## Introduction

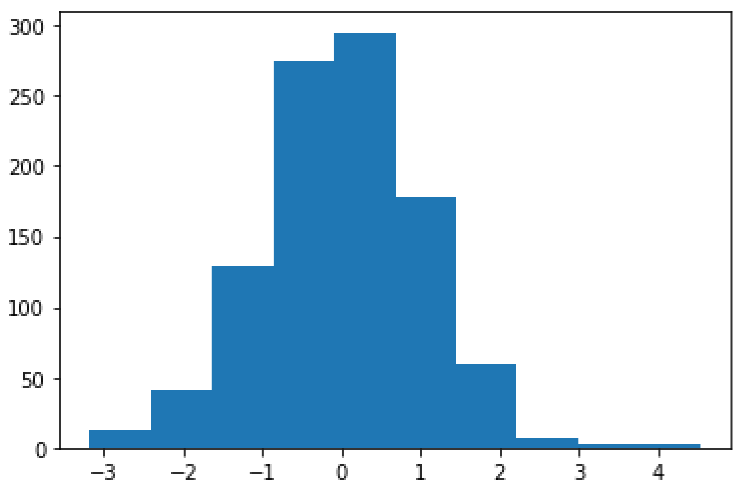
# 1. Sample a univariate Gaussian using scipy.stats.

```
import numpy as np
import random
import matplotlib.pyplot as plt
import scipy.stats as stats
from matplotlib.gridspec import GridSpec
```

```
X = stats.norm.rvs(0,1,size = 1000)
```

```
plt.hist(X)
```

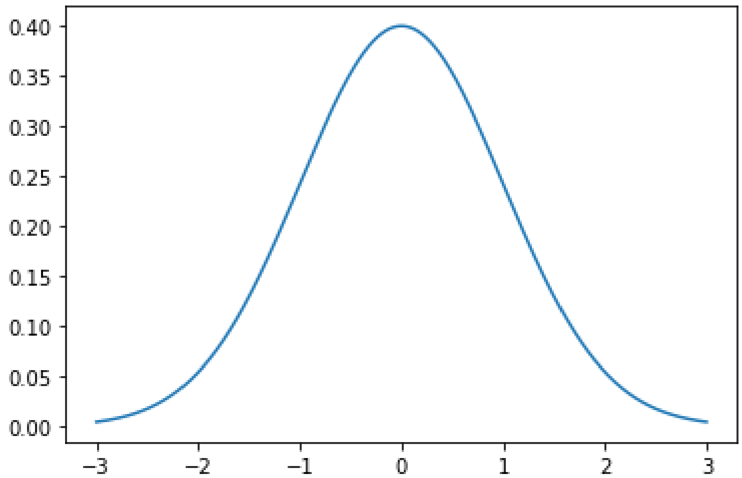
```
(array([ 12.,  41., 129., 274., 295., 178.,  59.,   7.,   2.,   3.]),
 array([-3.17727049, -2.4067069 , -1.63614332, -0.86557974, -0.09501615,
         0.67554743,  1.44611101,  2.2166746 ,  2.98723818,  3.75780176,
         4.52836535]),
 <a list of 10 Patch objects>)
```




# 2. Evaluate the PDF of a univariate Gaussian using scipy.stats.

```
xspace= np.linspace(-3,3,1000)
pdf = stats.norm.pdf(xspace)
plt.plot(xspace,pdf)
```

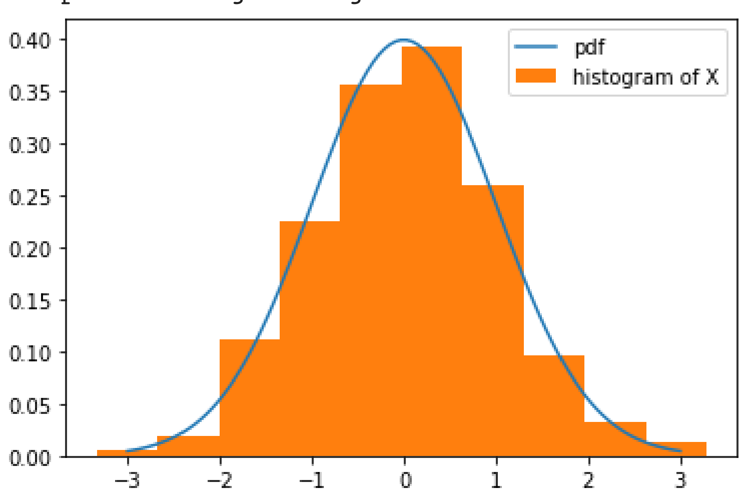
```
[<matplotlib.lines.Line2D at 0x7f4eb15c1190>]
```




## 3. Visualize the PDF of a univariate and a normalized sample histogram of samples from a univariate Gaussian with identical parameters on top of each other using Matplotlib.

```
xspace= np.linspace(-3,3,100)
X = stats.norm.rvs(0,1,size = 1000)

plt.plot(xspace,stats.norm.pdf(xspace), label = "pdf")
plt.hist(X,density=True, label="histogram of X")
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7f4eb1521bd0>
```




# Probability spaces

1. (Dice experiment 1) Consider the probability space model of tossing a fair dice. Let A = {2, 4, 6} and B = {1, 2, 3, 4} be two events. Then, $P(A) = 1/2$, $P(B) = 2/3$ and $P(A \cap B) = 1/3$. Since $P(A \cap B) = P(A)P(B)$, the events A and B are independent. Simulate draws from the outcome space and verify that $\hat{P}(A \cap B) = \hat{P}(A)\hat{P}(B)$, where $\hat{P}(E)$ denotes the proportion of times an event E occurs in the simulation.

```
import numpy as np
import stat
import scipy.stats as rv


#Probability spaces - Ex 1

A = {2, 4, 6}
B = {1, 2, 3, 4}

n = np.int(1e3)
n_A = 0
n_B = 0
n_AB = 0

for i in range(n):

   number = rv.randint.rvs(1,7)

   if number in A:
     n_A = n_A + 1
   if number in B:
     n_B = n_B + 1
   if number in A and number in B:
     n_AB = n_AB + 1

print("Dice Experiment I")
probA=n_A/n
print("Estimated P of event A : ", probA)
probB=n_B/n
print("Estimated P of event B : ", probB)
probAnB=n_AB/n
print("Estimated P of event A & B :",probAnB) # P^(A∩B)
```

```
print("Estimated P of event A & B = P(A)P(B): ",probA*probB) #P^(A)P^(B)

print("So, P^(A∩B) = P^(A)P^(B)  given events A and B are independent.")
```

```
Dice Experiment I
Estimated P of event A :  0.507
Estimated P of event B :  0.68
Estimated P of event A & B : 0.351
Estimated P of event A & B = P(A)P(B):  0.34476
So, P^(A∩B) = P^(A)P^(B)  given events A and B are independent.
```

## 2. (Dice experiment 2) Consider the probability space model of tossing a fair dice. Identify two events A and B that are not independent. Analytically, evaluate P(A), P(B), P(A ∩ B), P(A|B) and P(B|A) and verify these values by means of simulation.

```
#Probability spaces - Ex 2

A= [1,3,5]
B= [2,4,6]

n = np.int(1e3)
n_A = 0
n_B = 0
n_AB = 0

for i in range(n):

   number = rv.randint.rvs(1,7)

   if number in A:
     n_A = n_A + 1
   if number in B:
     n_B = n_B + 1
   if number in A and number in B:
     n_AB = n_AB + 1

print("Dice Experiment II")
probA=n_A/n
print("Estimated P of event A : ", probA)
probB=n_B/n
print("Estimated P of event B : ", probB)
probAnB=n_AB/n
print("Estimated P of event A & B :",probAnB)
```

```
print("Estimated P of event A & B : ",probAnB)
probAgivenB=n_AB/n_B
print("Estimated P of event A given event B : ",probAgivenB)
```

```
Dice Experiment II
Estimated P of event A :  0.509
Estimated P of event B :  0.491
Estimated P of event A & B : 0.0
Estimated P of event A given event B :  0.0
```

3. (Coin experiment) Consider the probability space model of tossing a fair coin twice, i.e. a uniform probability measure on Ω = {HH, HT, T H, T T}, where H indicates heads and T indicates tails. Simulate draws from this probability space and verify that the events  H appears on the  rst toss , H appears on the second toss , and  both tosses have the same outcome  each have probability 1/2.

```
#Probability spaces - Ex 3 , Coin Experiment

n = np.int(1e3)
n_H1 = 0  #no. of heads on toss 1
n_H2 = 0  #no. of heads on toss 2
n_SO = 0  # no. of same output

for i in range(n):

  C = np.full((2,1), np.nan)
  C[0] = rv.bernoulli.rvs(0.5)
  C[1] = rv.bernoulli.rvs(0.5)

  if C[0] == 0:
    n_H1 = n_H1 + 1
  if C[1] == 0:
    n_H2 = n_H2 + 1
  if C[0] == C[1]:
    n_SO = n_SO + 1

print('Coins Experiment')
print('Estimated P of events heads on firest toss is : ', n_H1/n)
print('Estimated P of events heads on second toss is : ', n_H2/n)
print('Estimated P of events heads on second toss is : ', n_SO/n)
```

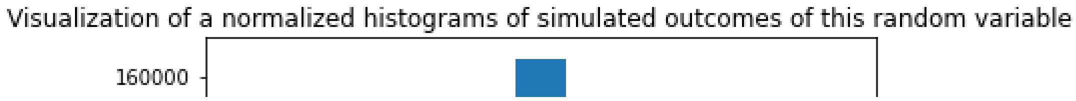
```
print("Hence we can verify that the events 1) H appears on the first toss 2) H appear
```

```
Coins Experiment
Estimated P of events heads on firest toss is :  0.507
Estimated P of events heads on second toss is :  0.473
Estimated P of events heads on second toss is :  0.492
Hence we can verify that the events 1) H appears on the first toss 2) H appears
```

