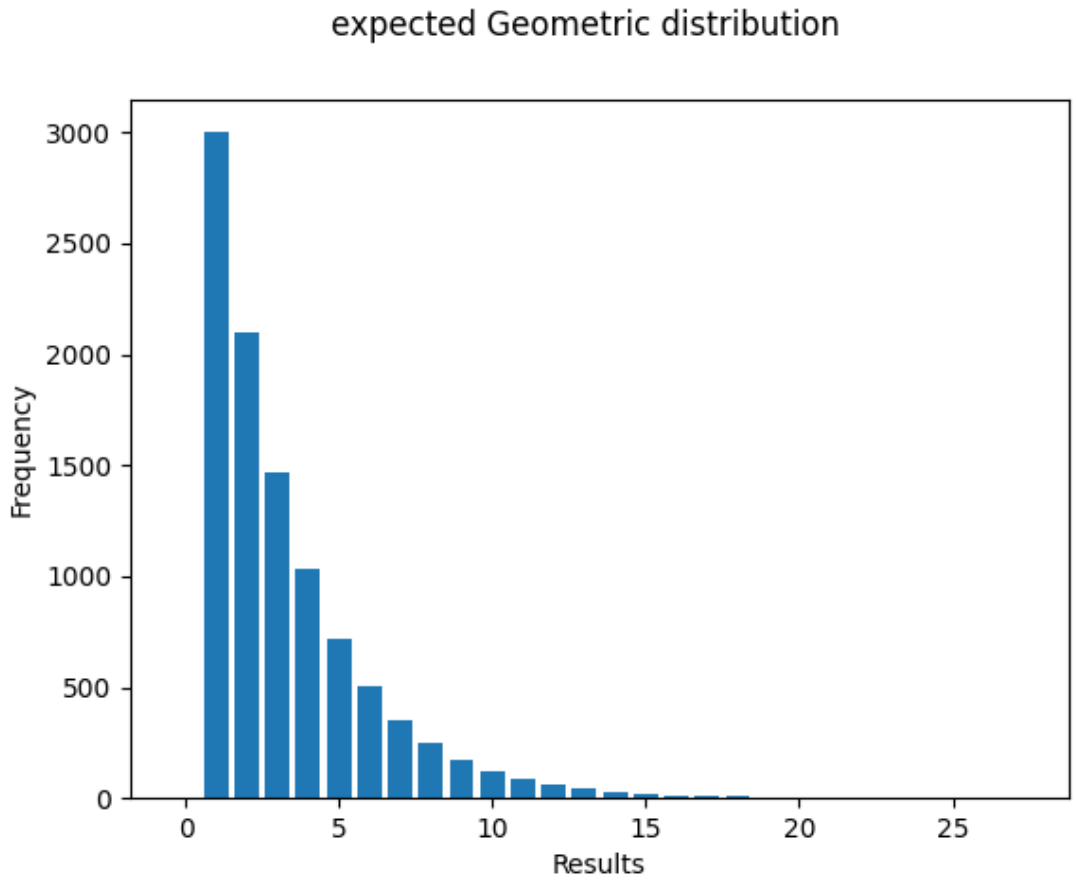
Ostatnie linijki powyższego kodu są odpowiedzialne za skrócenie tablicy poprzez odcięcie pół z zerami na końcu tablicy. Poniżej histogram tego rozkładu dla p = 0.3, n = 10000 oraz how_many_numbers = 1000000



Do testu chi kwadrat jako boxy obierzemy sobie numer próby Bernoulliego w której był pierwszy sukces (oś OX na histogramie) W celu wyliczenia wartości oczekiwanych dla każdego koszyka wyliczymy ze wzoru:

$$P(X = k) = (1 - p)^{k-1} p$$

Po wyliczeniu wszystkich wartości oczekiwanych otrzymujemy poniższy histogram:



Po wykonaniu testu chi kwadrat otrzymujemy wynik pozytywny. Implementacja tego testu:

```
def chi_square(self):

    frequency_exp = []
    frequency_exp.insert(0, 0)

    for k in range(1, self.max_result_freq):
        frequency_exp.insert(k, (1 - self.p) ** (k - 1) * self.p * self.n)

    degrees = 0
    chi = 0
    for i in range(1, self.max_result_freq):
        if self.results_frequency_without_tail[i] > 5 and frequency_exp[i]
> 5:
            chi += (self.results_frequency_without_tail[i] -
frequency_exp[i]) ** 2 / frequency_exp[i]
            degrees += 1

    alfa = 0.05
    crit = stats.chi2.ppf(q=(1 - alfa), df=degrees - 1)

    if chi < crit:
        print("The distribution is consistent with the geometric
distribution ")
    else:
        print("The distribution is not consistent with the geometric
distribution ")

    plt.bar(self.count, frequency_exp)
```

