Random variable generation

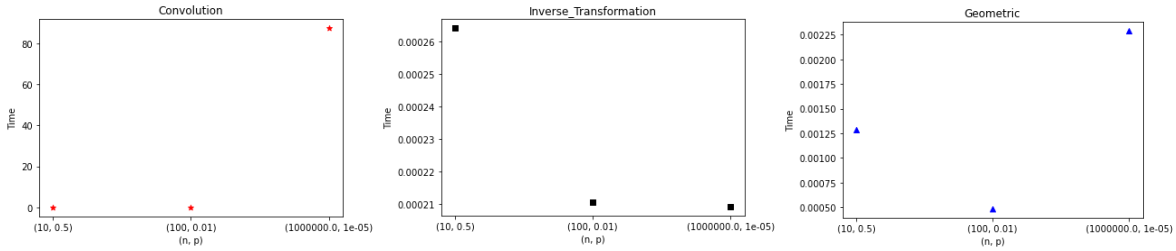
Simulators require generators of random variables with specific distributions. To so this, we use three different routines to generate Bin(n,p) random variables with three different approaches, and just one routine to generate normal random variables by adopting an acceptance/rejection method.

Regarding Bin(n,p), the input parameters are number of trials (n), probability (p) and number of generated r.v. (k). Furthermore, there exist three cases (n=10, p=0.5), (n=100, p=0.01) and (n=106 , p=10-) in which the efficiency (the time needed to generate k instances of the r.v.) of methods are compared.

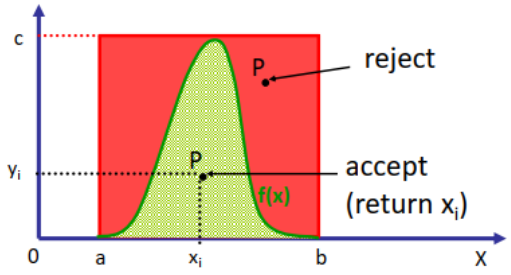
The methods are as follows:

1. *Convolution*: the idea is to produce r.v. based on counting how many numbers are less than probability p. These numbers are considered as successful trials.
2. *Inverse_Transform*: this method computes the cdf of binomial distribution and then returns a random uniform number u(0,1) to those value in way that the number lies between a consecutive support [cdf(i), cdf(i+1)].
3. *Geometric*: It is based on the observation that a geometric with parameter (1-p) assuming the value i+1 corresponds to i failed Bernoulli experiments.

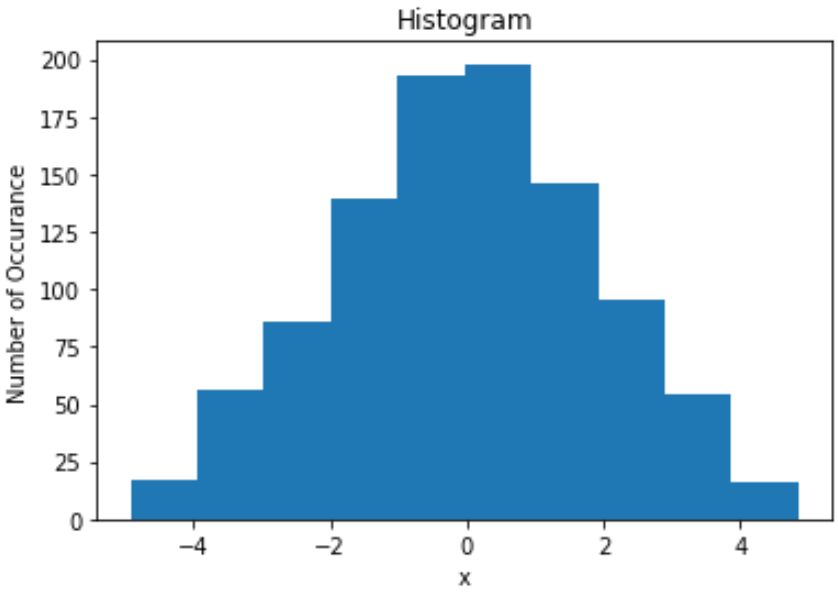
After generating 100 r.v. by each of these methods, it can be seen that as the number of trials increase, the required time for convolution method increase but the differences for the two others are quite small.