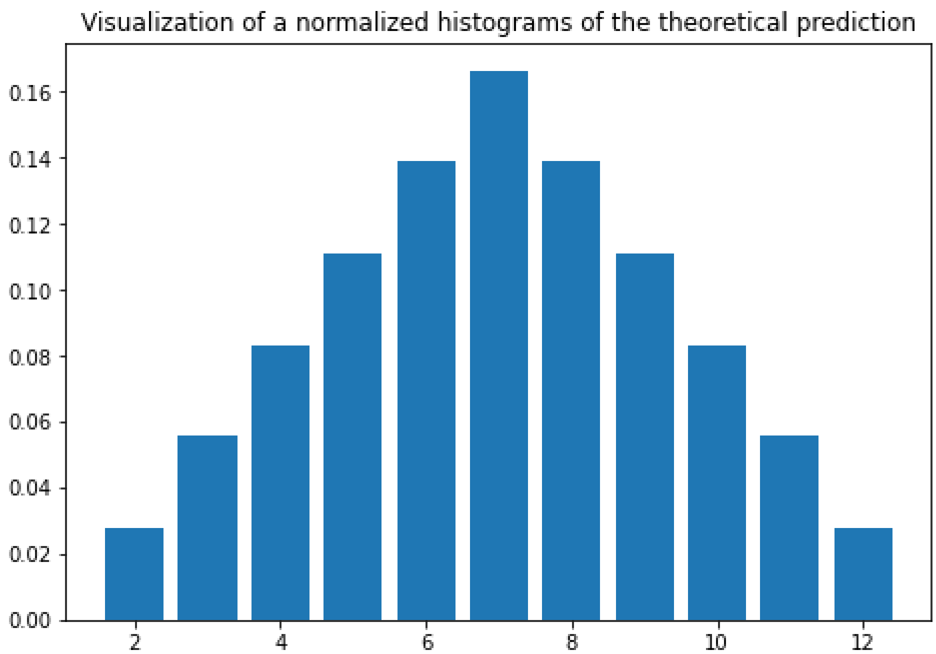
## Random Variables

# 1. Simulate the probability space model of throwing to dice and the random variable corresponding the sum of the pips. Visualize a normalized histograms of simulated outcomes of this random variable and compare it to the theoretical prediction.

```
#Programming Ex - 1

from scipy import *
import numpy as np
import matplotlib.pyplot as plt

N = int(1e6)
dice1 = np.random.randint(low=1, high=7, size=N)
dice2 = np.random.randint(low=1, high=7, size=N)
rv = dice1 + dice2
plt.title("Visualization of a normalized histograms of simulated outcomes of this ran
plt.hist(rv, bins=np.arange(2, 14), align="left", rwidth=0.9)
plt.show()
```




```
fig = plt.figure()
ax = fig.add_axes([0,0,1,1])

x=np.linspace(2,12,11)
p=np.array([1,2,3,4,5,6,5,4,3,2,1])/36

plt.title("Visualization of a normalized histograms of the theoretical prediction")
ax.bar(x,p)
plt.show()
```

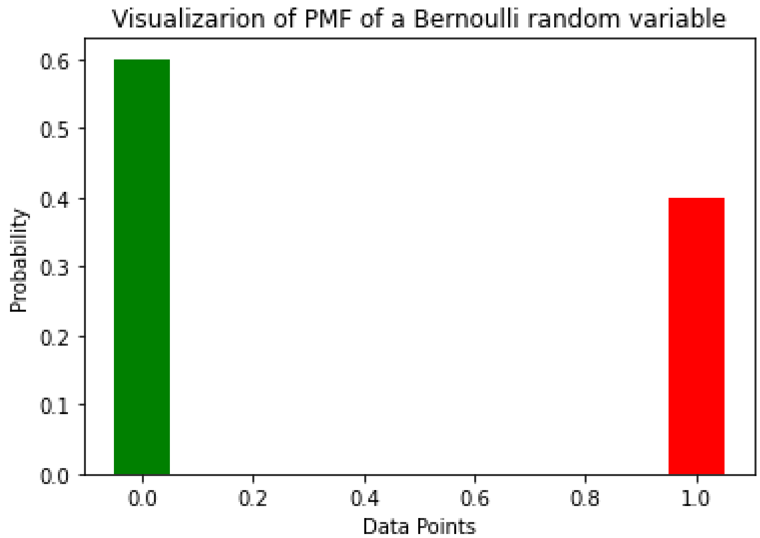



# 2. Visualize the PMF of a Bernoulli random variable and a normalized histogram of many samples of a Bernoulli random variable with identical parameter setting on top of each other.

```
#Programming Ex - 2

from scipy.stats import bernoulli
import matplotlib.pyplot as plt

p=0.4
x = bernoulli.rvs(p, size=100)
pmf = bernoulli.pmf(x,p)
```

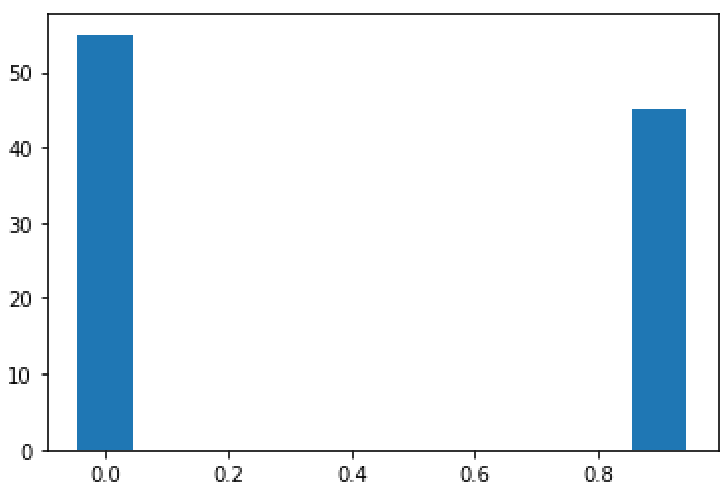
```
plt.bar(x,pmf,width=0.1,color=["r","g"])
plt.title("Visualizarion of PMF of a Bernoulli random variable")
plt.xlabel("Data Points")
plt.ylabel("Probability")
```

```
Text(0, 0.5, 'Probability')
```




```
print("Normalized histogram of many samples of a Bernoulli random variable ")
plt.hist(x, align="left", rwidth=0.9)
plt.show()
```

```
Normalized histogram of many samples of a Bernoulli random variable
```

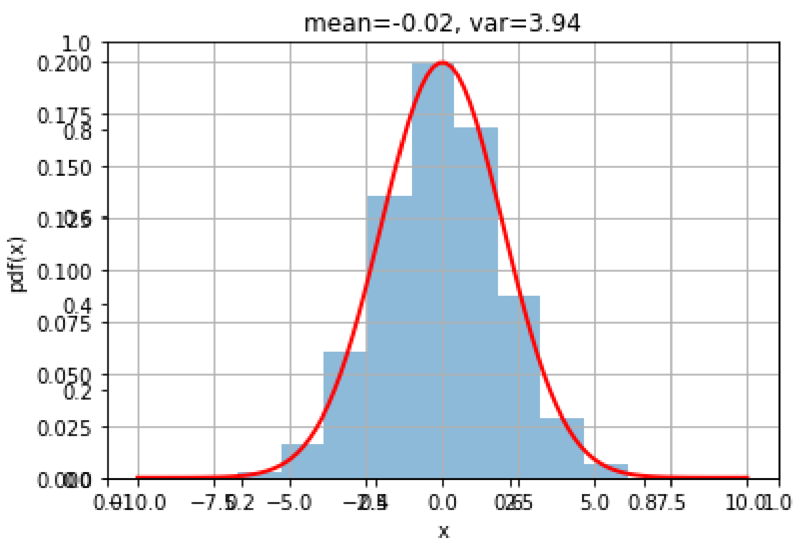



# 3. Visualize the PDF of a Gaussian random variable and a normalized histogram of many samples of a Gaussian

random variable with identical parameter settings on top of

```
#Programming Ex - 3

from scipy.stats import bernoulli
import matplotlib.pyplot as plt
from scipy.stats import norm



fig, ax = plt.subplots()
xs = norm.rvs(scale=2,size=10000) # generate random numbers from N(0,1)
x = np.linspace(-10,10,100)
p = norm.pdf(x,scale=2)         # generate pdf
v = np.var(xs)
m = np.mean(xs)

print("Visualize the PDF of a Gaussian random variable and histogram of many samples
ax = fig.add_subplot(111)
ax.hist(xs, bins=10, alpha=0.5, density=True)
ax.plot(x,p, 'r-', lw=2)
ax.set_xlabel('x')
ax.set_ylabel('pdf(x)')
ax.set_title(f'mean={m:.2f}, var={v:.2f}')
ax.grid(True)
```

```
Visualize the PDF of a Gaussian random variable and histogram of many samples of
```