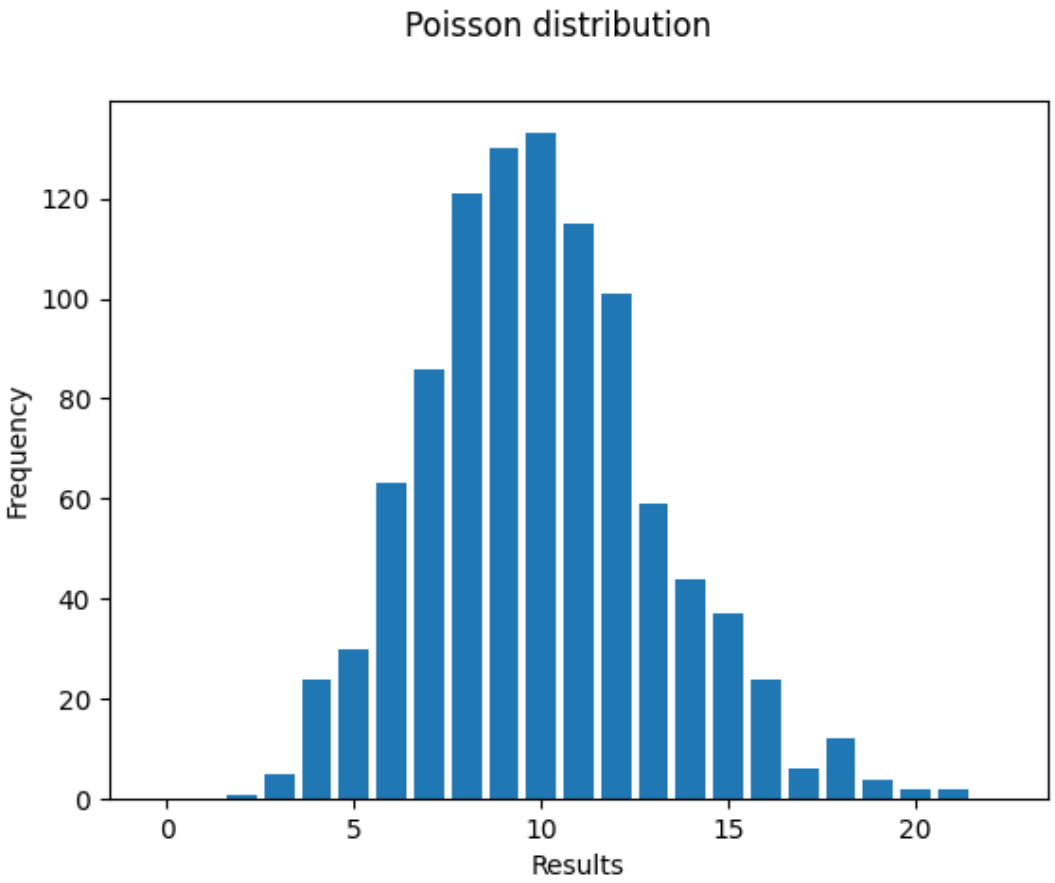
```
    plt.suptitle('expected Geometric distribution')
    plt.xlabel('Results')
    plt.ylabel('Frequency')
    plt.show()
```

6. ROZKŁAD POISSONA

Generator rozkładu Poissona zrobiłem według schematu poznanego na wykładzie i polega on na skorzystaniu z twierdzenia, że x mająca rozkład Poissona dana jest wzorem: $x = \max\{k: U_1 \cdot \ldots \cdot U_k \geq e^{-\lambda}\}$. Generacja pojedynczej realizacji zmiennej losowej z rozkładu Poissona o parametrze lambda polega na losowaniu U z rozkładu jednostajnego oraz ustawieniu X na 0 i dopóki $U \geq e^{-\lambda}$ to „domnażam" do U kolejną liczbę losową i dodaję do X jedynkę. Taką realizację robię n-krotnie.

```
class Poisson:
    def __init__(self, lamb, n, quantity, random_numbers):
        self.lamb = lamb  # oczekiwana liczba zdarzen
        self.n = n  # liczba generacji w generatorze losowym
        self.quantity = quantity
        results = []
        results_frequency = []
        self.max_result_freq = 0

        self.results_frequency_without_tail = []
        self.count = []

        for x in range(n):
            results_frequency.insert(x, 0)

        iterator = 0
        for x in range(n):
            if iterator == quantity:
                break
            u = random_numbers[iterator]
            iterator += 1

            X = 0

            while u >= math.exp((-1) * lamb):
                if iterator == quantity:
                    break

                u = u * random_numbers[iterator]
                iterator += 1
                X += 1

            results.insert(x, X)
            results_frequency[X] += 1

            if self.max_result_freq < X:
                self.max result freq = X

        for x in range(self.max_result_freq):  # delete a tail of zeros
            self.count.insert(x, x)
            self.results_frequency_without_tail.insert(x,
results_frequency[x])
```