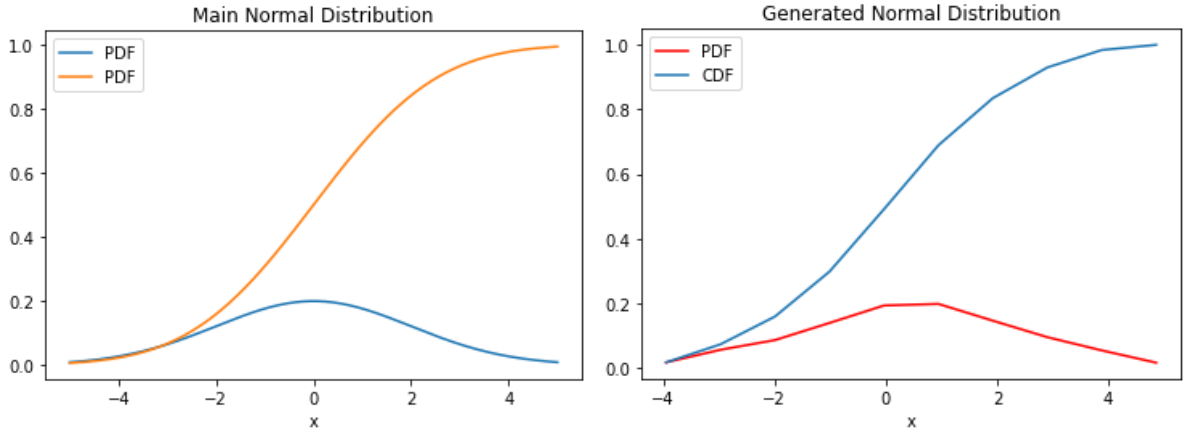


Relating normal r.v., there is a support in which we produce the numbers as our initial r.v. to generate pdf and cdf of the main normal distribution. We use acceptance/rejection method since it can be applied to random variables with continuous pdf f(x) defined over finite support [a,b] as the normal distribution. Considering figure below, the idea is that we just accept P if it is inside the green area.



Using a normal distribution with mean=0 and var=4, we obtain a respectable generator as the results show with generated mean =-0.0086 and variance=.6077.





The related code is provided below.

Main_Binomial_Normal_Distribution

November 2, 2022

[5]:
```
from Binomial_Inverse_Transform import inv_trans
from Binomial_Convolution import conv
from Binomial_Geometric import geo
import random
import timeit
import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import norm

random.seed(42)
```

1 Bin(n,p) random variables generation

[6]:
```
# initializition of input parameters

n = [10, 100, 1e6]
p = [0.5, .01, 1e-5]
#n = [10]
#p = [0.5]
k = 100
```

[7]:
```
# implementation of convolution method to generate k instances of r.v.
conv_dic = {}

for i in range(len(n)):

    start = timeit.default_timer()

    for j in range(k):
        x_conv = conv(n[i], p[i])

    stop = timeit.default_timer()
    conv_dic[str((n[i], p[i]))] = (stop - start)
```

[8]:
```
conv_dic.keys(), conv_dic.values()
```

[8]:
```
(dict_keys(['(10, 0.5)', '(100, 0.01)', '(1000000.0, 1e-05)']),
 dict_values([0.00121459999999729, 0.011000699999996755, 87.27229930000001]))
```

[9]:
```
# implementation of inverse-transform method to generate k instances of r.v.
inv_dic = {}

for i in range(len(n)):

    start = timeit.default_timer()

    for j in range(k):
        x_inv_trans = inv_trans(n[i], p[i])

    stop = timeit.default_timer()
    inv_dic[str((n[i], p[i]))] = (stop - start)
```