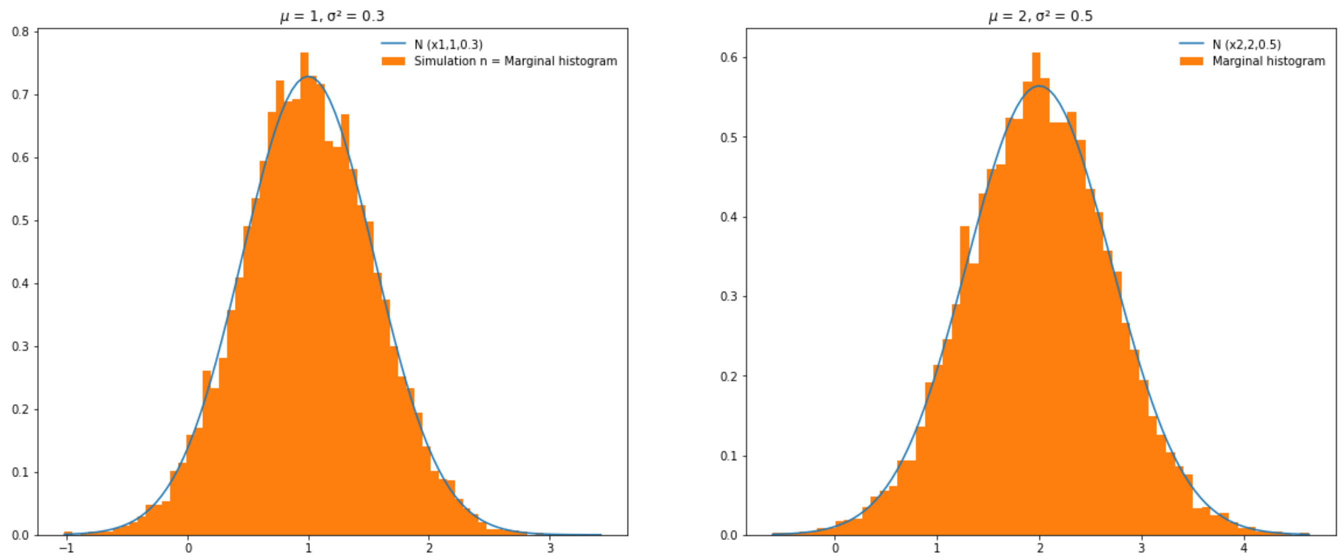

# Joint Distribution

# 1. Write a simulation that demonstrates that the marginal distributions of a bivariate Gaussian distribution with expectation parameter $\mu$=(1,2)$T$ and covariance matrix parameter Σ=(0.30.20.20.5) are given by univariate Gaussian distributions with expectation parameters $\mu$1=1,$\mu$2=2 and variance parameters $\sigma$2=0.3 and $\sigma$2=0.5 , respectively. For the simulation, make use of multivariate Gaussian probability density and random number generators. Visualize and document your results.

```
import numpy as np
import random
import matplotlib.pyplot as plt
import math
import scipy.stats as rv
from scipy.stats import norm
from matplotlib.gridspec import GridSpec


#Initialisation
exp = [1,2]

cov = [[0.3,0.2],
      [0.2,0.5]]

sample_multi_gaussian = np.random.multivariate_normal(exp, cov, size = 10000)


x = sample_multi_gaussian[:,0]
y = sample_multi_gaussian[:,1]

fig, ax = plt.subplots(1, 2, figsize = (20,8))


x1 = np.linspace(min(x), max(x), 100)
y1 = np.linspace(min(y), max(y), 100)

# Visualization
ax[0].plot(x1, norm.pdf(x1, 1, np.sqrt(0.3)),label='N (x1,1,0.3)')
ax[0].hist(x, density=True, bins = 'auto', label = "Simulation n = Marginal histogram
```

```
ax[0].hist(x, density=True, bins = 'auto', label = "Simulation n = Marginal histogram
ax[0].legend(frameon=False)
ax[0].set_title(r'$\mu$ = 1, σ² = 0.3')
ax[1].plot(y1, norm.pdf(y1, 2, np.sqrt(0.5)), label='N (x2,2,0.5)')
ax[1].hist(y, density=True, bins = 'auto', label = "Marginal histogram")
ax[1].legend(loc='best', frameon=False)
ax[1].set_title(r'$\mu$ = 2, σ² = 0.5')
```

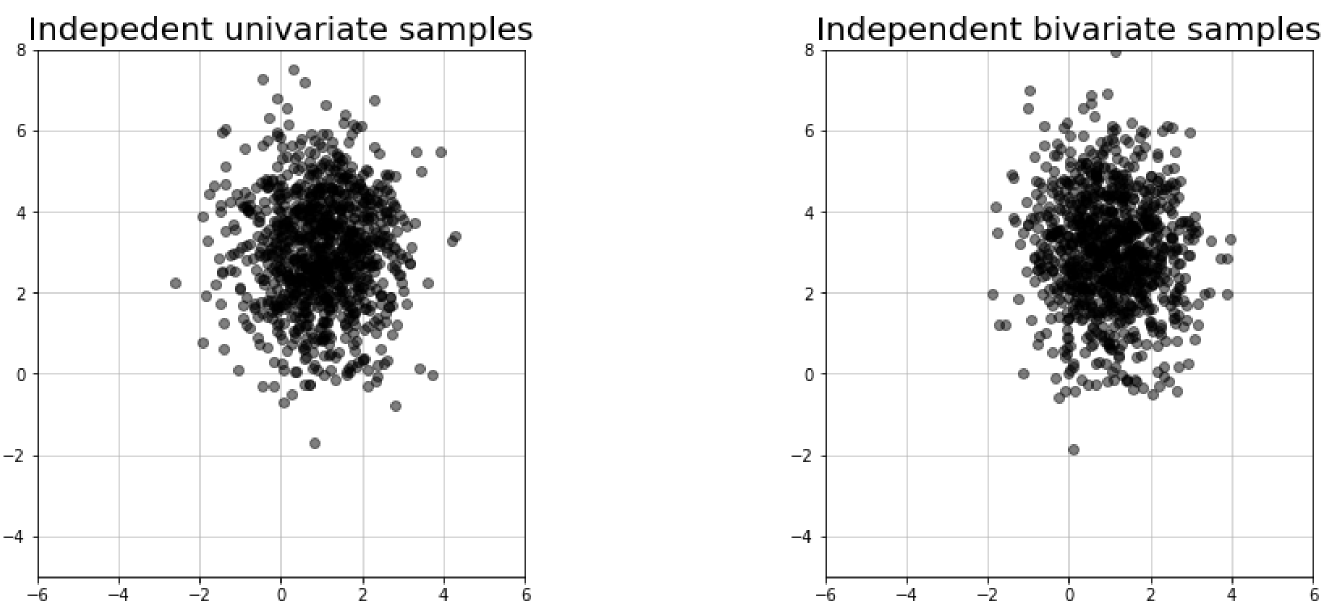
```
Text(0.5, 1.0, '$\\mu$ = 2, σ² = 0.5')
```




## 2. Write a simulation that verifies that obtaining samples from 2 independent univariate Gaussian distributions with parameters μi , σ2 i > 0, i = 1, 2 is equivalent to obtaining samples from a twodimensional Gaussian distribution with the appropriately specified parameters μ ∈ R 2and Σ ∈ R 2×2.

```
n = 1000
mu = [1,3]
sigsqr = [1,2]
X = np.full((n,2,2), np.nan)
subplotlab = ['Indepedent univariate samples', 'Independent bivariate samples']
```

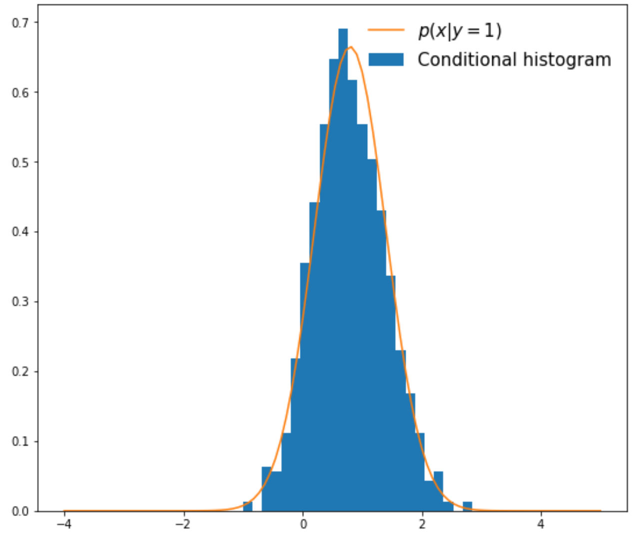
```
#iterative univariate sampling
for i in range(n):
  for j in range(2):
      X[i,j,0] = rv.norm.rvs(mu[j], np.sqrt(sigsqr[j]))

# non iterative bivariate sampling
Sigma = [[sigsqr[0],0], [0,sigsqr[1]]]
X[:,:,1] = rv.multivariate_normal.rvs(mu,Sigma,n)

#visualization
fig = plt.figure(figsize = (16,6))
gs = GridSpec(1,2)
ax = {}

for i in range(2):
  ax[i] = plt.subplot(gs[i])
  ax[i].plot( X[:,0,i],
              X[:,1,i], linestyle = '', marker = 'o', color = 'k', alpha = .5)
  ax[i].set_aspect('equal')
  ax[i].set_xlim(-6,6)
  ax[i].set_ylim(-5,8)
  ax[i].grid(True, linewidth = .5)
  ax[i].set_title(subplotlab[i], fontsize = 20)
```

