The report of Coputer-aided simulations lab

Lab6

QI AN s288400

Date: Otc. 12th 2022

Part 1 The introduction

1.1.The Introduction

(1). Learn what is the Binomial distribution;
(2). Learn what is the Acceptance/Rejection Technique
(3). Learn the recommended part what is Rice distribution;

1.2.The principles

1.2.1.Binomial distribution

In probability theory and statistics, the binomial distribution with parameters n and p is the discrete probability distribution of the number of successes in a sequence of n independent experiments, each asking a yes–no question, and each with its own Boolean-valued outcome: success (with probability p) or failure (with probability {display style q=1-p}q=1-p). A single success/failure experiment is also called a Bernoulli trial or Bernoulli experiment, and a sequence of outcomes is called a Bernoulli process; for a single trial, i.e., n = 1, the binomial distribution is a Bernoulli distribution. The binomial distribution is the basis for the popular binomial test of statistical significance.

1.2.2. Acceptance/Rejection Technique

The Accept-Reject method is a classical sampling method which allows one to sample from a distribution which is difficult or impossible to simulate by an inverse transformation. Instead, draws are taken from an instrumental density and accepted with a carefully chosen probability.

The function is that the acceptance/rejection technique can be applied to random variables with continuous pdf f(x) defined over finite support [a,b] n Being c the maximum value for f(x), we apply the following procedure:

1. Generate xi =U(a,b), uniform in [a,b]
2. Generate yi =U(0,c), uniform in [0,c]
3. If yi ≤f(xi) return xi , otherwise go back to step 1

1.2.3.

In probability theory, the Rice distribution or Rician distribution (or, less commonly, Ricean distribution) is the probability distribution of the magnitude of a circularly-symmetric bivariate normal random variable, possibly with non-zero mean (noncentral).

The function is:

$$f(x \mid \nu, \sigma) = \frac{x}{\sigma^2} \exp\left(\frac{-(x^2+\nu^2)}{2\sigma^2}\right) I_0\left(\frac{x\nu}{\sigma^2}\right), \tag{1}$$

In this project, I am going to be simulating the 3 different generation method of Binomial distribution, the plot of Acceptance/Rejection Technique figure and the generation of Rice distribution.

1.3.Tool and platform

Program language: Python 3.8
Tool: the Google-Colaboratory for coding;
Text: Microsoft word

Part 2 The procedure

2.1 Code Explanation

Import lab:

```
#import the lib
from collections import Counter
import random
import numpy as np
import math
```

Make the generationMethod1 funciton and test

```
def generationMethod_1(n,p):
    x = 0
    y = 0
    for i in range(n):
        # generate the random variable between 0,1
        u = np.random.random()
        # count the num of <p or >p
        if u < p:
            x = x+1
        if u > p:
            y = y+1
    return x
```

```
#test generationMethod1
print(generationMethod_1(1000,0.5))
```

```
495
```

Make the fact function and the binomial distribution function, in the last, I made a the method 2 generation and test.

```
#the fact function
def fact_fun(n):
    if n == 0:
        return 1
    n += 1
    fact_list = [i for i in range(1,n)]
    a = 1
    for i in fact_list:
        a = a*i
    return a
```

```
#the function for f(x) of binomial distribution
def fBinomialDistribution(n,x,p):
        #1.calculate f(x) and store in vectorA
        #calculate c(n,x)
        fact_n = fact_fun(n)
        fact_x = fact_fun(x)
        fact_n_x = fact_fun(n - x)
        c_n_x_num = fact_n / (fact_x * fact_n_x)

        pi = (p ** x) * ((1 - p) ** (n - x))
        binomial_num = c_n_x_num * pi
        return binomial_num
```

```python
#the  generationMethod2
def  generationMethod_2(n,x,p):
    vA  =  []
    for  i  in  range(n):
        a  =  fBinomialDistribution(n,i,p)
        vA.append(a)
    #  sort  the  vA
    vA.sort()
    #  generate  the  random  variable  between  0,1
    u  =  np.random.random()
    xRe  =  0
    #serch  the  x  positon  in  the  list
    for  j  in  vA:
        if  u  >=  j:
            xRe  =  j
        else:
            break
    return  xRe
```

```python
#test  generationMethod2
print(generationMethod_2(1000,400,0.3))
```

```
0.02752100382126686
```

Make the 3rd method generation

```
def generationMethod_3(n,p):
    # initialization
    m = 1
    q = 0
    while(True):
        #um = U(0,1)
        um = np.random.random()
        #generate the geometric distribution
        gm = math.ceil(math.log(um)/math.log(1-p))
        q = q+gm
        if q>n:
            return m-1
            break
        else:
            m+=1
```

```
#test generationMethod_3
print(generationMethod_3(1000,0.5))
```

```
491
```