[10]:
```
inv_dic.keys(), inv_dic.values()
```

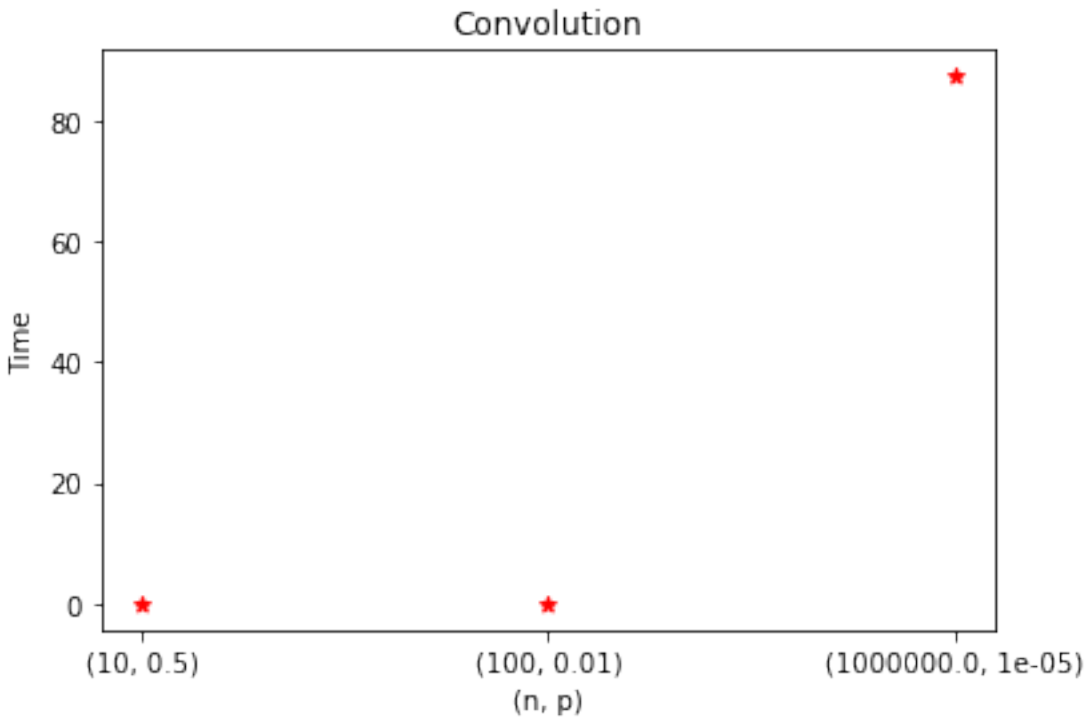
[10]:
```
(dict_keys(['(10, 0.5)', '(100, 0.01)', '(1000000.0, 1e-05)']),
 dict_values([0.0002640000000155851, 0.00021069999999667743,
0.00020930000002294946]))
```

[11]:
```
# implementation of geometric method to generate k instances of r.v.
geo_dic = {}


for i in range(len(n)):

    start = timeit.default_timer()

    for j in range(k):
        x_geo = geo(n[i], p[i])

    stop = timeit.default_timer()
    geo_dic[str((n[i], p[i]))] = (stop - start)
```

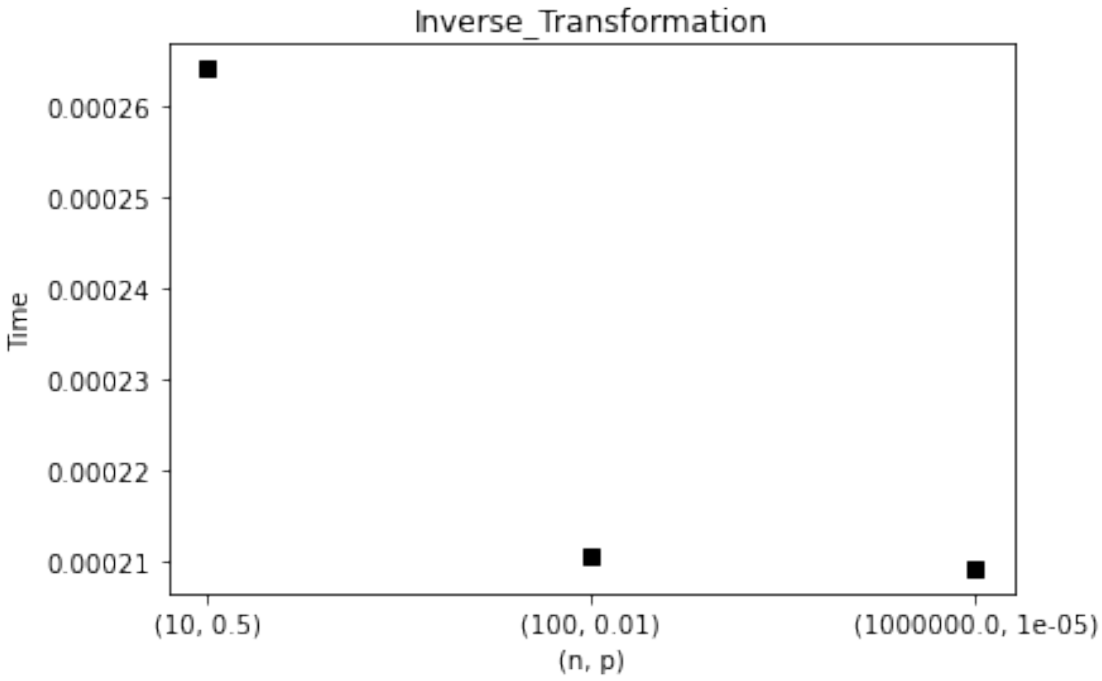
[12]:
```
geo_dic.keys(), geo_dic.values()
```