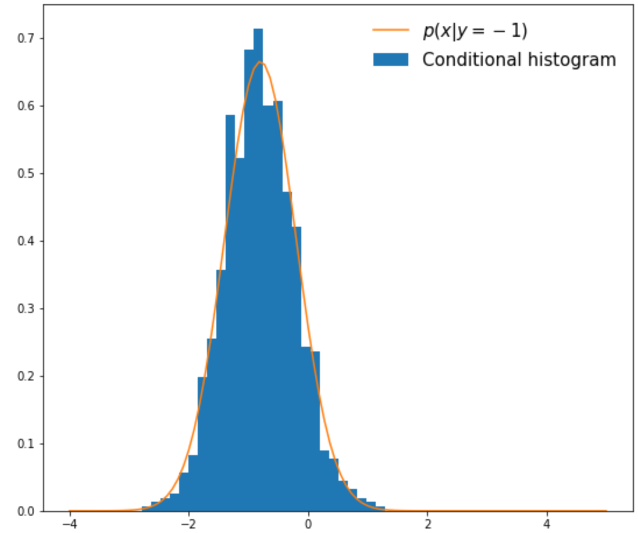



3. Write a simulation that exemplary verifies the analytical results on conditional Gaussian distributions for the case of

a bivariate Gaussian distribution.

```
#Joint distr specifications
mu = [0,0]
Sigma = np.array([[1,.8],[.8,1]])

#A conditional distri specifications
x = np.linspace(-4,5,100)
y = [1,-1]
n = 1000
S = np.full((n,2), np.nan)


#a censored bivariate sampling
for i in range(2):
  j = 0
  while j < n:
    X = rv.multivariate_normal.rvs(mu,Sigma)
    if X[1] > y[i] - 1e-2 and X[1] < y[i] + 1e-2:
      S[j,i] = X[0]
      j = j + 1


fig = plt.figure(figsize = (20,8))
gs = GridSpec(1,2)
ax = {}

# parameter iteratoins
for i in range(2):
  ax[i] = plt.subplot(gs[i])
  ax[i].hist( S[:,i],
              bins = 'auto',
              density = True,

              label = r'Conditional histogram',

              linewidth = .5)
  mu_x_giv_y = mu[0] + Sigma[0,1]*(1/Sigma[1,1])*(y[i] - mu[1])
  Sigma_x_giv_y = Sigma[0,0] - Sigma[0,1]*(1/Sigma[1,1])*Sigma[1,0]
  ax[i].plot(x,
          rv.norm.pdf(x,mu_x_giv_y,np.sqrt(Sigma_x_giv_y)),
          label = r"$p(x|y = {0:1.0f})$".format(y[i]))
  ax[i].legend(frameon = False, fontsize = 15, loc = 'upper right')
```



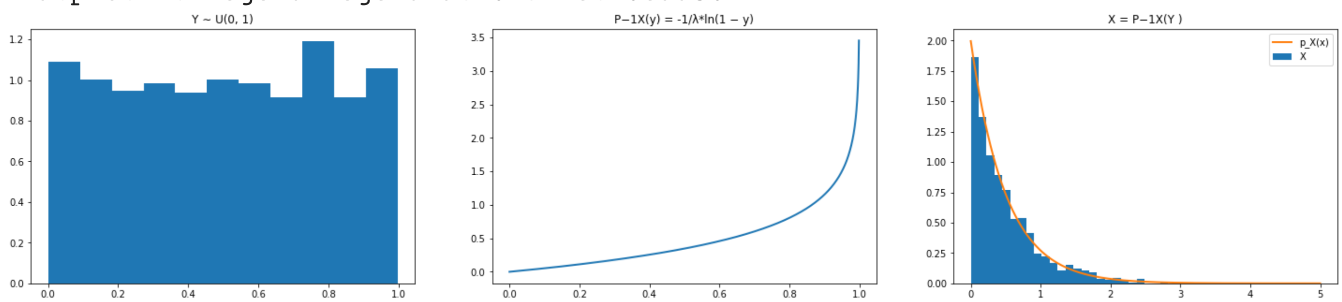

## Transformations

### 1. Write a program that generates pseudo-random numbers from an exponential distribution using a uniform pseudo-random number generator and the probability integral transform theorem

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as stats
from matplotlib.gridspec import GridSpec

x = np.linspace(0,1,1000)
y=  np.linspace(0.001,5,1000)
n=1000
lamb = 2
Y = stats.uniform.rvs(size = n)
transform = -(1/lamb)*np.log(1-Y)
#pdf = stats.expon.pdf(y)
pdf2= lamb*np.exp(-lamb*y)
#Visualization
fig = plt.figure(figsize = (25,5))
gs = GridSpec(1,3)
ax = {}
```

```
ax[0] = plt.subplot(gs[0])
ax[0].hist(Y, density = True, bins = 'auto', linewidth = .5)
ax[0].set_title("Y ~ U(0, 1)")


ax[1] = plt.subplot(gs[1])
ax[1].plot(x,-1/lamb*np.log(1-x),linewidth = 2)
ax[1].set_title("P−1X(y) = -1/λ*ln(1 − y)")

ax[2] = plt.subplot(gs[2])
ax[2].hist(transform, density = True, bins = 'auto', linewidth = .5, label ="X")
ax[2].plot(y,pdf2,linewidth = 2, label="p_X(x)")
ax[2].set_title("X = P−1X(Y )")
ax[2].legend()
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:25: RuntimeWarning:
<matplotlib.legend.Legend at 0x7f4eb10eaa50>
```

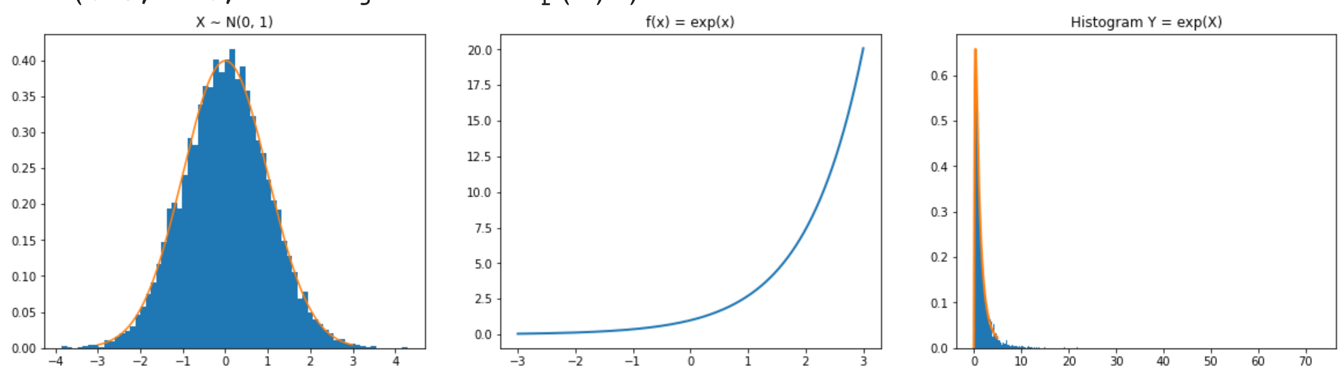



## 2. Let X ~ N(0, 1) and let Y = exp(X). Evaluate the PDF of Y analytically and verify your evaluation using a simulation based on drawing random numbers from N(0, 1).

```
x = np.linspace(-3,3,1000)
y = np.linspace(0.001,5,1000)
n = 10000
zsample = stats.norm.rvs(size = n)
exp_zsample = np.exp(zsample)

#Visualization
fig = plt.figure(figsize = (20,5))
gs = GridSpec(1,3)
ax = {}
```

```
ax[0] = plt.subplot(gs[0])
ax[0].hist(zsample, density = True, bins = 'auto', linewidth = .5)
ax[0].plot(x,stats.norm.pdf(x))
ax[0].set_title("X ~ N(0, 1)")

ax[1] = plt.subplot(gs[1])
ax[1].plot(x,np.exp(x),linewidth = 2)
ax[1].set_title("f(x) = exp(x)")

ax[2] = plt.subplot(gs[2])
ax[2].hist(exp_zsample,density = True, bins = 'auto', linewidth = 0.5)
ax[2].plot(y,(1/np.sqrt(2*np.pi)) * (1/np.abs(y)) * np.exp(-1/2*(np.log(y)**2)),linew
ax[2].set_title("Histogram Y = exp(X)")
```

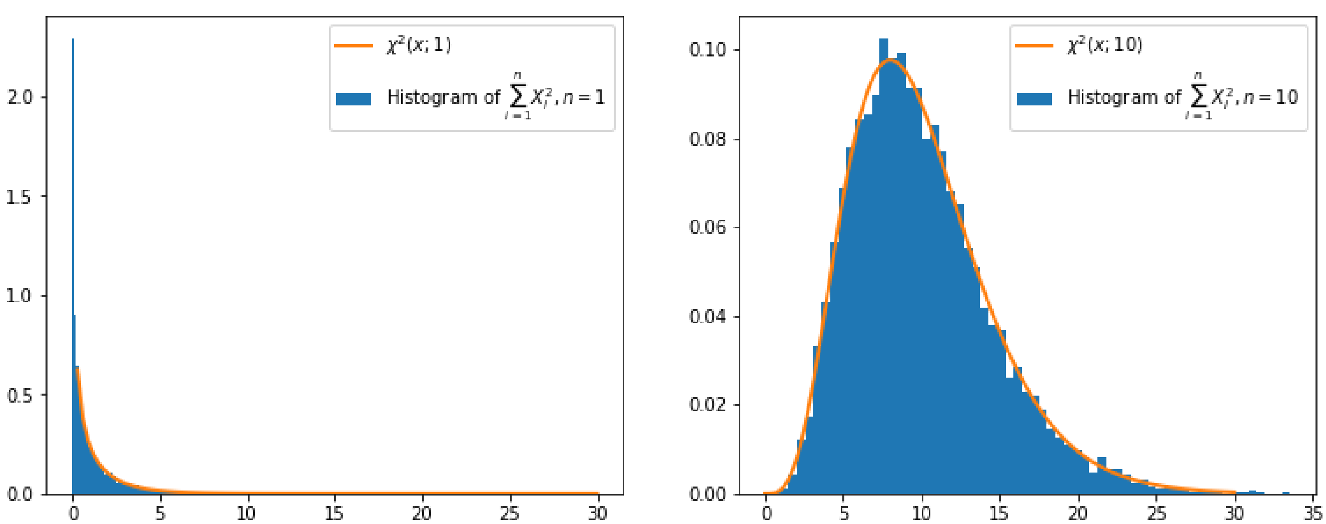
```
Text(0.5, 1.0, 'Histogram Y = exp(X)')
```




## 3. Let X ~ N(0, 1) and let Y = X2. By simulation, validate that Y is distributed according to a chi-squared distribution with one degree of freedom Next, let X1, ..., X10 ~ N(0, 1) and let Y = Sumi=1 to 10 X^2_i. By simulation validate that Y is distributed according to a chi-squared distribution with ten degrees of freedom.

