Rachunek prawdopodobieństwa i statystyka – projekt

Temat projektu: Generator liczb pseudolosowych

Wykonawca: Artur Górak

Spis treści

[1. Opis projektu 1](#_Toc74427253)

[2. Generatory 2](#_Toc74427254)

[2.1 Generator Blum Blum Shub 2](#_Toc74427255)

[2.2 Generator mieszany 6](#_Toc74427256)

[2.3 Generator Marsenne Twister 8](#_Toc74427257)

[3. Rozkład Bernoulliego 10](#_Toc74427258)

[4. Rozkład dwumianowy 11](#_Toc74427259)

[5. Rozkład geometryczny 14](#_Toc74427260)

[6. Rozkład Poissona 17](#_Toc74427261)

[7. Rozkład wykładniczy 19](#_Toc74427262)

[8. Rozkład naturalny 21](#_Toc74427263)

[9. Testy generatorów 26](#_Toc74427264)

[10. Bibliografia 28](#_Toc74427265)

# 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:

gdzie to kolejne stany, a 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.

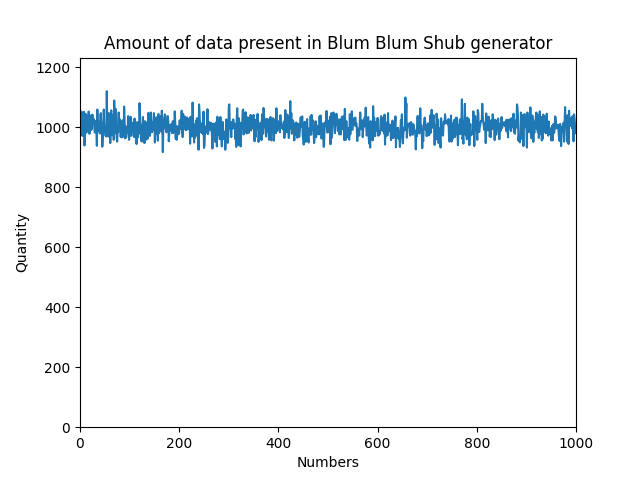
Moja implementacja:

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)

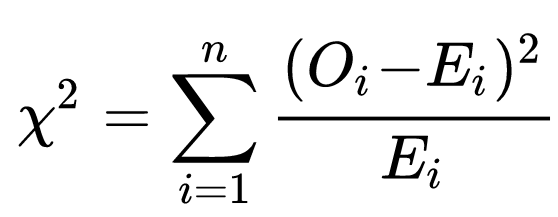
Liczby z przedziału [0, 1] tworzymy przy pomocy modulo każdej z wygenerowanych poprzednio liczb przez gdzie n wybieramy w zależności jak wiele różnych liczb z przedziału chcemy uzyskać. Po czym otrzymane wyniki dzielimy wcześniejsze . Za te działania odpowiada u mnie metoda *generate\_random\_numbers*,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

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 . Następnie boxy, które skorzystałem ze wzoru:



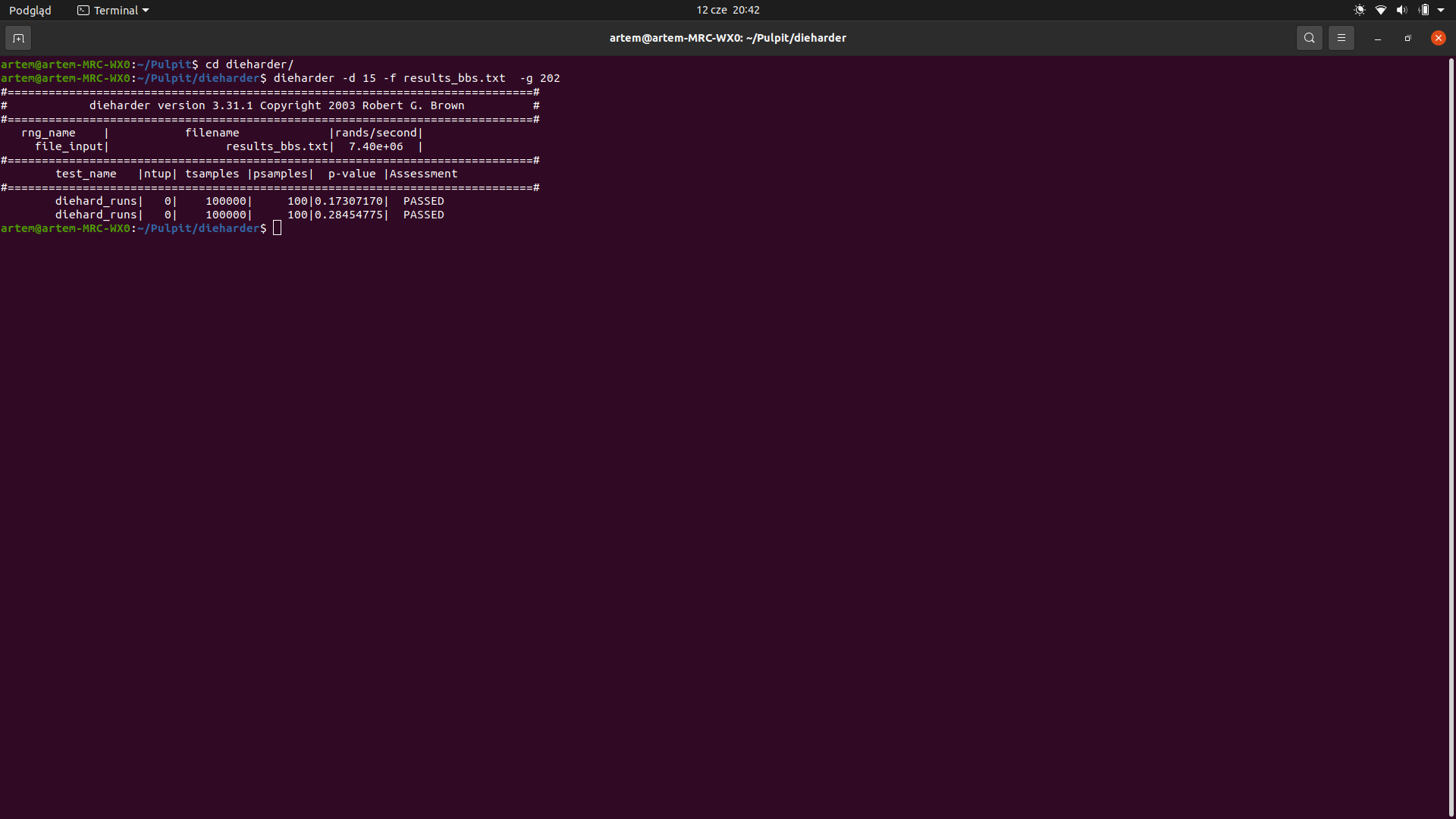
gdzie to ilość elementów, zaś to oczekiwana ilość elementów w i-tym boxie. Jako wartości oczekiwane w każdym boxie brałem 1/7 \* n, gdzie n to ilość wszystkich wygenerowanych liczb. Akceptowałem tylko te boxy do których wpadło więcej niż 5 liczb. Następnie pobieram odpowiednią wartość z tablic rozkładu 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**