[12]:
```
(dict_keys(['(10, 0.5)', '(100, 0.01)', '(1000000.0, 1e-05)']),
 dict_values([0.001284400000002961, 0.0004840999999999421416,
0.0022873000000060983]))
```

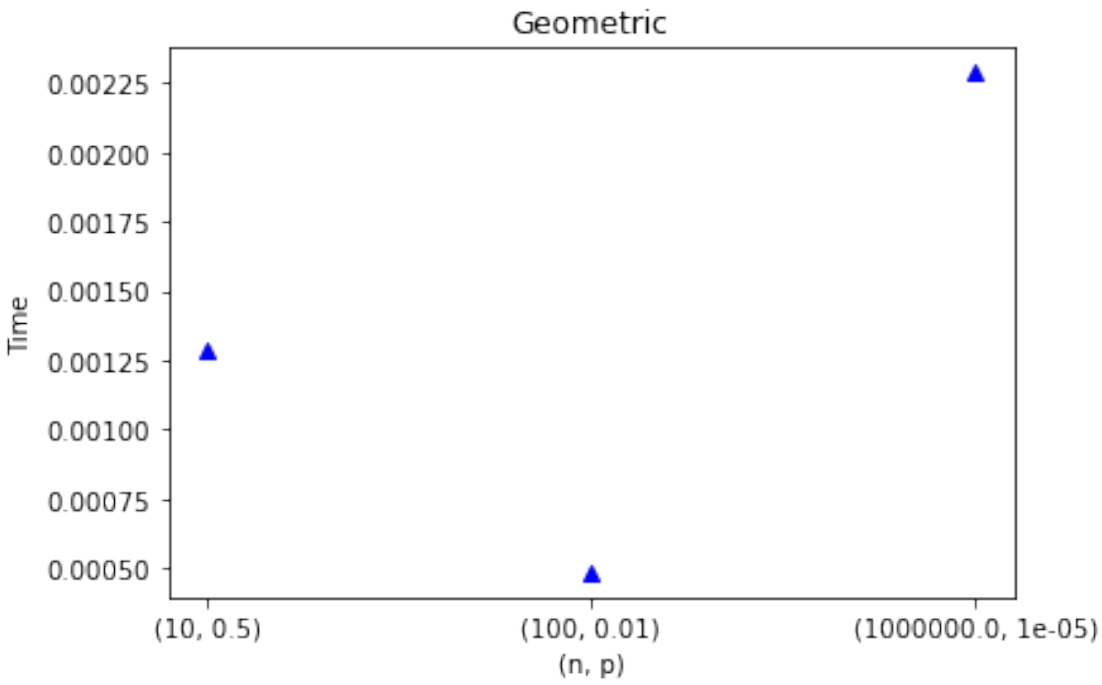
[13]:
```
plt.scatter(list(conv_dic.keys()), list(conv_dic.values()), c='red', marker='*')
plt.xlabel('(n, p)')
plt.ylabel('Time')
plt.title('Convolution')
plt.show()
```



[14]:
```
plt.scatter(list(inv_dic.keys()), list(inv_dic.values()), c='black', marker='s')
plt.xlabel('(n, p)')
plt.ylabel('Time')
plt.title('Inverse_Transformation')
plt.show()
```



[15]:
```
plt.scatter(list(geo_dic.keys()), list(geo_dic.values()), c='blue', marker='^')
plt.xlabel('(n, p)')
plt.ylabel('Time')
plt.title('Geometric')
plt.show()
```



2 Normal random variables generation

[21]:
```
# defining function to generate normal r.v.

def normal(snd, x_lin):

    c = max(snd.pdf(x_lin))

    for i in range(len(x_lin)):

        x = random.uniform(a, b)
        y = random.uniform(0, c)

        if y <= snd.pdf(x):
            return x
```