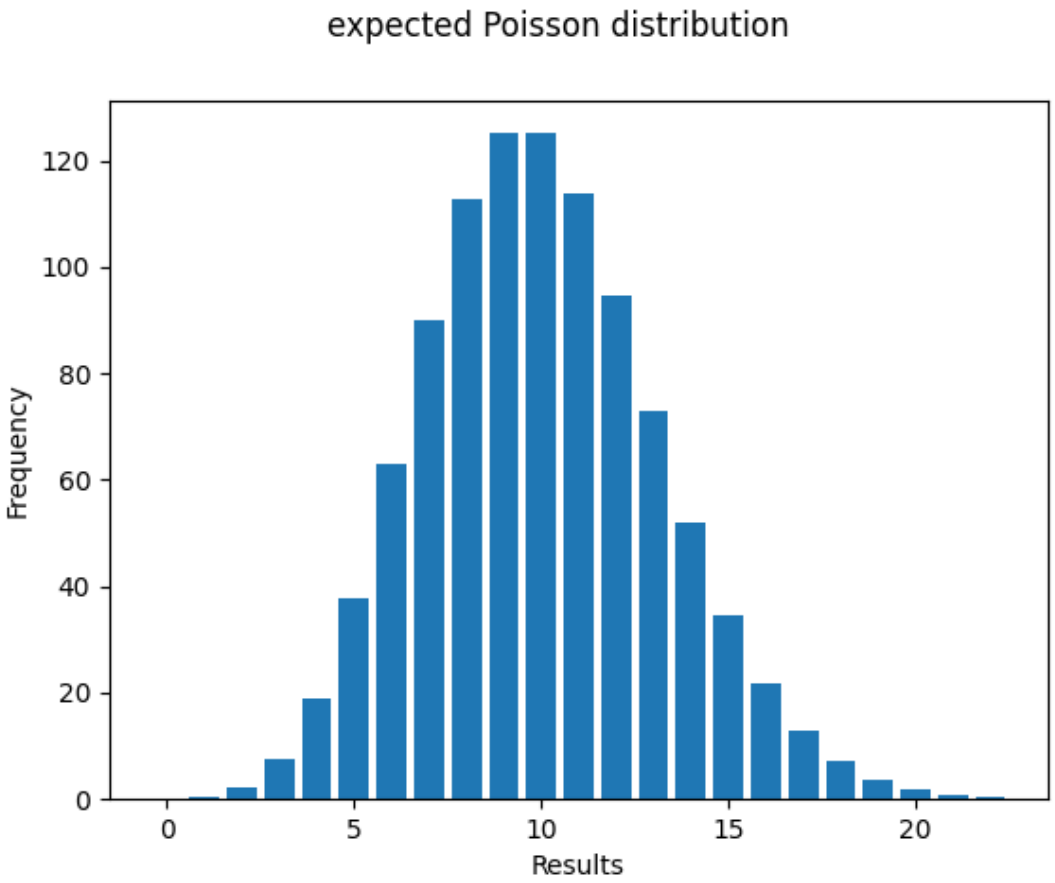
Histogram dla otrzymanego rozkładu Poissona dla lambda = 10 i n = 1000.



W teście chi kwadrat jako koszyki weźmiemy konkretne wartości X. Wartości oczekiwane odnośnie ilości wystąpień danego X wyliczymy ze poniższego wzoru na funkcję rozkładu prawdopodobieństwa dla rozkładu Poissona:

$$f(k,\lambda) = \frac{\lambda^k e^{-\lambda}}{k!}$$

Otrzymane wartości widoczne są na poniższym histogramie.



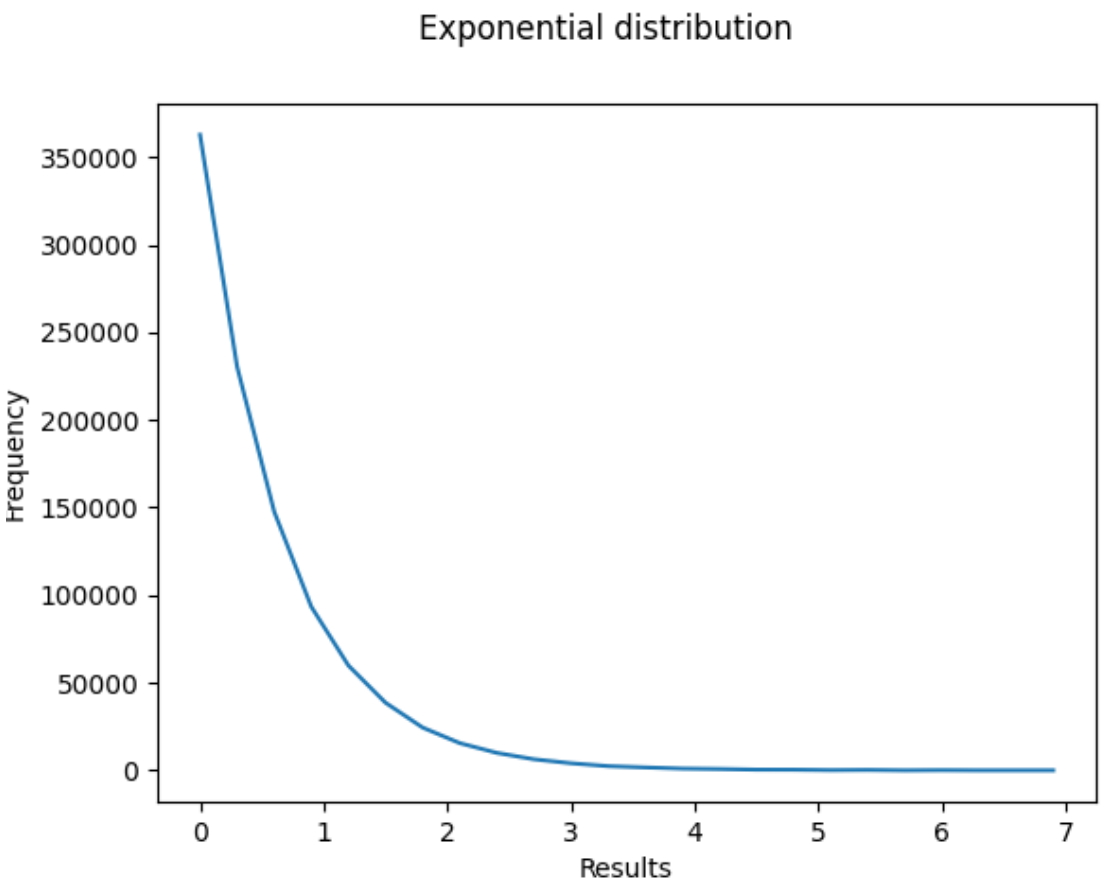
Test po wykonaniu go na tym rozkładzie daje wynik pozytywny.