**"**)

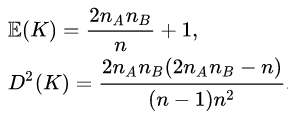
W kolejce do sprawdzenia został tylko sprawdzenie postulatu losowości próby. Zrobiłem to na 2 sposoby. Pierwszym z nich jest wygenerowanie przez generator do pliku tekstowego 1000000000 liczb i skorzystanie z zaproponowanego w treści projektu programu dieharder. Powyższy generator pomyślnie zdał przygotowane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



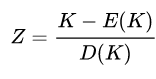
Drugim sposobem jest ręczne zaimplementowanie testu serii, który by sprawdził zgodność z danych z postulatem. U mnie odpowiada za to metoda *runs\_test()*

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:



Później musimy wyliczyć statystykę Z:



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

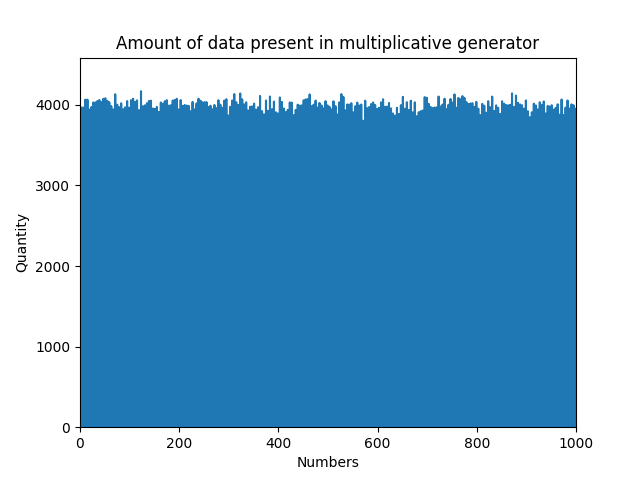
## 2.2 Generator mieszany

Kolejnym generatorem jest generator multiplikatywny dany wzorem:

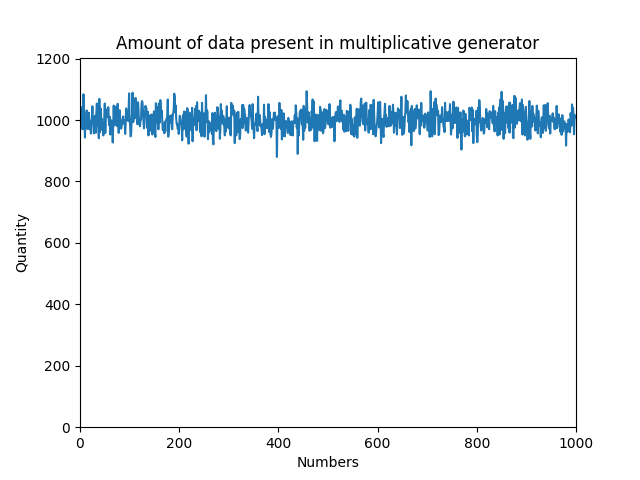
Gdzie my dobieramy odpowiednie a, c oraz m (w sytuacji gdy c = 0 generator nazywany jest multiplikatywnym). W moim przypadku zdecydowałem się przetestować 3 zestawy tych danych proponowane przez różne książki i firmy (Numerical Recipes, APPLE, Microsoft Visual). Poniżej implementacja:

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:

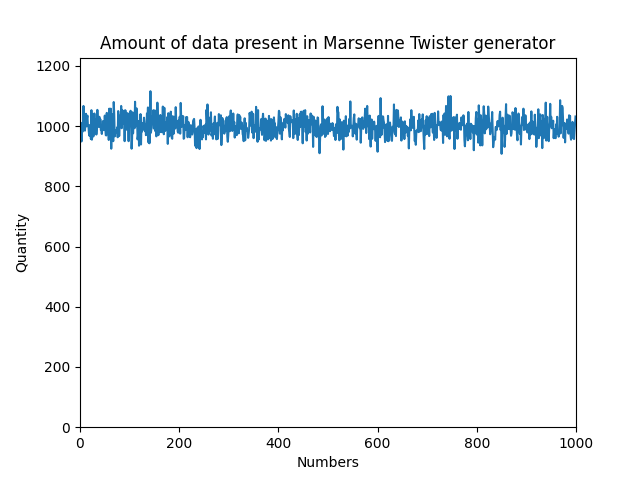


## 2.3 Generator Marsenne Twister

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 . 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.