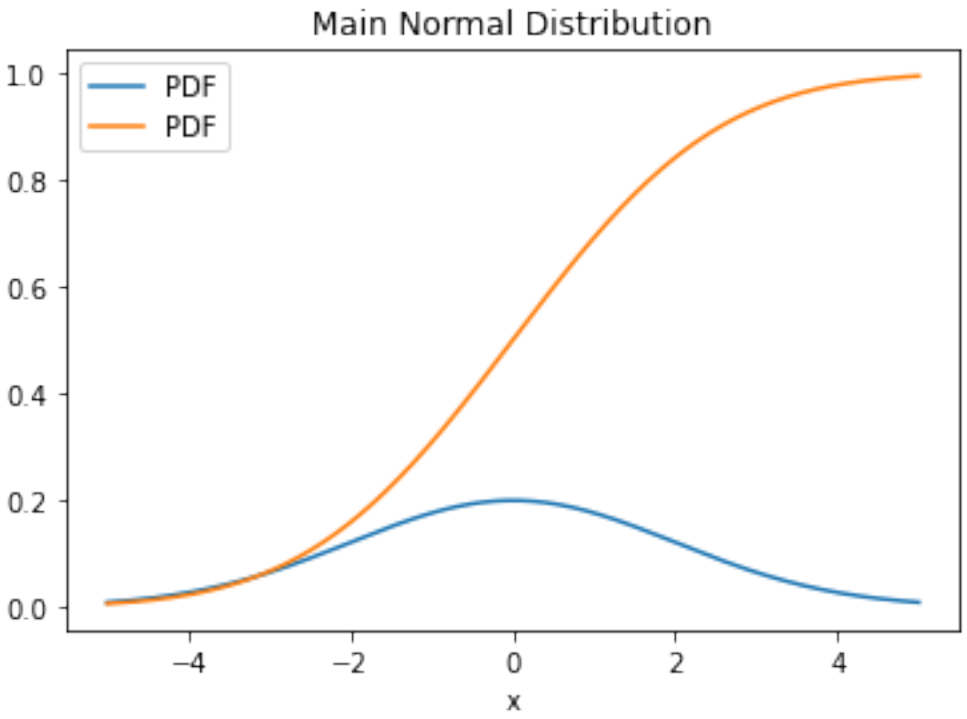
[22]:
```
# parameters initialization
num = 1000
rv_list = []

a = -5
b = 5
```

```
# computing mean and standard deviation based on the parameters
mean = (a + b)/2
std = 2

# generating numbers over support [a,b] and normal distribution
x_lin = np.linspace(a, b, num)
snd = norm(mean, std)
f = snd.pdf(x_lin)

# generating a list of normal r.v.
for i in range(num):
    rv_list.append(normal(snd, x_lin))
```

Plotting pdf and cdf of two distributions

[27]:
```
# the main normal distribution

m, v = snd.stats(moments='mv')
print(f'mean is {m} and variance is {v}')
plt.plot(x_lin, f, label='PDF')
plt.plot(x_lin, snd.cdf(x_lin), label='PDF')
plt.legend()
plt.title('Main Normal Distribution')
plt.xlabel('x')
plt.show()
```