Make the plot of acceptance and rejection

```
import matplotlib.pyplot as plt
import seaborn as sns
#acceptance and rejection
#xi = U(a,b)
#continuous pdf f(x) defined over finite support [a,b]
#c is the maximum value for f(x)
def plot_AorR(a,b,c,x,y):
    px = np.arange(a,1+b,0.01)
    py = c

    fig,ax = plt.subplots()
    temp = ax.hist(x,y,density=True)
    ax.plot(px,py)
    plt.title("acceptance and rejection plot")
    plt.show()
```

Make a class of rice function

```python
class RiceDistribution:
    # the number of samples used in the simulation
    r = np.linspace(0, 6, 6000) # theoretical envelope PDF x axes
    theta = np.linspace(-np.pi, np.pi, 6000)      # theoretical phase PDF x axes

    def __init__(self,K,r_hat_2,phi,numSamples):

        # # user input checks and assigns value
        self.K = K
        self.r_hat_2 = r_hat_2
        self.phi = phi
        self.numSamples = numSamples
        # user input checks and assigns value

        # simulating and their densities
        self.multipathFading = self.multipath_Fading()
        self.xOutcomeEnv, self.yOutcomeEnv = self.envelope_Density()
        self.xOutcomePh, self.yOutcomePh = self.phase_Density()

        # theoretical PDFs calculated
        self.envelopeProbability = self.envelope_PDF()
        self.phaseProbability = self.phase_PDF()
```

```
def calculate_Means(self):
    # calculate the means of the Gaussians representing the in-phase and quadrature
    p = np.sqrt(self.K * self.r_hat_2 / (1+self.K)) * np.cos(self.phi)
    q = np.sqrt(self.K * self.r_hat_2 / (1+self.K)) * np.sin(self.phi)
    return p, q

def scattered_Component(self):
    #calculate the power of the scattered signal component
    sigma = np.sqrt(self.r_hat_2 / ( 2 * (1+self.K) ) )
    return sigma

def generate_Gaussians(self, mean, sigma):
    #generates the Gaussian random variable
    gaussians = np.random.default_rng().normal(mean, sigma, self.numSamples)
    return gaussians

def multipath_Fading(self):
    #generate the fading random variables
    p, q = self.calculate_Means() #the mean
    sigma = self.scattered_Component() #the scattered signal component
    multipathFading = self.generate_Gaussians(p, sigma) + (1j*self.generate_Gaussians(q, s
    return multipathFading
```

```
def envelope_PDF(self):
    #calculate the theoretical envelope PDF
    PDF = 2 * (1+self.K) * self.r / self.r_hat_2 \
        * np.exp(- self.K - ((1+self.K)*self.r**2)/self.r_hat_2) \
        * np.i0(2 * self.r * np.sqrt(self.K*(1+self.K)/self.r_hat_2))
    return PDF

def q_func(self,x):
    # Q-function
    return 0.5-0.5*sp.erf(x/np.sqrt(2))

def phase_PDF(self):
    #calculate the theoretical phase PDF
    PDF = (1/(2*np.pi))* np.exp(- self.K) * (1 + (np.sqrt(4*np.pi*self.K) \
            * np.exp(self.K * (np.cos(self.theta-self.phi))**2) *np.cos(self.theta-self.
            * (1 - self.q_func(np.sqrt(2*self.K) * np.cos(self.theta-self.phi))))
    return PDF
```

```
def envelope_Density(self):
    #the envelope PDF of the simulated random variables

    R = np.sqrt((np.real(self.multipathFading))**2 + (np.imag(self.multipathFading))**2)
    kde = kdf(R)
    x = np.linspace(R.min(), R.max(), 100)
    p = kde(x)
    return x, p

def phase_Density(self):
    #the phase PDF of the simulated random variables
    R = np.angle(self.multipathFading)
    kde = kdf(R)
    x = np.linspace(R.min(), R.max(), 100)
    p = kde(x)
    return x, p
```

And test

```
#test rice distribution
K =1 # Kfactor
hat = 1 # amplitude
phi = 1 # phase
numSamples = 16 #length of random samples
s = RiceDistribution(K,hat,phi,numSamples)
# rician random variables
print(s.multipathFading)
```

```
[ 0.73489773+1.26250811j  1.69587359+0.24988565j  0.04602713+0.18332217j
  0.09306215+0.56922115j  0.53118518+0.75571818j  0.11019175+1.24130384j
  0.65126469+1.00204174j  1.27231403+1.46668899j  0.41588962+0.29777712j
  0.33628332+0.47767725j  1.22489098+0.33386301j  0.7917071 +0.69947111j
 -0.10090369+0.69991351j  0.82254623+0.89098798j  0.45033899+0.7466638j
 -0.5212161 +1.71794919j]
```

2.2. conclusion

The larger of the n and x, the closer we get the outcome to p. But the time we spend increases with it.