7. ROZKŁAD WYKŁADNICZY

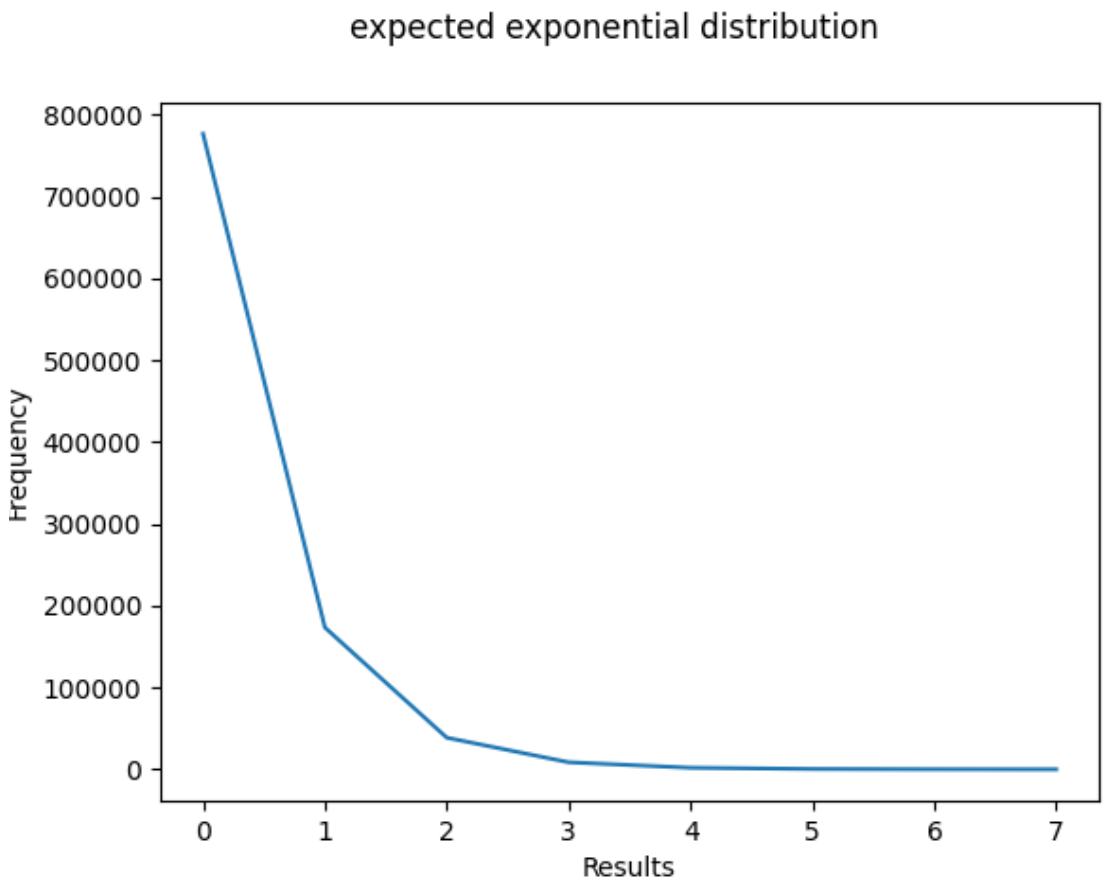
W celu utworzenia generatora rozkładu wykładniczego posłużyłem się metodą odwrotnej dystrybuanty. W tym celu losowałem za pomocą generatora wartość z przedziału [0,1] a następnie sprawdzałem dla jakiego x dystrybuanta rozkładu wykładniczego przyjmie taką wartość. Następnie po wyliczeniu wszystkich punktów podzieliłem przestrzeń x na boxy o szerokości accurancy. I do każdego wpisywałem liczbę x-ów należących do wyznaczanego przez każdy kubełek przedziału. Poniżej znajduje się implementacja tego algorytmu oraz wykres wyliczonego rozkładu dla λ = 1.5, n = 1000 oraz accurancy = 0.3.

```
class Exponential:
    def __init__(self, lamb, n, accuracy, quantity, random_numbers):
        self.lamb = lamb
        self.n = n
        self.quantity = quantity
        self.accuracy = accuracy
        self.results = []
        self.results_frequency = []
        self.max_result_freq = 0
        self.count = []

        for x in range(quantity):
            self.results.insert(x, (-1) * math.log(1 - random_numbers[x]) /
```

```
lamb)

        self.results.sort()
        self.maximum = self.results[quantity - 1]
        self.maximum = round(self.maximum) + 1

        self.iterator = 0
        i = 0
        while i <= self.maximum:
            self.count.insert(self.iterator, i)
            i += self.accuracy
            self.iterator += 1

        for x in range(self.iterator):
            self.results_frequency.insert(x, 0)

        for x in self.results:
            for j in range(self.iterator):
                if x < self.count[j]:
                    self.results_frequency[j - 1] += 1
                    break
```



Aby móc skorzystać z testu chi kwadrat podzieliłem oś x na 7 przedziałów i celu wyliczenia wartości oczekiwanej dla każdego z nich scałkowałem na danym przedziale wzór na gęstość prawdopodobieństwa rozkładu wykładniczego $\lambda e^{-\lambda x}$. W tym celu skorzystałem z funkcji squad z biblioteki scipy. Następnie analogicznie podzieliłem wcześniej wyliczone w generatorze wartości, podstawiłem do wzoru na wartość chi kwadrat i porównałem z wartością krytyczna. Test zakończył się wynikiem pozytywnym. Poniżej znajduje się wykres dla wartości oczekiwanych w przedziałach w rozkładzie wykładniczym.

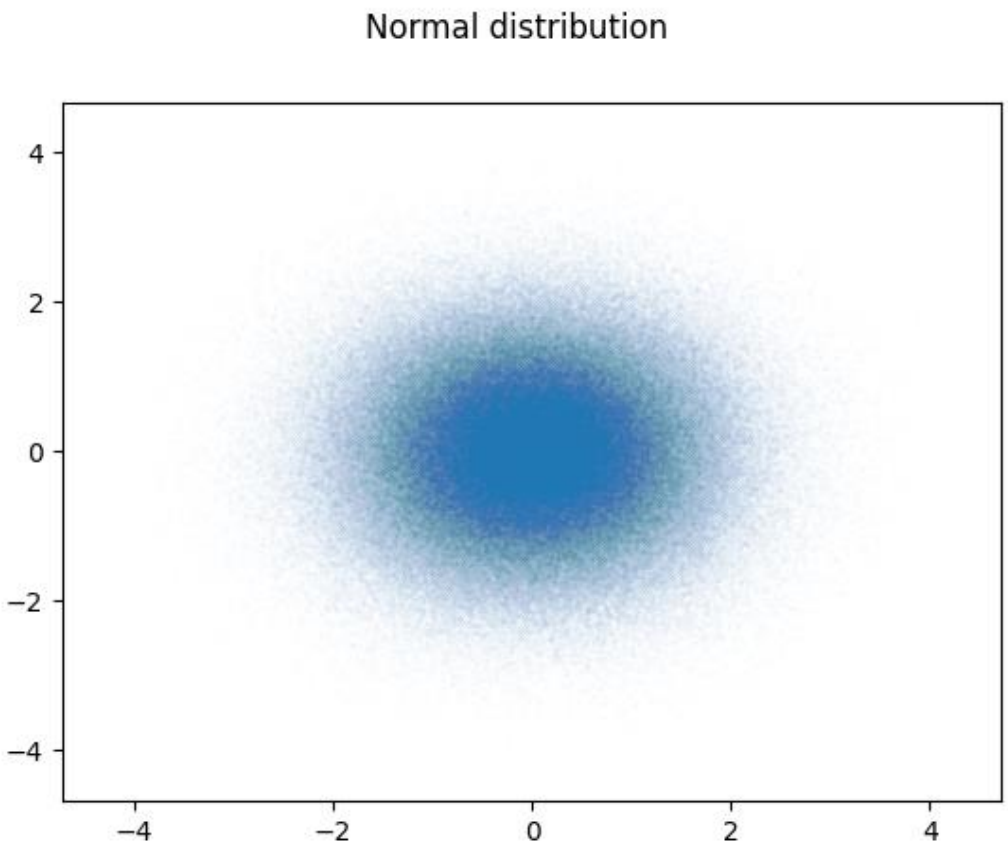


8. ROZKŁAD NATURALNY

Ostatnim rozkładem, którego generator zrobiłem jest generator rozkładu naturalnego. W jego realizacji posłużyłem zaproponowaną na wykładzie się transformacją Boxa-Mullera. Za pomocą poniższych wzorów pozwala wygenerować niezależnie zmienne losowe Z_1 oraz Z_2 o rozkładzie normalnym i odchyleniu standardowym 1. Niech U_1 i U_2 będą niezależnymi zmiennymi losowym o rozkładzie jednostajnym na przedziale (0, 1]. Wówczas:

$R^2 = -2 \cdot \ln U_1$
$\Theta = 2\pi U_2$
$Z_1 = R \cos \Theta$
$Z_2 = R \sin \Theta$

Traktując Z_1 oraz Z_2 jako współrzędne x i y na układzie współrzędnych otrzymamy wówczas:

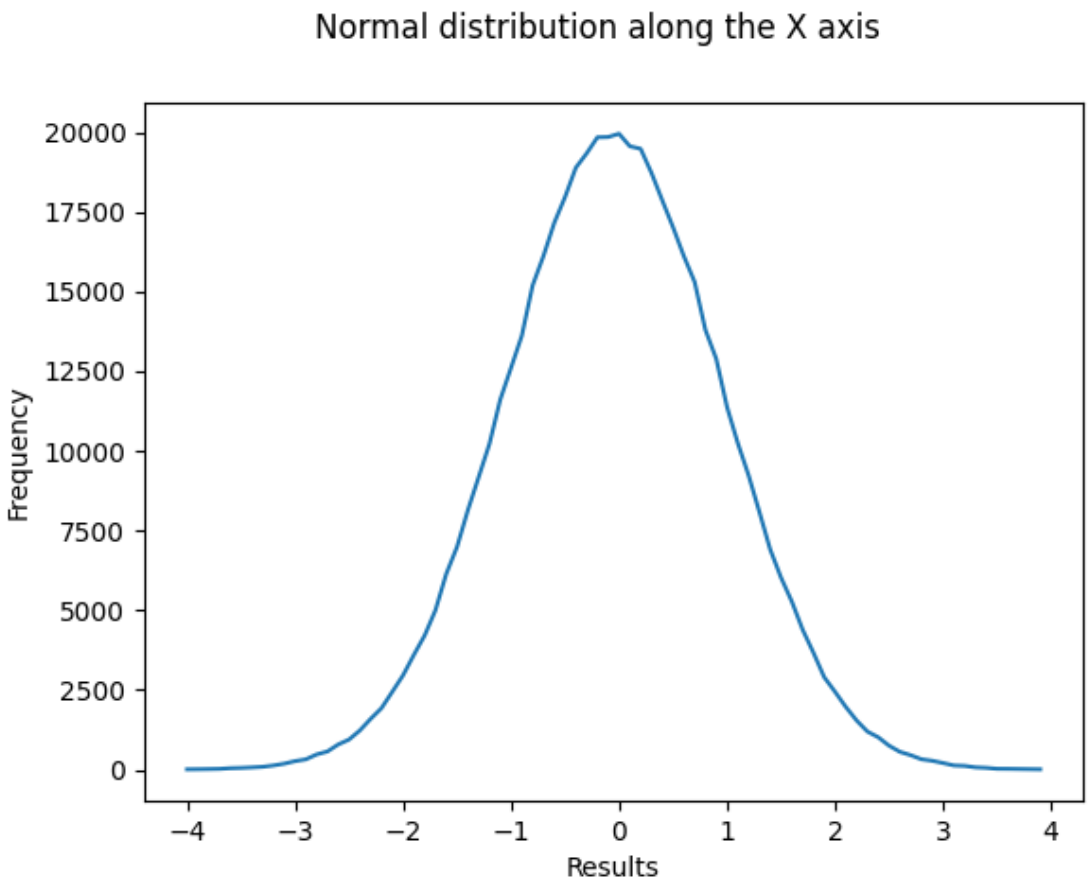


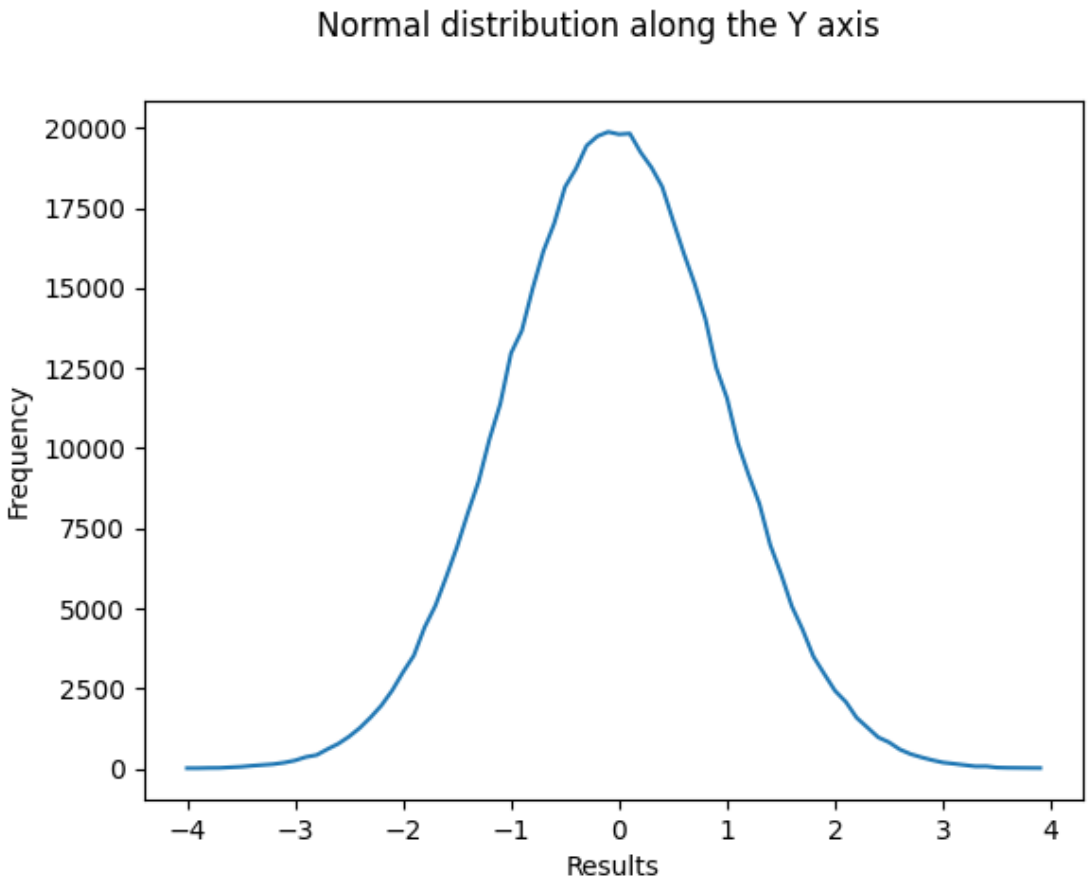
Implementacja wygląda następująco:

```
class Normal:
    def __init__(self, quantity, random_numbers, accuracy):
        self.n = quantity
        self.accuracy = accuracy
        self.x = []
        self.y = []
        i = 0
        iterator = 0
        while i < quantity:
            while random_numbers[i] == 0:
                i += 1
                if i == quantity:
                    break

            if i == quantity:
                break
            theta = 2 * math.pi * random_numbers[i]
            i += 1
            if i == quantity:
                break

            while random_numbers[i] == 0:
                i += 1
                if i == quantity:
                    break
            if i == quantity:
```

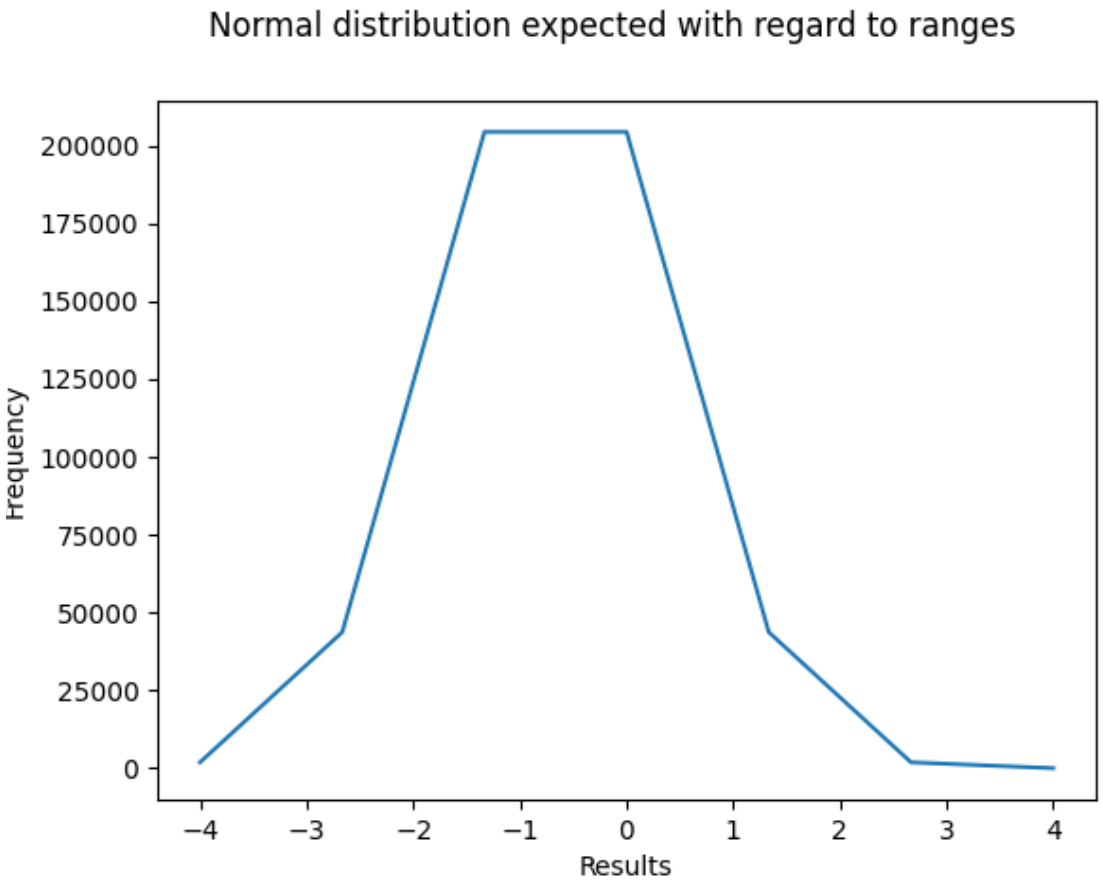
```
            break
        r = math.sqrt((-2) * math.log(random_numbers[i]))

        i += 1
        self.x.insert(iterator, r * math.cos(theta))
        self.y.insert(iterator, r * math.sin(theta))
        iterator += 1

self.frequency_x = []
self.frequency_y = []

self.count = []
self.x.sort()
self.y.sort()

minimum = round(self.x[0])
self.maksimum = (-1) * minimum
iterator = 0
i = minimum
while i <= self.maksimum:
    self.count.insert(iterator, i)
    i += accuracy
    iterator += 1

for x in range(iterator):
    self.frequency_x.insert(x, 0)
    self.frequency_y.insert(x, 0)

for x in self.x:
    for j in range(iterator):
        if x < self.count[j]:
            self.frequency_x[j - 1] += 1
            break

for x in self.y:
    for j in range(iterator):
        if x < self.count[j]:
            self.frequency_y[j - 1] += 1
            break
```

Aby lepiej uwidocznić że powyższy wykres typu scatter przedstawia rozkład normalny wyodrębnię składowe x i y i przedstawię je na osobnych wykresach.





Tak jak poprzednio w celu zastosowania testu chi kwadrat podzieliłem dziedzinę x na 6 przedziałów na których liczyłem ilość punktów na nich się znajdujących. I tak jak poprzednio scałkowałem funkcję gęstości na odpowiednich przedziałach aby uzyskać wartość oczekiwaną. Wynik końcowy widoczny jest na poniższym wykresie.



Po zastosowaniu testu zarówno dla x jak i dla y uzyskałem wynik pozytywny.

9. TESTY GENERATORÓW

Aby przetestować generatory przepuściłem wszystkie przez test serii oraz sprawdziłem czy na ich podstawie wszystkie generatory innych rozkładów będą działały prawidłowo. Poniżej outputy dla kolejnych generatorów:

-Blum Blum Shub generator

```
C:\Users\Artem\Desktop\rps_projekt\apka\venv\Scripts\python.exe C:/Users/Artem/Desktop/rps_projekt/apka/main.py
We cannot reject the null hypothesis, i.e. the randomness of the sample
The distribution is consistent with the uniform distribution
The distribution is consistent with the Bernoulli distribution
The distribution is consistent with the binomial distribution
The distribution is consistent with the geometric distribution
The distribution is consistent with the Poisson distribution
The distribution is consistent with the exponential distribution
The X distribution is consistent with the natural distribution
The Y distribution is consistent with the natural distribution

Process finished with exit code 0
```

- generator mieszany w wersji Numerical Recipes

```
C:\Users\Artem\Desktop\rps_projekt\apka\venv\Scripts\python.exe C:/Users/Artem/Desktop/rps_projekt/apka/main.py
The distribution is consistent with the uniform distribution
We cannot reject the null hypothesis, i.e. the randomness of the sample
The distribution is consistent with the Bernoulli distribution
The distribution is consistent with the binomial distribution
The distribution is consistent with the geometric distribution
The distribution is consistent with the Poisson distribution
The distribution is not consistent with the exponential distribution
The X distribution is consistent with the natural distribution
The Y distribution is consistent with the natural distribution

Process finished with exit code 0
```

- generator multiplikatywny w wersji APPLE

```
C:\Users\Artem\Desktop\rps_projekt\apka\venv\Scripts\python.exe C:/Users/Artem/Desktop/rps_projekt/apka/main.py
The distribution is not consistent with the uniform distribution
We cannot reject the null hypothesis, i.e. the randomness of the sample
The distribution is consistent with the Bernoulli distribution
The distribution is consistent with the binomial distribution
The distribution is consistent with the geometric distribution
The distribution is consistent with the Poisson distribution
The distribution is not consistent with the exponential distribution
The X distribution is not consistent with the natural distribution
The Y distribution is not consistent with the natural distribution

Process finished with exit code 0
```

- generator mieszany w wersji Microsoft Visual

```
C:\Users\Artem\Desktop\rps_projekt\apka\venv\Scripts\python.exe C:/Users/Artem/Desktop/rps_projekt/apka/main.py
The distribution is consistent with the uniform distribution
We cannot reject the null hypothesis, i.e. the randomness of the sample
The distribution is consistent with the Bernoulli distribution
The distribution is consistent with the binomial distribution
The distribution is consistent with the geometric distribution
The distribution is consistent with the Poisson distribution
The distribution is consistent with the exponential distribution
The X distribution is consistent with the natural distribution
The Y distribution is consistent with the natural distribution

Process finished with exit code 0
```

- generator Marsenne Twister

```
C:\Users\Artem\Desktop\rps_projekt\apka\venv\Scripts\python.exe C:/Users/Artem/Desktop/rps_projekt/apka/main.py
The distribution is consistent with the uniform distribution
We cannot reject the null hypothesis, i.e. the randomness of the sample
The distribution is consistent with the Bernoulli distribution
The distribution is consistent with the binomial distribution
The distribution is consistent with the geometric distribution
The distribution is consistent with the Poisson distribution
The distribution is consistent with the exponential distribution
The X distribution is consistent with the natural distribution
The Y distribution is consistent with the natural distribution

Process finished with exit code 0
```

Jak widać zdecydowanie najlepszymi generatorami okazały Blum Blum Shub oraz Marsenne Twister. Bez problemu przeszły wszystkie testy i dały się przekształcić na generatory innych rozkładów. Niewątpliwą wadą tego drugiego jest za to moim zdaniem przekombinowana budowa. Prawdopodobnie na w innych zastosowaniach niż pisanie projektu na studia takie manewrowanie przesunięciami bitowymi na lewo i prawo da się usprawiedliwić, tak w takich małych projektach myślę że zaimplementowanie małego, przejrzystego generatora jest o wiele bardziej na miejscu. Zwłaszcza, że oba są tak samo skuteczne. Co do generatorów mieszanym to są one jeszcze prostsze niż Blum Blum Shub, głównie ze względu na brak warunków względem a, c oraz m, ale przez to również bardzo łatwo wybrać złe współczynniki. Widać to w powyższych testach, gdzie tylko wersja Microsoftu spełniła pokładane w sobie nadzieje. Ba, nawet generator oparty na współczynnikach z generatorów Apple'a nie wygenerowały rozkładu jednostajnego (najprawdopodobniej jest to kwestia źle wybranego ziarna ale to już przemilczę).

10. BIBLIOGRAFIA

Materiały pomocnicze do wykładów dr. Adama Romana

https://pl.wikipedia.org/wiki/Blum_Blum_Shub

https://pl.wikipedia.org/wiki/Test_zgodności_chi-kwadrat

https://www.itl.nist.gov/div898/handbook/eda/section3/eda35d.htm?fbclid=IwAR1gZRBgo9f0WkLC45yUFUXmejjtBRnUrOFuyOPl2mzUyJ2ttV5JVkC-y1Q

https://pl.wikipedia.org/wiki/Test_serii

http://www.algorytm.org/liczby-pseudolosowe/generator-lcg-liniowy-generator-kongruentny.html

https://pl.wikipedia.org/wiki/Mersenne_Twister

https://pl.wikipedia.org/wiki/Rozkład_zero-jedynkowy

https://pl.wikipedia.org/wiki/Rozkład_dwumianowy

https://pl.wikipedia.org/wiki/Rozkład_Poissona

https://pl.wikipedia.org/wiki/Rozkład_geometryczny

https://pl.wikipedia.org/wiki/Rozkład_Poissona

https://pl.wikipedia.org/wiki/Odwrotna_dystrybuanta

https://pl.wikipedia.org/wiki/Rozkład_wykładniczy

https://pl.wikipedia.org/wiki/Transformacja_Boxa-Mullera

https://pl.wikipedia.org/wiki/Rozkład_normalny