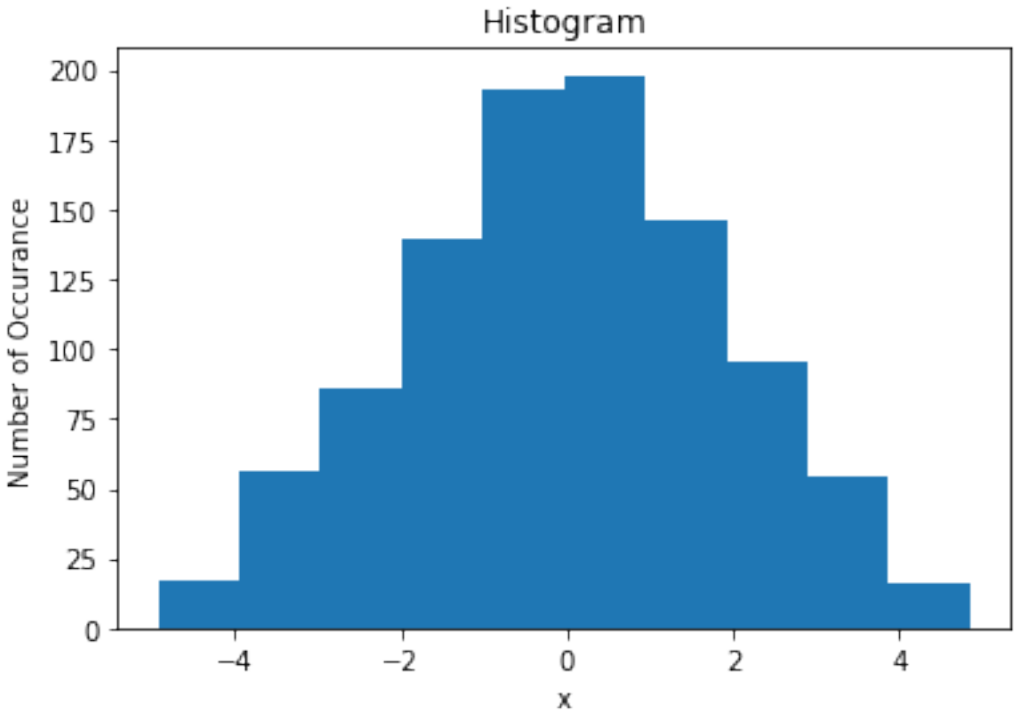
```
mean is 0.0 and variance is 4.0
```



[25]:
```
# histogram of generated r.v. distribution
rv_mean = sum(rv_list)/len(rv_list)
rv_var = np.var(rv_list)
print(f'r.v. mean is {rv_mean} and r.v. variance is {rv_var}')

plt.hist(rv_list)
plt.ylabel('Number of Occurance')
plt.xlabel('x')
plt.title('Histogram')
```

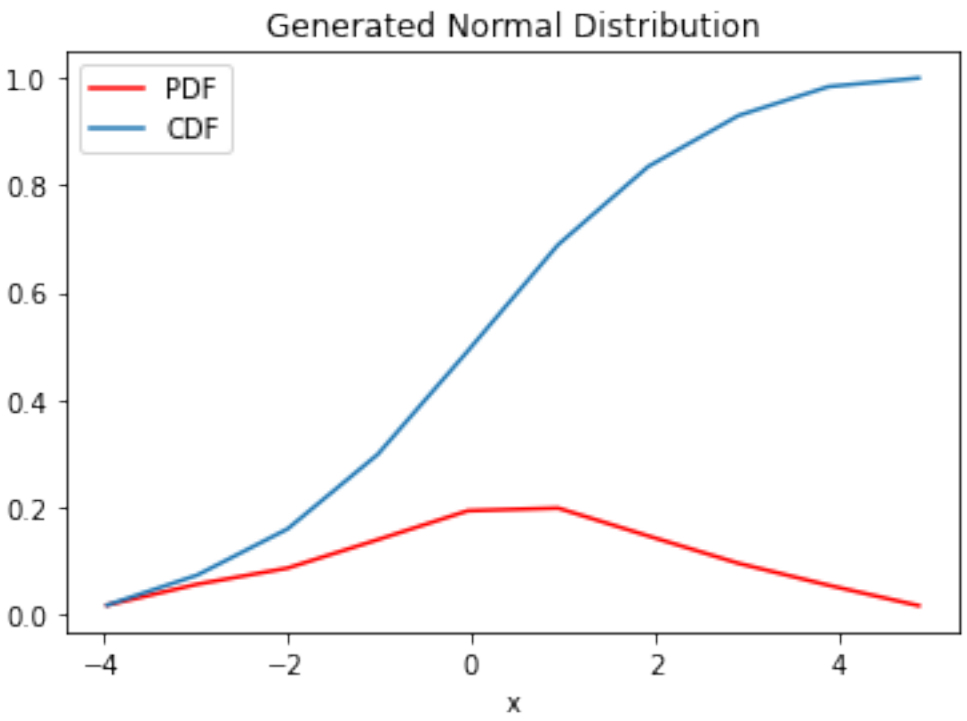
```
r.v. mean is -0.00860546682337297 and r.v. variance is 3.6077580627353116
```

[25]:
```
Text(0.5, 1.0, 'Histogram')
```



[26]:
```
count, bins_count = np.histogram(rv_list)

# finding the PDF of the histogram using count values
pdf = count / sum(count)

# using numpy np.cumsum to calculate the CDF
# We can also find using the PDF values by looping and adding
cdf = np.cumsum(pdf)

# plotting PDF and CDF
plt.plot(bins_count[1:], pdf, color="red", label="PDF")
plt.plot(bins_count[1:], cdf, label="CDF")
plt.legend()
plt.title('Generated Normal Distribution')
plt.xlabel('x')
```

[26]:
```
Text(0.5, 0, 'x')
```


Generated Normal Distribution
PDF
CDF
1.0
0.8
0.6
0.4
0.2
0.0
−4
−2
0
2
4
x

[ ]: