Simulation

Assingnment 1.2. - Random number generation

Ari Viitala

Implement the acceptance/rejection method for generating random variates from the half-normal distribution

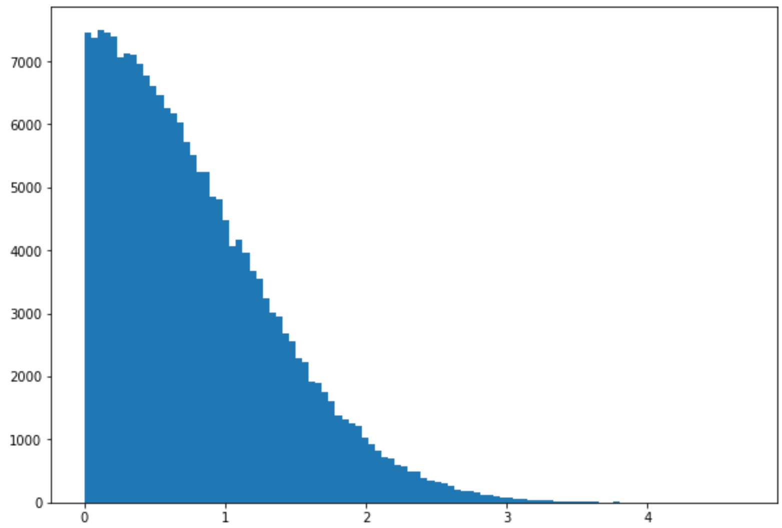
In [4]:
```
import numpy as np
import matplotlib.pyplot as plt
```

In [5]:
```
def halfNormal():
    #from the lecture slides
    c = np.sqrt(2 * np.e / np.pi)

    #the counter for tries needed for viable number. At least one is needed.
    needed = 1

    while True:
        #sample number from g(x, lambda = 1)
        y = np.random.exponential(1)
        #sample numeber from u(0,1)
        u = np.random.random()
        #calculate f(x)
        f = 2 / np.sqrt(2 * np.pi) * np.exp(-y**2 / 2)
        #calculate g(x)
        g = np.exp(-y)

        if u <= f/(g*c):
            #if number is viable return it and the number of tries
            return (y, needed)
        else:
            #else increase the tries and loop again
            needed += 1
```

In [9]:
```
#making the vectors to store the numbers
halfNormals = []
rejections = []

#drawing 200000 numbers from the half normal distribution and storing the values and the amount of tries needed
for i in range(0,200000):
    number = halfNormal()
    halfNormals.append(number[0])
    rejections.append(number[1])
```

In [10]:
```
%matplotlib inline
```

In [11]:
```
plt.figure(1, (10,7))
plt.hist(halfNormals, bins = 100)
plt.show()
```



Based on the histogram, it seems like the numbers are sampled from a half normal distribution.

Observe the average number of variates X that is needed to produce one accepted random variate Y and compare it to the value C

In [12]:
```
#The average number of tries needed on average
np.mean(rejections)
```

Out[12]: 1.3172900000000001

In [14]:
```
#value of C
c = np.sqrt(2 * np.e / np.pi)
c
```

Out[14]: 1.3154892469589139

The value c and the average number of tries needed to get a number that is distributed like we want it be is pretty much the same. This means that the smaller the largest distance between the two density functions the smaller the expected tries to get a viable number hence shorter the runtime. So for efficient generating algorithm we should choose function g(x) that is as close to f(x) as possible.