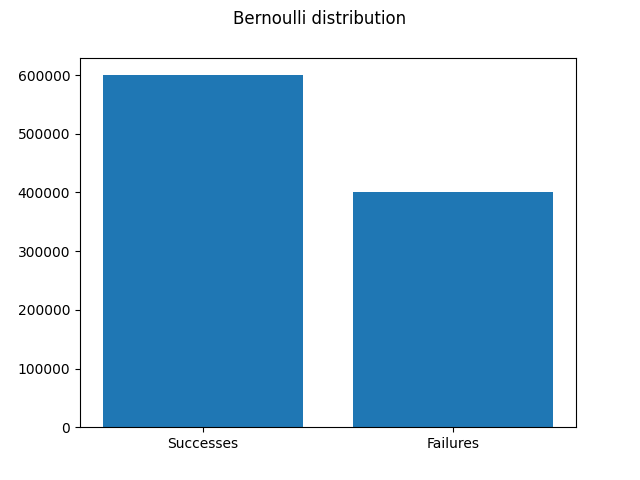


# 3. Rozkład Bernoulliego

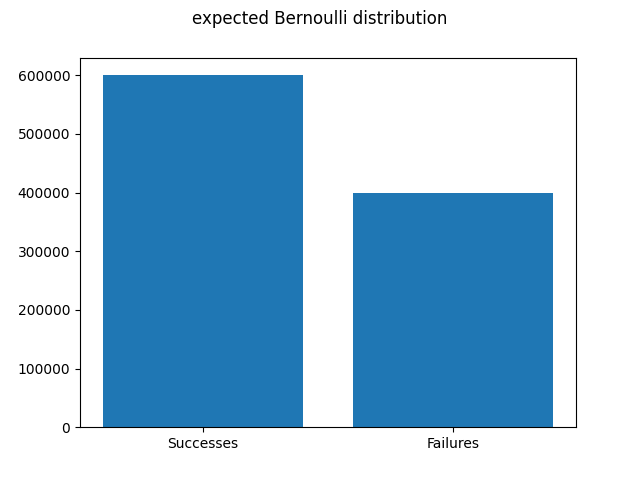
Aby stworzyć rozkład Bernoulliego losuję liczbę z przedziału [0, 1], jeśli jest mniejsza bądź równa danemu prawdopodobieństwu p to odnotowujemy ją jako sukces, w przeciwnym wypadku jako porażkę. Implementacja:

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:



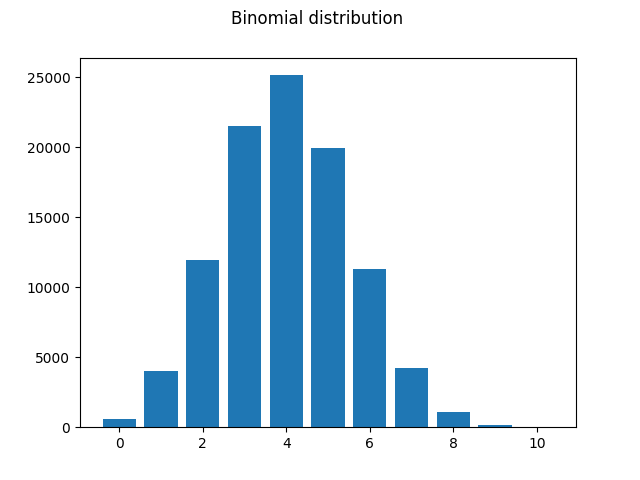
Następnie korzystam z przedstawionego już wcześniej wzoru, pobieram z tablic wartość krytyczną i porównuję. Wygenerowany przeze mnie rozkład zdaje na test na rozkład Bernoulliego.

# 4. Rozkład dwumianowy

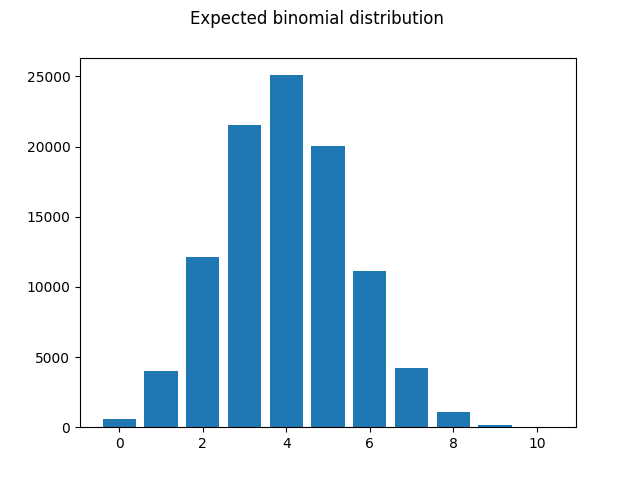
Rozkład bezpośrednio związany z rozkładem Bernoulliego. Aby go utworzyć wykonuję na n liczbach próbę Bernoulliego z prawdopodobieństwem p i zliczam sukcesy. Moja implementacja, gdzie p – prawdopodobieństwo, size – nasze wcześniejsze n, random\_numbers – tablica losowych liczb z przedziału [0,1]:

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:

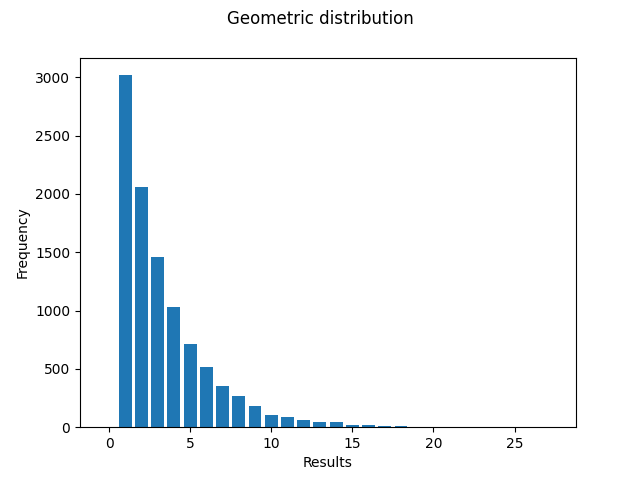
def chi\_square(self):  
 expected = []  
 for x in range(self.size + 1):  
 expected.insert(x, (math.factorial(self.size)/(math.factorial(self.size - x) \* math.factorial(x)) \* math.pow(self.p, x) \* math.pow(1 - self.p, self.size - x))\*(self.rn\_size / self.size))  
  
 plt.bar(self.count, expected)  
 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

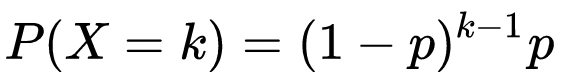
Rozkład jest podobny do rozkładu dwumianowego z tą różnicą, że zamiast ilość sukcesów w n próbach, badamy w którym teście Bernoulliego będzie pierwszy sukces. Implementacja:

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])

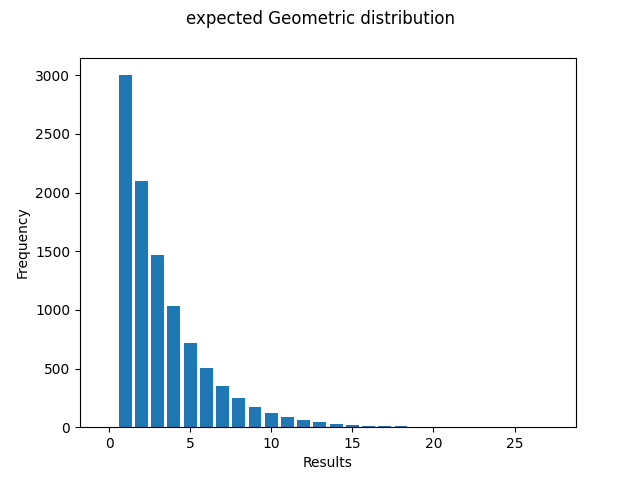
Ostatnie linijki powyższego kodu są odpowiedzialne za skrócenie tablicy poprzez odcięcie pół z zerami na końcu tablicy. Poniżej histogram tego rozkładu dla p = 0.3, n = 10000 oraz how\_many\_numbers = 1000000



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:



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()

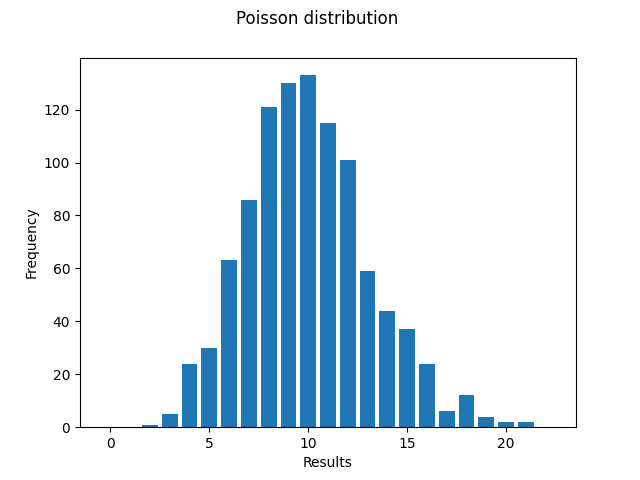
# 6. Rozkład Poissona

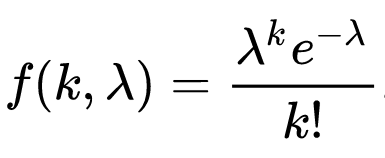
Generator rozkładu Poissona zrobiłem według schematu poznanego na wykładzie i polega on na skorzystaniu z twierdzenia, że x mająca rozkład Poissona dana jest wzorem: . Generacja pojedynczej realizacji zmiennej losowej z rozkładu Poissona o parametrze lambda polega na losowaniu U z rozkładu jednostajnego oraz ustawieniu X na 0 i dopóki 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])

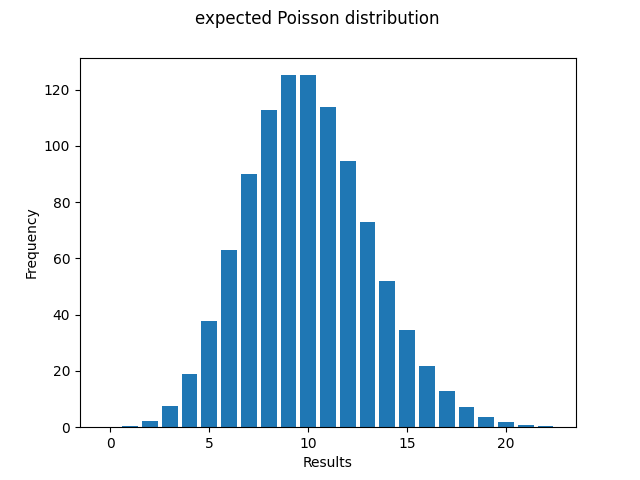
Histogram dla otrzymanego rozkładu Poissona dla lambda = 10 i n = 1000.

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:





Otrzymane wartości widoczne są na poniższym histogramie.

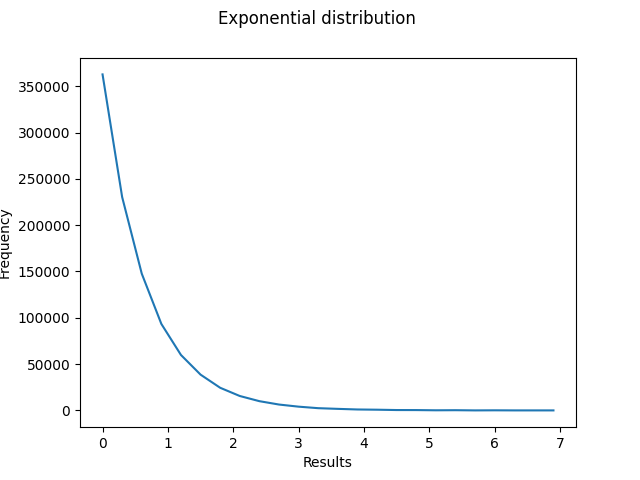


Test po wykonaniu go na tym rozkładzie daje wynik pozytywny.

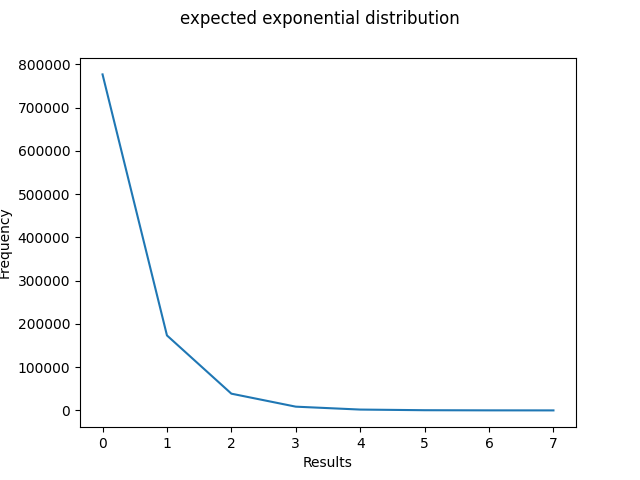
# 7. Rozkład wykładniczy

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



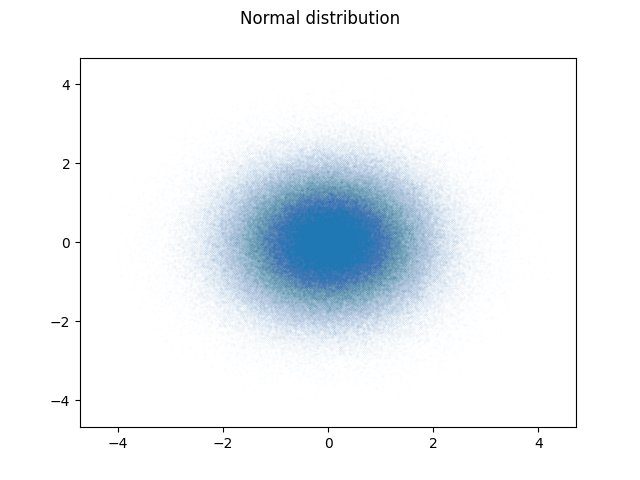
Aby móc skorzystać z testu chi kwadrat podzieliłem oś x na 7 przedziałów i celu wyliczenia wartości oczekiwanej dla każdego z nich scałkowałem na danym przedziale wzór na gęstość prawdopodobieństwa rozkładu wykładniczego . W tym celu skorzystałem z funkcji squad z biblioteki scipy. Następnie analogicznie 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 oraz o rozkładzie normalnym i odchyleniu standardowym 1. Niech i będą niezależnymi zmiennymi losowym o rozkładzie jednostajnym na przedziale (0, 1]. Wówczas:

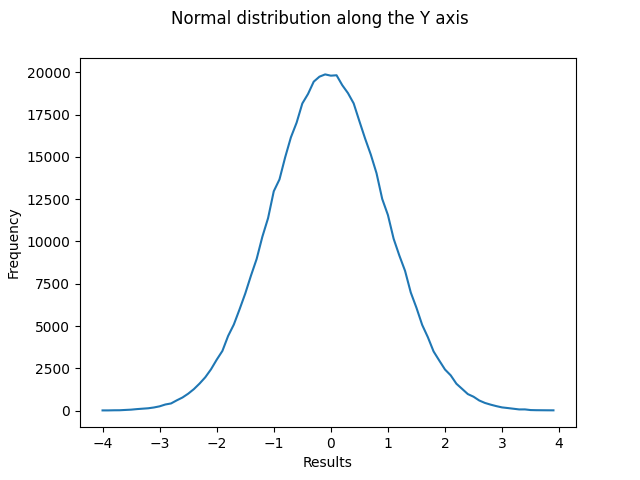
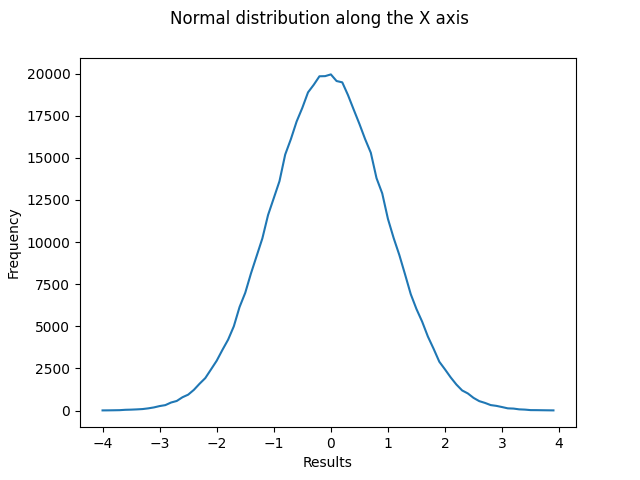
Traktując oraz jako współrzędne x i y na układzie współrzędnych otrzymamy wówczas:



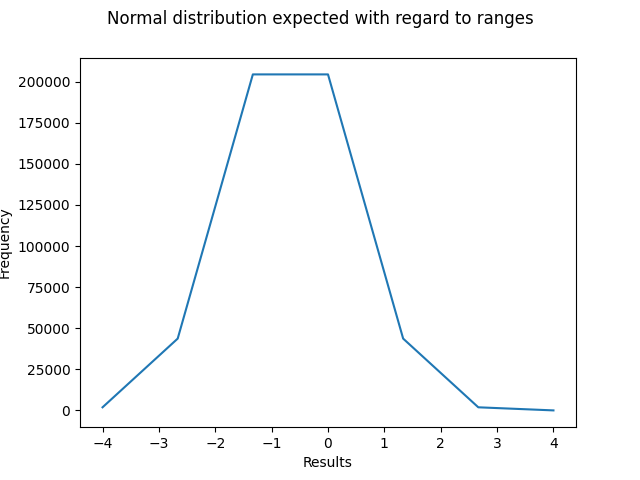
Implementacja wygląda następująco:

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.

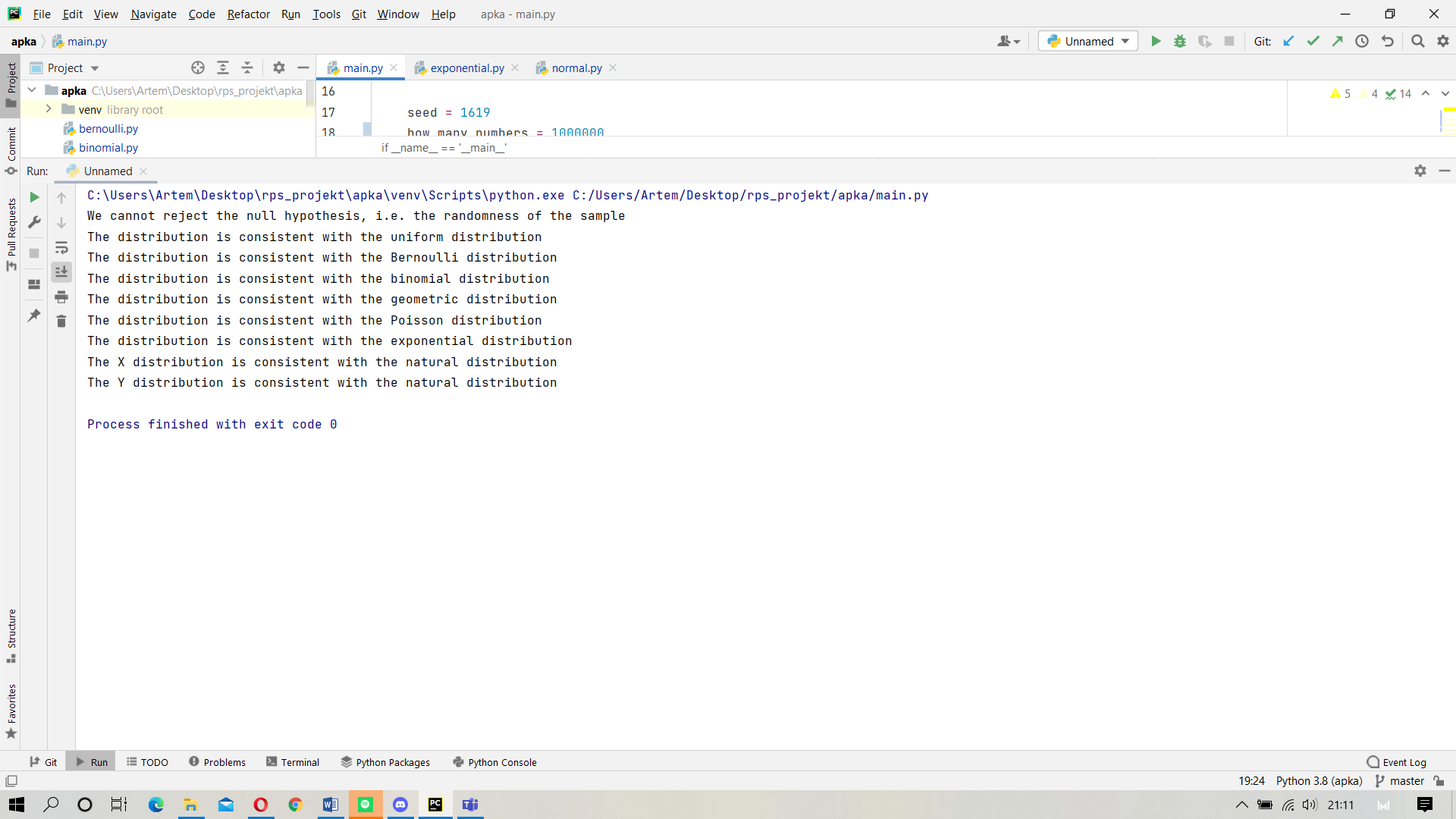


Po zastosowaniu testu zarówno dla x jak i dla y uzyskałem wynik pozytywny.

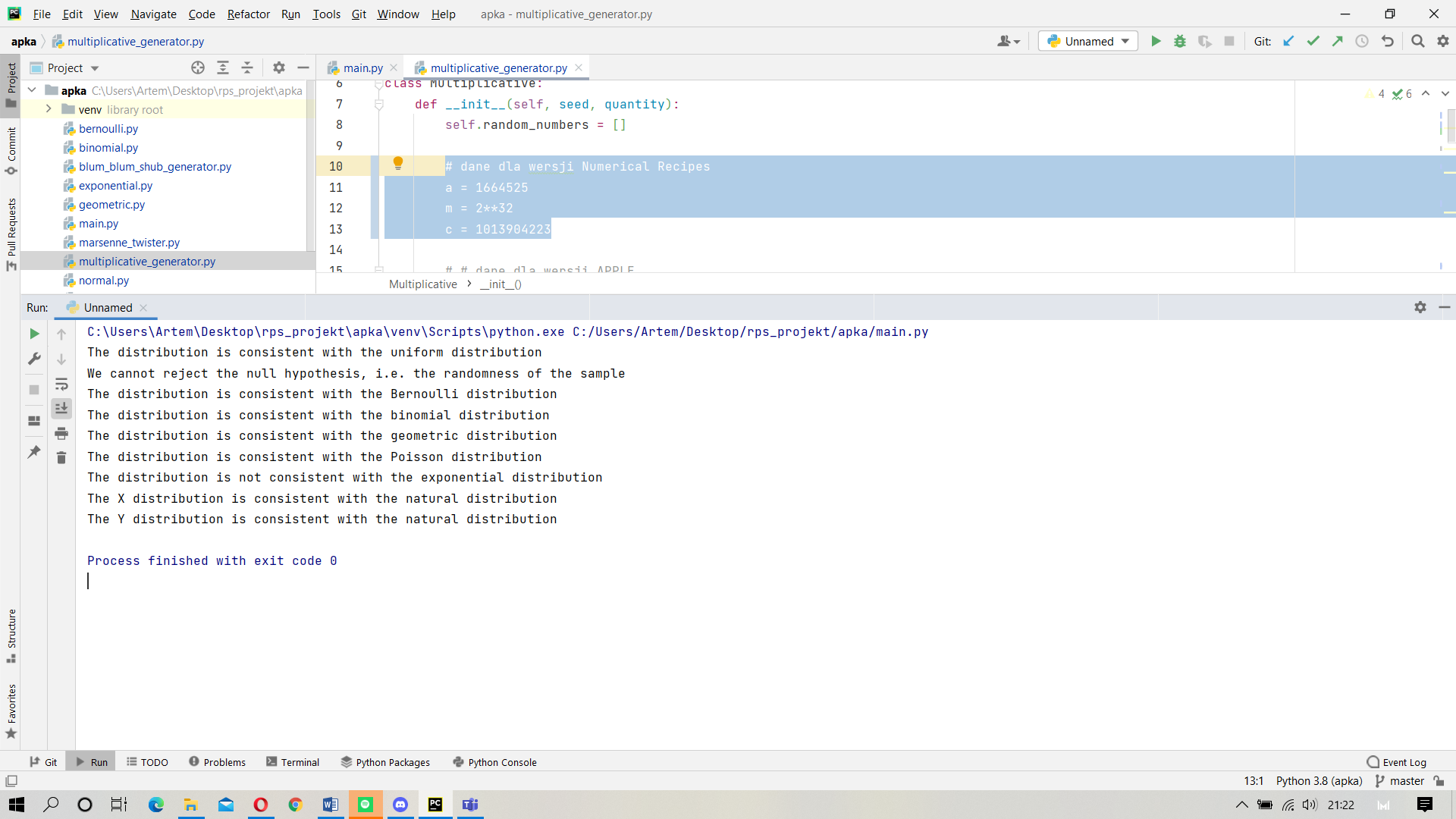
# 9. Testy generatorów

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:

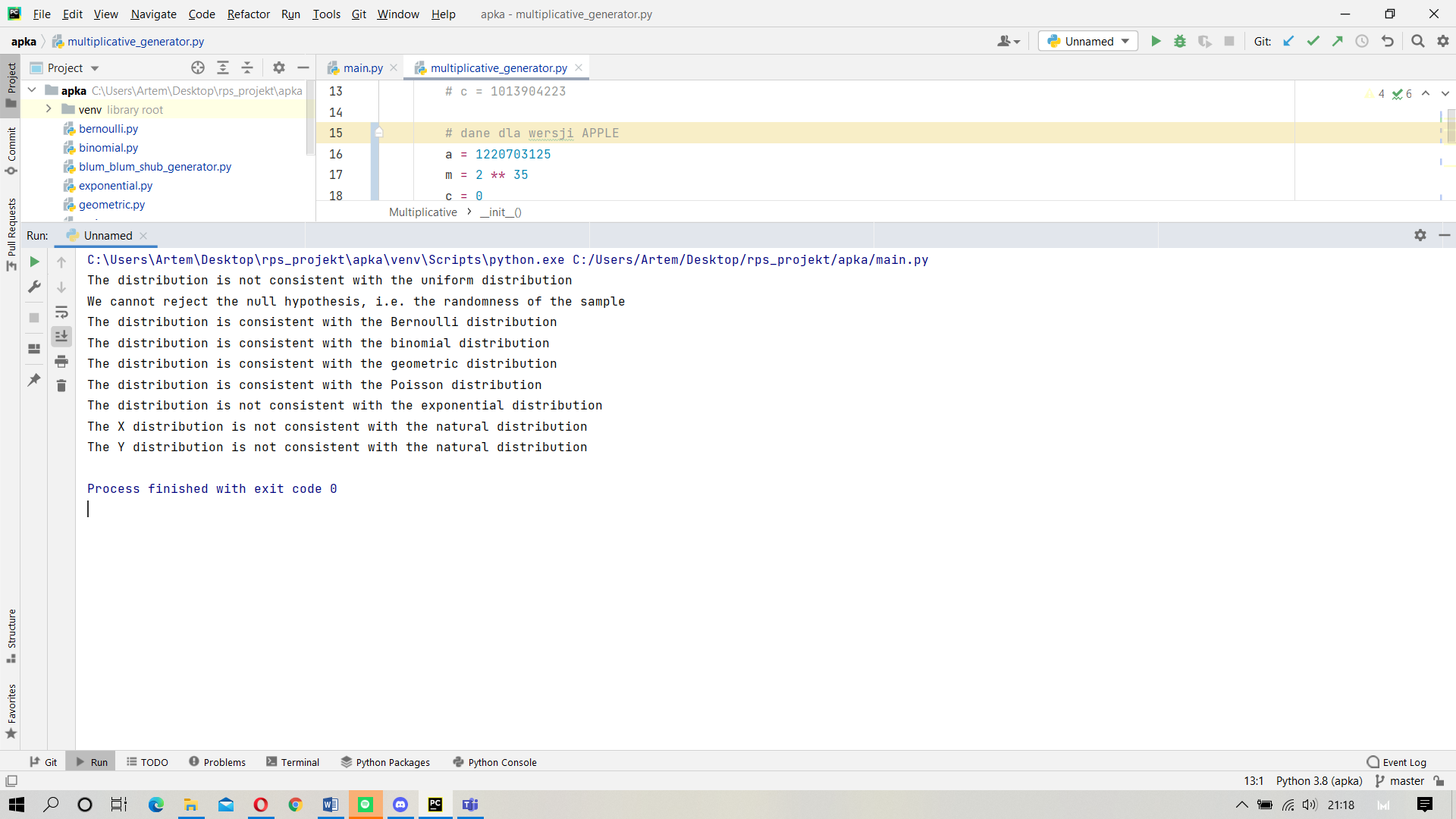
-Blum Blum Shub generator



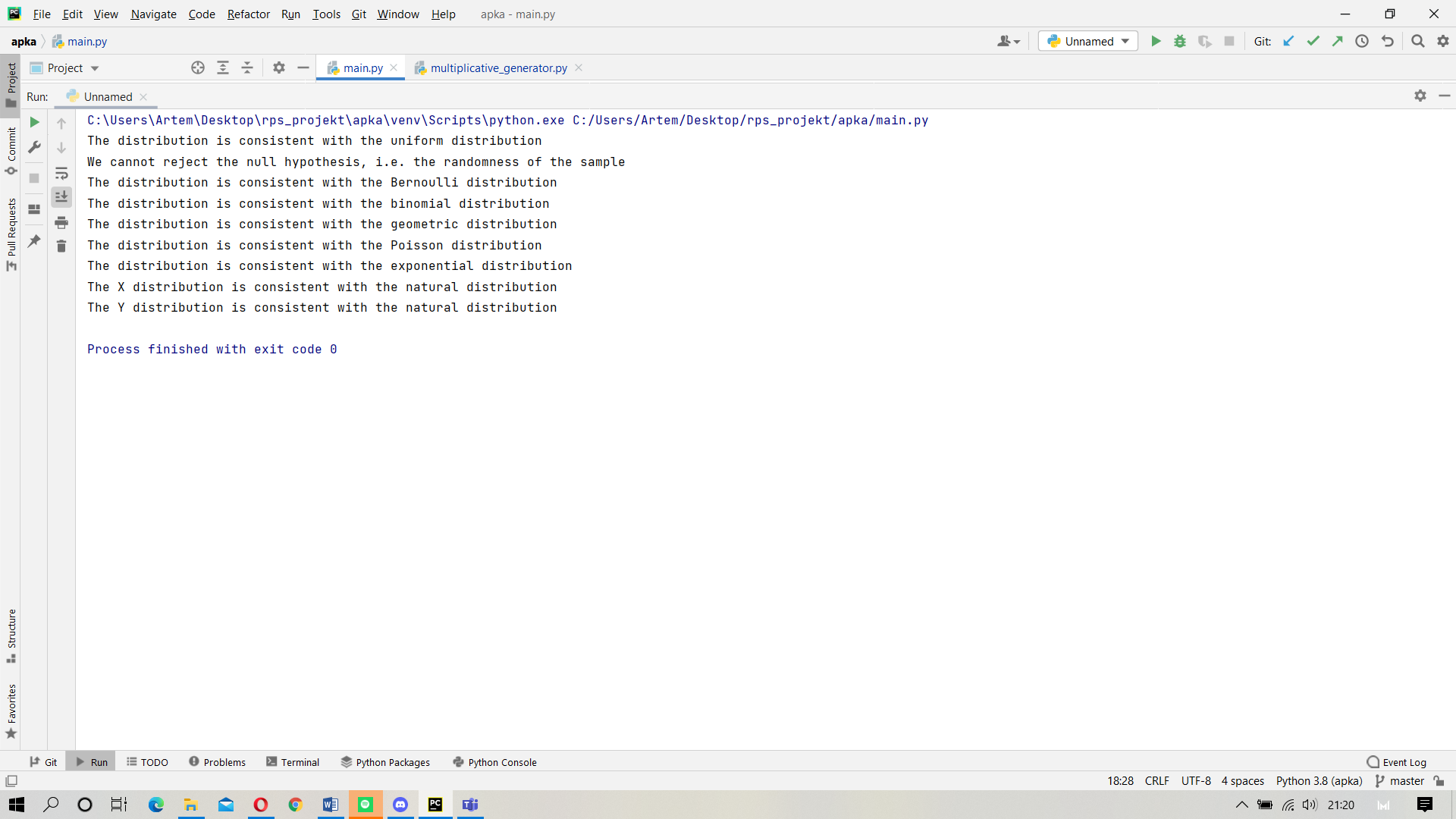
- generator mieszany w wersji Numerical Recipes



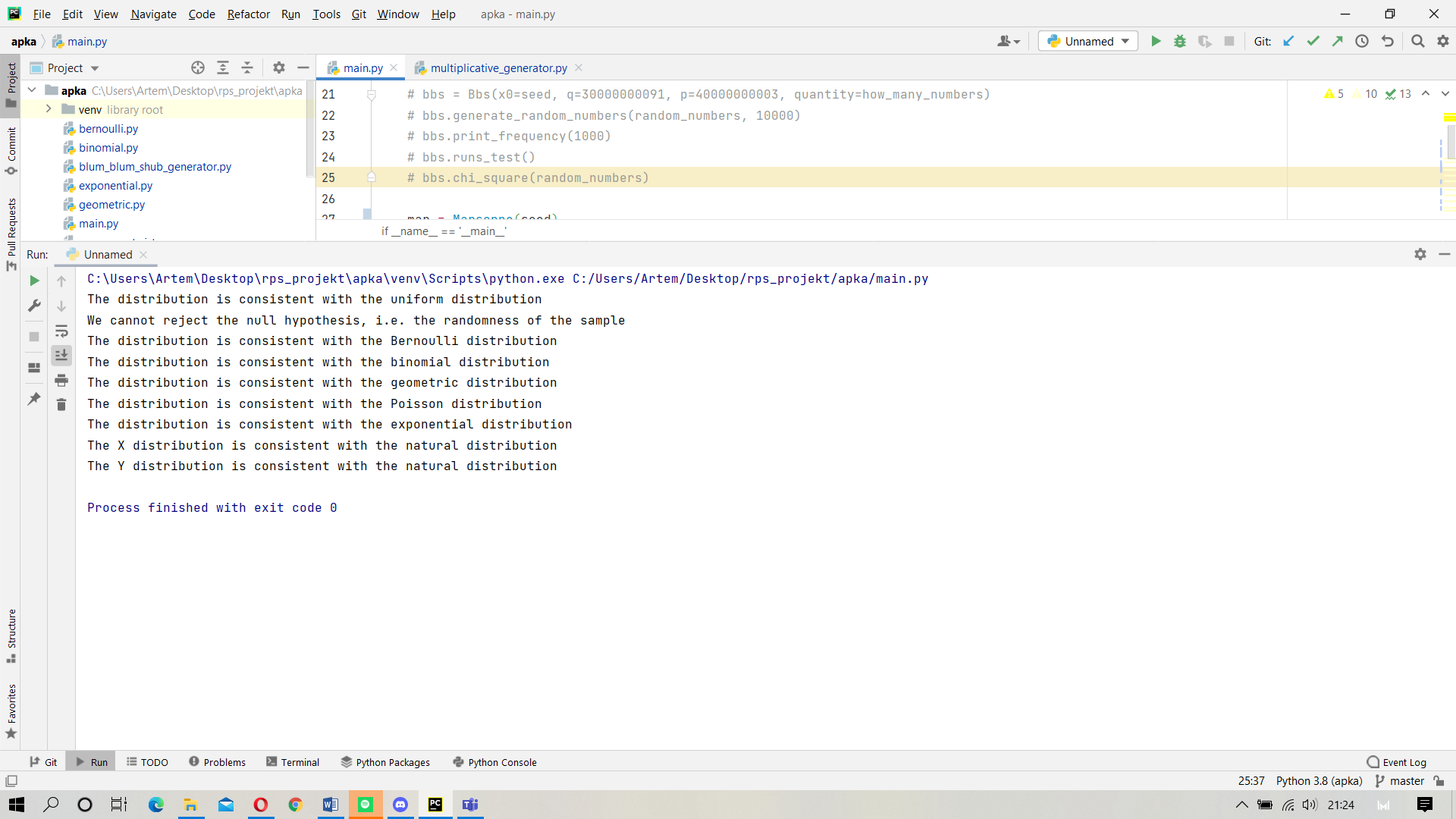
- generator multiplikatywny w wersji APPLE



- generator mieszany w wersji Microsoft Visual



- generator Marsenne Twister



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