```
x = np.linspace(0,30,100)
Theta = (1,10)
n = 10000

fig = plt.figure(figsize = (20,5))
```

```
fig = plt.figure(figsize = (20,5))
gs = GridSpec(1,3)
ax = {}

#df iterations
for i,theta in enumerate(Theta):
  sample = np.full((n,theta), np.nan)
  #rv iterations
  for j in range(theta):
    #sampling iterations
    for s in range(n):
      sample[s,j] = stats.norm.rvs(0,1)**2

  sample = np.sum(sample,axis=1)

  ax = {}
  ax[i] = plt.subplot(gs[i])
  ax[i].hist(sample, density = True, bins = 'auto',linewidth =2, label = r'Histogram
  ax[i].plot(x,stats.chi2.pdf(x,theta),linewidth =2, label = r'$\chi^2(x;{})$'.format
  ax[i].legend()
```




## Expectation and covariance

1. Sample n = 10 data points of a univariate Gaussian distribution and evaluate the sample mean, sample variance, and sample standard deviation.

```
import scipy.stats as rv
```

```
import numpy as np

x = np.array ([ 0.35,2.3,-2.1,0.8,0.56,0.9,-1.2,1.23,0.2,0.6])
n = len(x)
bar_x = (1/n)*np.sum(x) #sample mean
s = (1/(n-1))*np.sum((x - bar_x)**2)  #sample variance
S = np.sqrt(s) #sample std deviation

print("Total n:", n)
print("Sample mean: {0:2.2f}, np mean: {1:2.2f}".format(bar_x, np.mean(x)))
print("Sample variance: {0:2.2f}, np variance: {1:2.2f}".format(s, np.var(x, ddof=1))
print("Sample std deviation: {0:2.2f}, np std deviation: {1:2.2f}".format(S, np.std(x
```

```
Total n: 10
Sample mean: 0.36, np mean: 0.36
Sample variance: 1.51, np variance: 1.51
Sample std deviation: 1.23, np std deviation: 1.23
```

## 2. Sample n = 10 data points of a bivariate Gaussian distribution and evaluate the sample covariation and sample correlation.

```
x = np.array([[1.2,-0.9],
              [0.7,2],
              [1.2,-0.7],
              [0.2,0.9],
              [-0.3,-0.8],
              [1.2,1.5],
              [2.3,-1.0],
              [0.1,1.1],
              [0.5,0.6],
              [0.8,-0.5]])

n = len(x)
xybar = np.mean(x, axis=0) #sample mean
sx = (1/(n-1)) * np.sum((x[:,0] - xybar[0]) ** 2) # sample variance X^2
sy = (1/(n-1)) * np.sum((x[:,1] - xybar[1]) ** 2) # sample variance Y^2
Sx = np.sqrt(sx) #sample std dev X
Sy = np.sqrt(sy) #samlpe std dev Y
cov = (1/(n-1)) * np.sum((x[:,0] - xybar[0]) * (x[:,1] -xybar[1]))
corr = cov/(Sx * Sy)
npcov = np.cov(x.T, ddof = 1)
npcorr = np.corrcoef(x.T)
```

```
print("Sample covariance:", cov)
print("Sample correlation:", corr)
print("Numpy covariance:", npcov[0,1])
print("Numpy correlation:", npcorr[0,1])
```

```
Sample covariance: -0.2586666666666666
Sample correlation: -0.3127109138566976
Numpy covariance: -0.2586666666666666
Numpy correlation: -0.31271091385669764
```

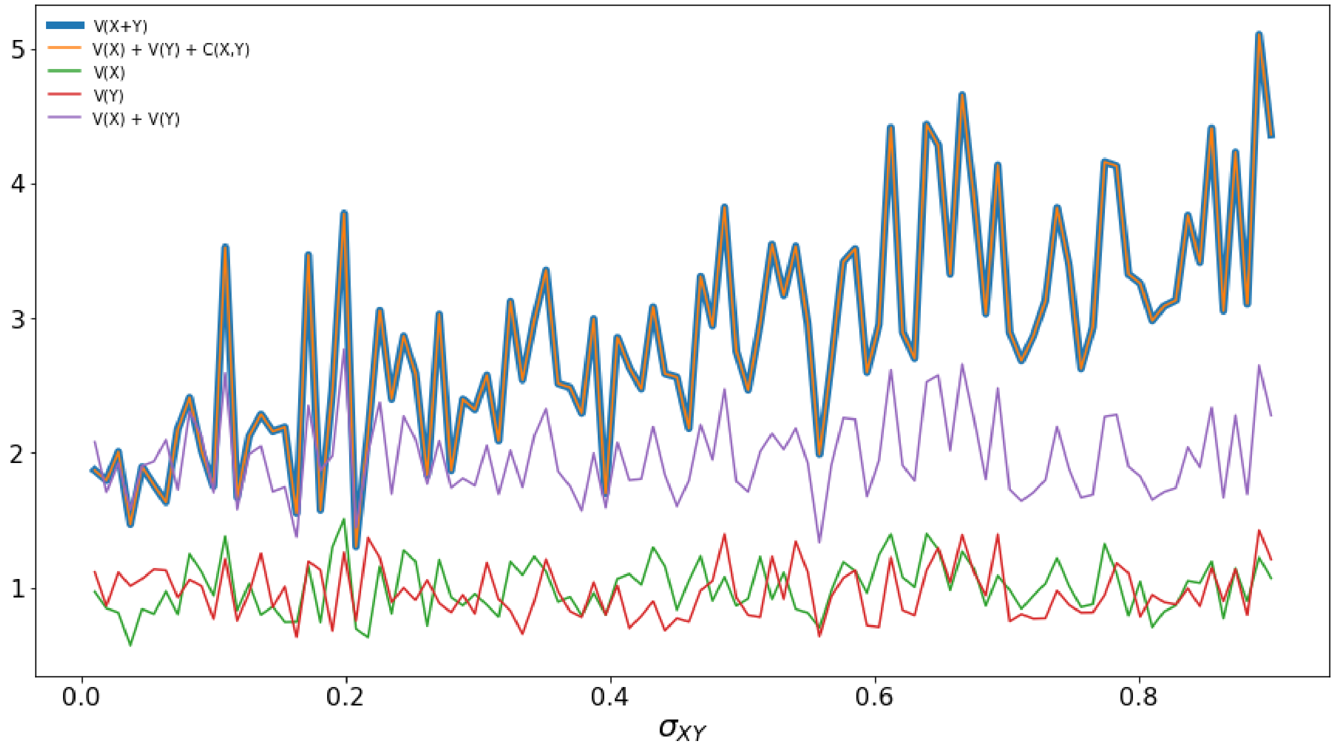
# 3. Validate the theorem on the variances of sums and di erences of random variables using a sampling approach in a bivariate Gaussian scenario.

```
mu = np.array([0,0]) #exp parameter
n = 50 # no of samples
min_sxy = 0.01 # min cov parameter value
max_sxy = .9 # max cov parameter value
res_sxy = 100 # cov parameter space resolution
sxy = np.linspace(min_sxy,max_sxy,res_sxy) # 0.1 to 0.9 - 100 points - cov param spac
V_XY = np.full((res_sxy,1),np.nan) # Sample variance of XY
V_X = np.full((res_sxy,1),np.nan) # Sample variance of X
V_Y = np.full((res_sxy,1),np.nan) # Sample variance of Y
C_XY = np.full((res_sxy,1),np.nan) # Corr of XY


# covariance parameter iterations
for i, sxy_i in np.ndenumerate(sxy):
  Sigma = np.array([[1,sxy_i],[sxy_i,1]])
  XY = rv.multivariate_normal.rvs(mu, Sigma, n)
  XplusY = XY[:,0] + XY[:,1]
  V_XY[i[0]] = np.var(XplusY, ddof =1)
  V_X[i[0]] = np.var(XY[:,0], ddof =1)
  V_Y[i[0]] = np.var(XY[:,1], ddof =1)
  C_XY_matrix = np.cov(XY.T, ddof = 1)
  C_XY[i[0]] =  C_XY_matrix[0,1]
```

```
import matplotlib
import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec
#visualization

fig = plt.figure(figsize = (15,8))
gs = GridSpec(1,1)
ax = {}
```

```
ax[0] = plt.subplot(gs[0])
ax[0].plot(sxy,V_XY,label = "V(X+Y)", linewidth = 5)
ax[0].plot(sxy,V_X + V_Y + 2*C_XY, label = "V(X) + V(Y) + C(X,Y)")
ax[0].plot(sxy,V_X, label = "V(X)")
ax[0].plot(sxy,V_Y, label = "V(Y)")
ax[0].plot(sxy,V_X + V_Y, label = "V(X) + V(Y)")
ax[0].tick_params(labelsize = 16)
ax[0].legend(frameon = False, fontsize = 10, loc = 'upper left')
ax[0].set_xlabel(r'$\sigma_{XY}$', fontsize = 20)
```

```
Text(0.5, 0, '$\\sigma_{XY}$')
```



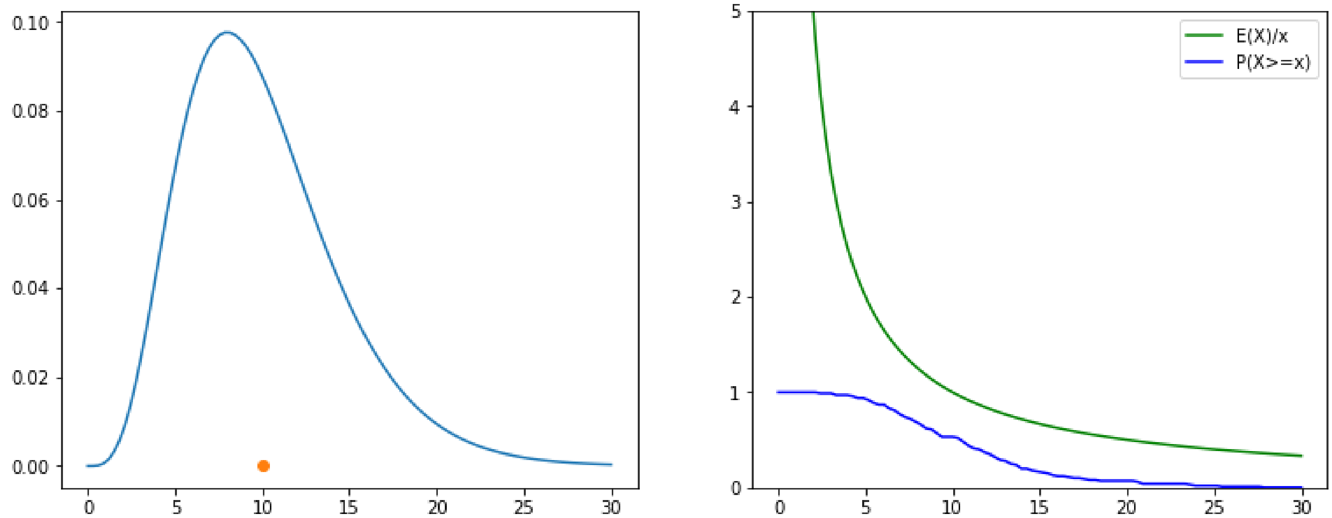

## Inequalities and limits

1. Write simulations that validate the Markov and Chebychev inequalities.

```
#MARKOV
```

```
import numpy as np
import random
import matplotlib.pyplot as plt
import scipy.stats as stats
from matplotlib.gridspec import GridSpec


x_space = np.linspace(0,30,100)
n = 100
Theta = (1,10)

fig = plt.figure(figsize = (20,5))
gs = GridSpec(1,3)
ax = {}

#df iterations
for i,theta in enumerate(Theta):
  sample = np.full((n,theta), np.nan)

  #rv iterations
  for j in range(theta):
    #sampling iterations
    for s in range(n):
      sample[s,j] = stats.norm.rvs(0,1)**2

X = np.sum(sample,axis=1)
exp_X_over_x = np.zeros(100)
prob = []
for i, xx in np.ndenumerate(x_space):
  exp_X_over_x[i] = theta/xx
  P = (X >= xx).sum()/100
  prob.append(P)


ax[0] = plt.subplot(gs[0])
ax[0].plot(x_space,stats.chi2.pdf(x_space,theta))

ax[0].plot(theta, 0, marker = 'o')




ax[1] = plt.subplot(gs[1])
ax[1].plot(x_space, exp_X_over_x , label='E(X)/x', color = 'g')
ax[1].plot(x_space, prob, label = 'P(X>=x)', color = 'b')
ax[1].set_ylim(0,5)
ax[1].legend()
```

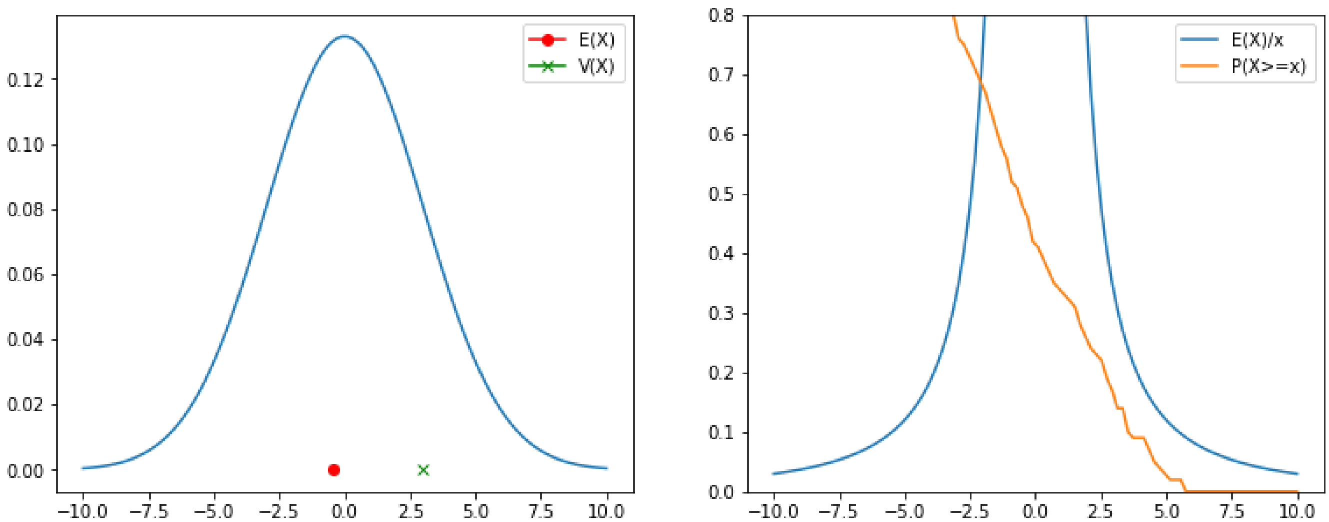
```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:32: RuntimeWarning:
<matplotlib.legend.Legend at 0x7f4eaf0e5090>
```




```
#Chebyshev

import math
x_space = np.linspace(-10,10,100)

n_sim = 100

fig = plt.figure(figsize = (20,5))
gs = GridSpec(1,3)
ax = {}
sample = np.full((n_sim),np.nan)
#df iterations
results = []
result_p = []
EX=0
same_sample_res = []
for i, x in np.ndenumerate(x_space):
  empirical_vals = np.zeros(1000)
  for j in range(n_sim):
      X = stats.norm.rvs(0,3)
      sample[j] = X
  result_p = (abs((sample)- (EX)) >= x).sum()/100
  same_sample_res.append(result_p)

exp = np.mean(sample)
expresult = []
VX=3
expresult2 = []
prb = []
for k, xx in np.ndenumerate(x_space):
   res = exp/xx
   expresult.append(res)
```

```
    res2 = VX/(xx**2)
    expresult2.append(res2)
    res3 = (sample - 0 >= xx).sum()/100
    prb.append(res3)


ax[0] = plt.subplot(gs[0])
ax[0].plot(x_space, stats.norm.pdf(x_space,0,3))
ax[0].plot(np.mean(exp), 0 , marker = 'o', color ='r',label="E(X)")
ax[0].plot(3, 0 , marker = 'x', color ='g',label="V(X)")
ax[0].legend()

ax[1] = plt.subplot(gs[1])
ax[1].plot(x_space, expresult2 , label='E(X)/x')
ax[1].set_ylim(0,0.8)
ax[1].plot(x_space, prb, label = 'P(X>=x)')
ax[1].legend()
```

```
<matplotlib.legend.Legend at 0x7f4eaf017f50>
```

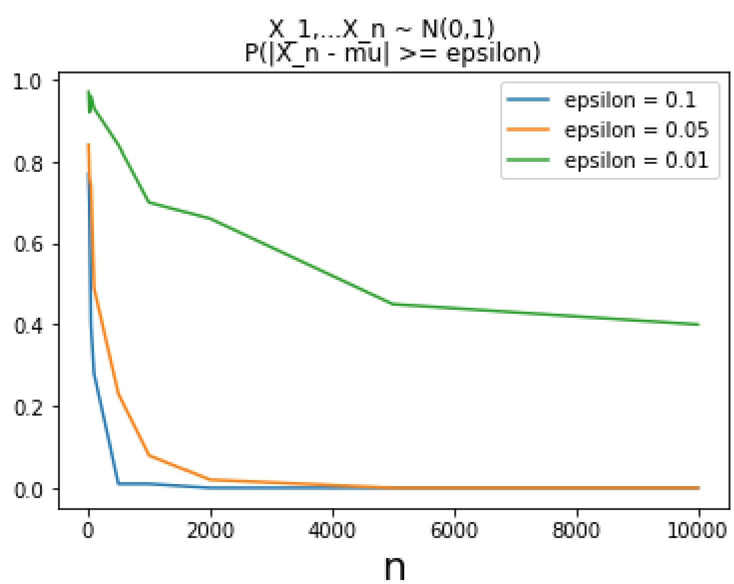



## 2. Write a simulation that validates the Weak Law of Large Numbers.

```
def simulation(epsi = 0.01):
  mu=0
  S =1
  n_sim = 100
  samples = [10,25,50,100,500,1000,2000,5000,10000]
  #simulation iterations
  y_final= []
```

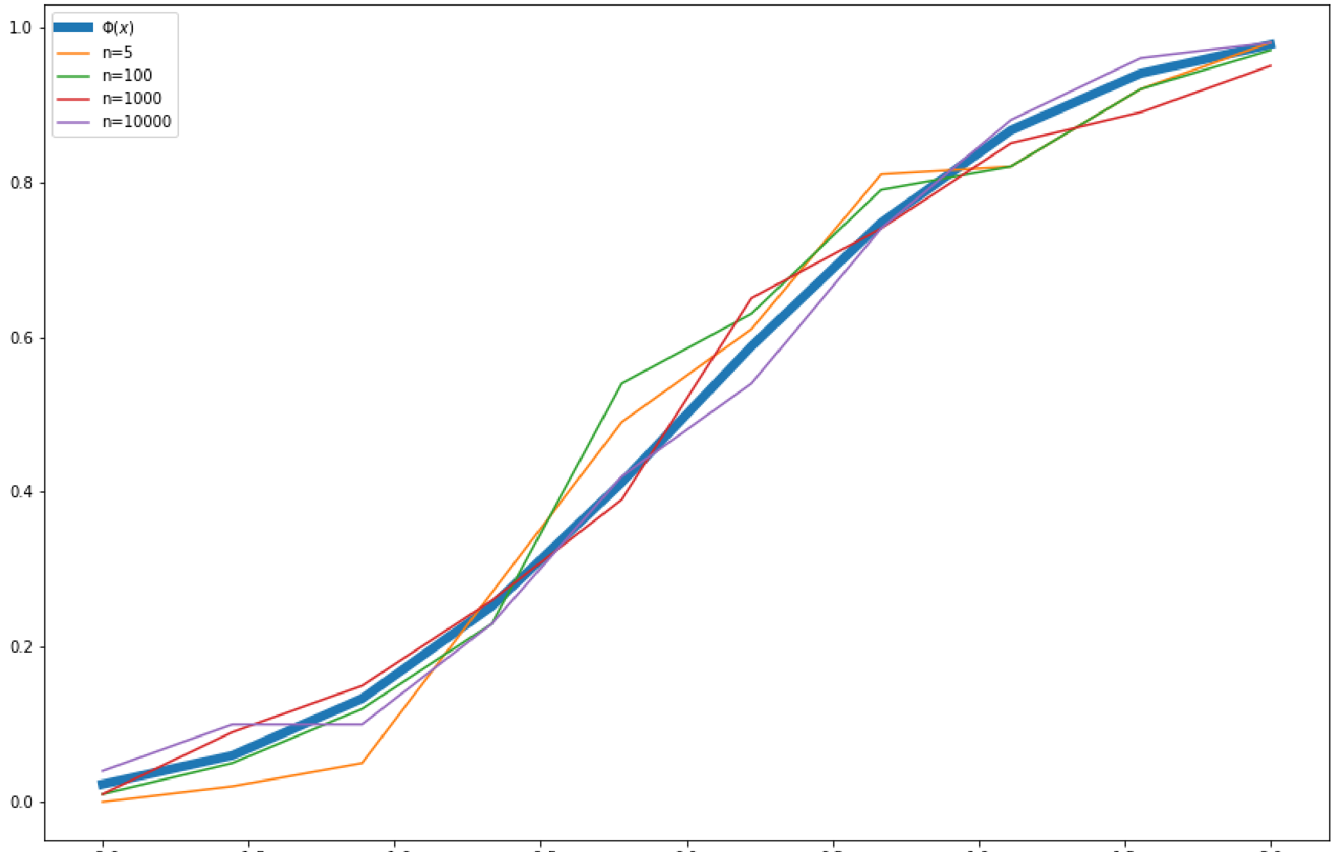
```
    for SIZE in samples:
        size_res = np.full(n_sim,np.nan)
        for i in range(n_sim):
            X = stats.norm.rvs(mu,np.sqrt(S), size = SIZE)
            Xbar = np.mean(X)
            size_res[i] = Xbar
        pr = (abs(size_res - mu) >= epsi).sum()/100
        y_final.append(pr)
    return y_final


epsilon = [0.1,0.05,0.01]
samples = [10,25,50,100,500,1000,2000,5000,10000]
y=np.full((3,9), np.nan)
for i, eps in np.ndenumerate(epsilon):
    y[i] = simulation(eps)


plt.plot(samples,y[0], label = "epsilon = 0.1")
plt.plot(samples,y[1], label = "epsilon = 0.05")
plt.plot(samples,y[2], label = "epsilon = 0.01")
plt.xlabel("n",size=20)
plt.suptitle("X_1,...X_n ~ N(0,1)")
plt.title("P(|X_n - mu| >= epsilon)")
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7f4eaef1fa90>
```




## 3. Write a simulation that validates the Lindenberg-Lévy Central Limit Theorem.

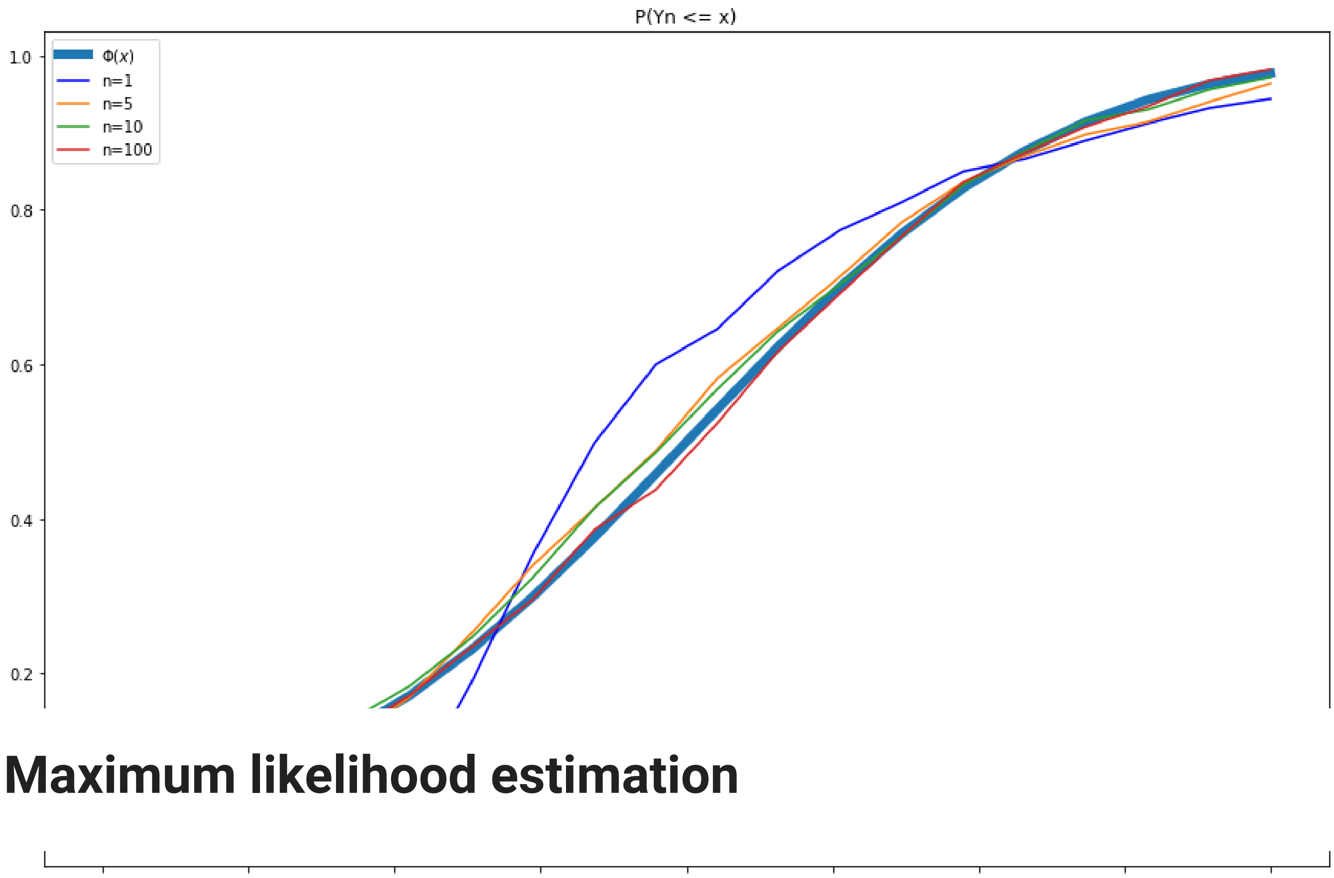
```
import numpy as np
import scipy.stats as stats
import matplotlib.pyplot as plt



x_space = np.linspace(-2,2,10)

def formula(sample):
  mu = stats.gamma.stats(1, moments = 'm')
  sigma = stats.gamma.stats(1,moments = 'v')
  n = len(sample)
  return (np.mean(sample)- mu)/ (sigma / (n** .5))

def GSample(siz):
  return stats.gamma.rvs(1,size = siz)

sample_sizes = [5,100,1000,10000]
results = []
for siz in sample_sizes:
  same_sample_res = []
  for i, x in np.ndenumerate(x_space):
    empirical_vals = np.zeros(100)
    for j in range(100):
      empirical_vals[j] = formula(GSample(siz))

    p= (empirical_vals <=x).sum()/100
    same_sample_res.append(p)
  results.append(same_sample_res)

plt.figure(figsize = (15,10))
plt.plot(x_space, stats.norm.cdf(x_space,0,1), linewidth = 6)
plt.plot(x_space, results[0])
plt.plot(x_space, results[1])
plt.plot(x_space, results[2])
plt.plot(x_space, results[3])
plt.legend(['$\Phi(x)$',
           "n={}".format(sample_sizes[0]),
           "n={}".format(sample_sizes[1]),
           "n={}".format(sample_sizes[2]),
           "n={}".format(sample_sizes[3])], loc = 'upper left')
plt.show()
```




# 4. Write a simulation that validates the Liapunov Central Limit Theorem.

```
import random
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as rv
from matplotlib.pyplot import cm



s_sizes = [1,5,10,100]
x_space = np.linspace(-2,2,20)

repeats = 500
ecdf = np.full((20, len(s_sizes)),np.nan)


#sample size iterations
for idx, ss in np.ndenumerate(s_sizes):
  #simulation repeat iterations
  Y = np.full(repeats, np.nan)
  for s in range(repeats):
    X = np.full(ss, np.nan) #sample realisations
    mu = np.full(ss, np.nan) #expectation
```

```
        sigsqr = np.full(ss, np.nan) #variance

        #random variable iterations
        for k in np.arange(ss):

          mu[k],sigsqr[k] = rv.gamma.stats(k+1,moments ='mv')
          X[k] = rv.gamma.rvs(k+1)

        #summar variable of interest evaluation
        Y[s] = (np.sum(X) - np.sum(mu))/np.sqrt(np.sum(sigsqr))

      #x axis iterations
      for x_idx, x in np.ndenumerate(x_space):
        ecdf[x_idx,idx] = np.mean(Y<=x)



    #plotting

    plt.figure(figsize = (15,10))

    plt.plot(x_space,rv.norm.cdf(x_space,0,1), linewidth = 6)
    plt.plot(x_space, ecdf[:,0], color='b')
    plt.plot(x_space, ecdf[:,1])
    plt.plot(x_space, ecdf[:,2])
    plt.plot(x_space, ecdf[:,3])
    plt.legend(['$\Phi(x)$',
                "n={}".format(s_sizes[0]),
                "n={}".format(s_sizes[1]),
                "n={}".format(s_sizes[2]),
                "n={}".format(s_sizes[3])], loc = 'upper left')
    plt.title("P(Yn <= x)")
    plt.show()
    #plt.set_xlabel(r'$X$', fontsize = 17)
    #plt.set_ylim(0,1)
```




## (8) Maximum likelihood estimation

1) Let X1, ..., Xn ~ Bern(μ) be n = 20 i.i.d. Bernoulli random variables. Using an optimization routine of your choice, formulate and implement the numerical maximum likelihood estimation of μ for true, but unknown values of μ = 0.7 and μ = 1 based on X1, ..., Xn.

```
import scipy.stats as rv
import matplotlib
import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec
import numpy as np
import matplotlib.pyplot as plt
import math
plt.style.use('ggplot')
from scipy.optimize import minimize
```

```
#Initialisation
n = 20
mus = [0.7,1.0]
```

```
rvs = np.zeros((len(mus),n))
numerical_mle = np.zeros((len(mus))) #store num MLE

nspace = np.linspace(0,1,1000)
#ln = np.zeros((len(mus),len(nspace))) #loglikelihood for both mus

db_1h = np.zeros((len(mus), len(nspace)))

fig = plt.figure(figsize = (15,17))
gs = GridSpec(1,2)
ax = {}
```

```
<Figure size 1080x1224 with 0 Axes>
```

```
def bernln(mu,x):
  mu = mu
  n = len(x)
  bernlnmu = np.log(mu)*np.sum(x) + np.log(1-mu)*(n-np.sum(x))
  return  bernlnmu

def mle(X):
  res = minimize(negbernln,
                 args = X,
                 x0 = 0.5,
                 bounds = [1e-3,1-1e-3],
                 method = 'Nelder-Mead')
  mu_hat = res.x
  return mu_hat

def negbernln(mu,x):
  mu = mu
  n = len(x)
  bernlnmu = - np.log(mu)*np.sum(x) - np.log(1-mu)*(n-np.sum(x))
  return  bernlnmu


#Store realisations of n i.i.d Bern RV
for i,mu in enumerate(mus):
  rvs[i] = rv.bernoulli.rvs(mu, size=n)

#iterations
for i, mu in np.ndenumerate(mus):
    for j, l in np.ndenumerate(nspace):
      db_1h[i,j] = bernln(l,rvs[i])
    numerical_mle[i] = mle(rvs[i])

#plotting
ax[0] = plt.subplot(gs[0])
ax[0].plot(nspace,
           db_1h[0],
           label =r'$\ell(\mu)$',
           color = [0,0,0])
```

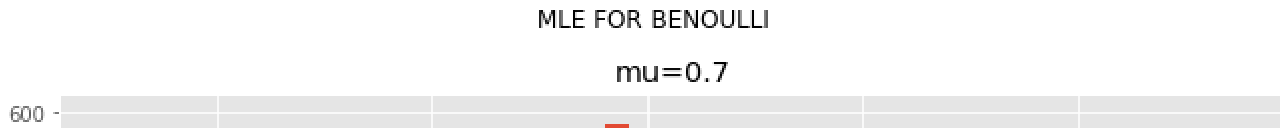
```
           color = [0,0,0])
ax[0].plot(numerical_mle[0],
          min(db_1h[0,1:999]),
          marker = 'o',
          color = 'r',
          mfc = 'r',
          label =r'$\hat{\mu^{ML}}$')
ax[0].plot(numerical_mle[0],
          max(db_1h[0]),
          marker = '^',
          color = 'b',
          mfc = 'b',
          label =r'$\ell(\hat{\mu^{ML}})$')
ax[0].legend()

ax[1] = plt.subplot(gs[1])
ax[1].plot(nspace,
           db_1h[1],
           label =r'$\ell(\mu)$',
           color = [0,0,0])
ax[1].plot(numerical_mle[1],
           min(db_1h[1,1:999]),
          marker = 'o',
          color = 'r',
          mfc = 'r',
         label =r'$\hat{\mu^{ML}}$')
ax[1].plot(numerical_mle[1],
           max(db_1h[1]),
          marker = '^',
          color = 'b',
          mfc = 'b',
          label =r'$\ell(\hat{\mu^{ML}})$')
ax[1].legend()
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:4: RuntimeWarning:
  after removing the cwd from sys.path.
/usr/local/lib/python3.7/dist-packages/scipy/optimize/_minimize.py:522: RuntimeW
  RuntimeWarning)
```

# 2) Let X1, ..., Xn ~ Bern(μ). For a large number n, sample the X1, ..., Xn and evaluate the maximum likelihood estimator μˆML. Repeat this m times and create a histogram of the realized μˆML 1 , ..., μˆML m .


```
mu = 0.7
n= 100000 #samples
m = 10000 #no. of tries

ML = np.zeros(m)

#Iteration
for i in range(m):
  s = rv.bernoulli.rvs(mu,size=n)
  smean = 1/n * np.sum(s)
  ML[i] = smean

fig, ax1 = plt.subplots(1, figsize=(11,6))
fig.suptitle("MLE FOR BENOULLI")

ax1.hist(ML,bins=50)
ax1.set_title('mu=0.7')
```

```
Text(0.5, 1.0, 'mu=0.7')
```



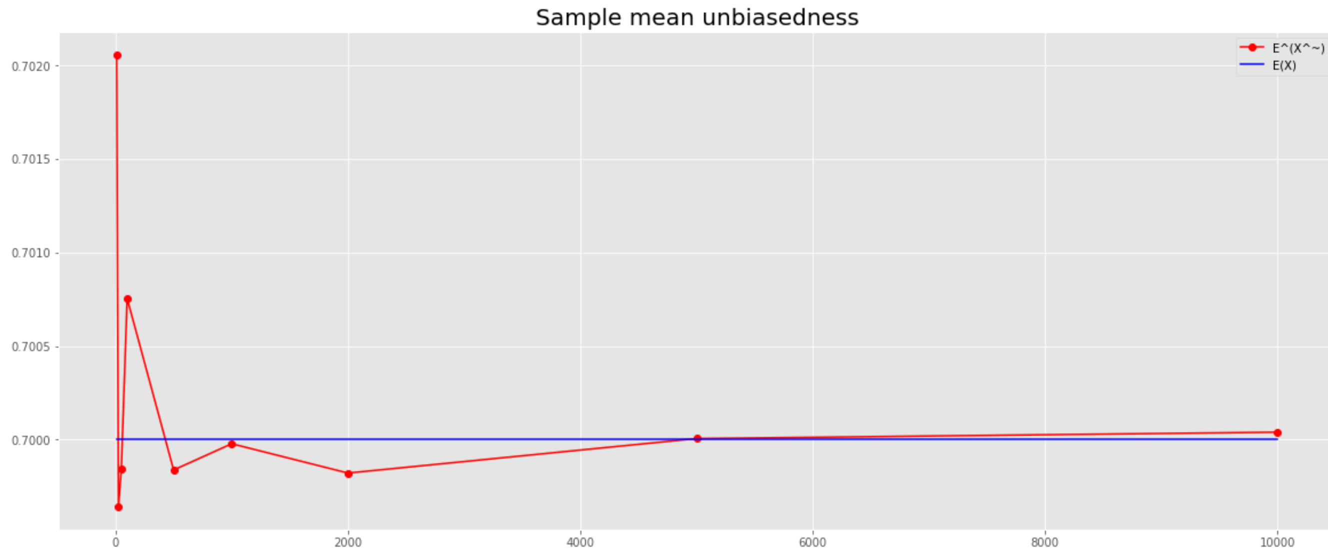

# (9) Finite estimator properties

## 1. For X1, ..., Xn ~ Bern(μ) implement a simulation which validates the unbiasedness of the sample mean, the unbiasedness of the sample variance, the biasedness of the sample standard deviation, and the biasedness of the maximum likelihood variance parameter estimator.

```
import scipy.stats as rv
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec
import scipy.stats as stats
import numpy as np
import matplotlib.pyplot as plt
import math
plt.style.use('ggplot')
from progressbar import ProgressBar


mu=0.7
S =1
n=100000 #sample size
n_sim = 10000
s = range(n_sim)

samples = [10,25,50,100,500,1000,2000,5000,10000]

#simulation iterations
Sn_s = []
for SIZE in samples:
    size_res = []
    for i in range(n_sim):
        X = rv.bernoulli.rvs(mu, size = SIZE)
        size_res.append(np.mean(X))
    Sn_s.append(np.sum(np.array(size_res))/n_sim)
```

```
fig = plt.figure(figsize = (20,8))
gs = GridSpec(1,2)
ax = {}

ax[0] = plt.subplot()

ax[0].plot (samples,
            Sn_s,
            color = 'r',
            label = "E^(X^~)",
            marker ='o')
ax[0].plot (samples,
            mu*np.ones(9),
            color = 'b', label = "E(X)")
ax[0].set_title("Sample mean unbiasedness", fontsize=20)
ax[0].legend()
```

```
<matplotlib.legend.Legend at 0x7f4eaef04850>
```



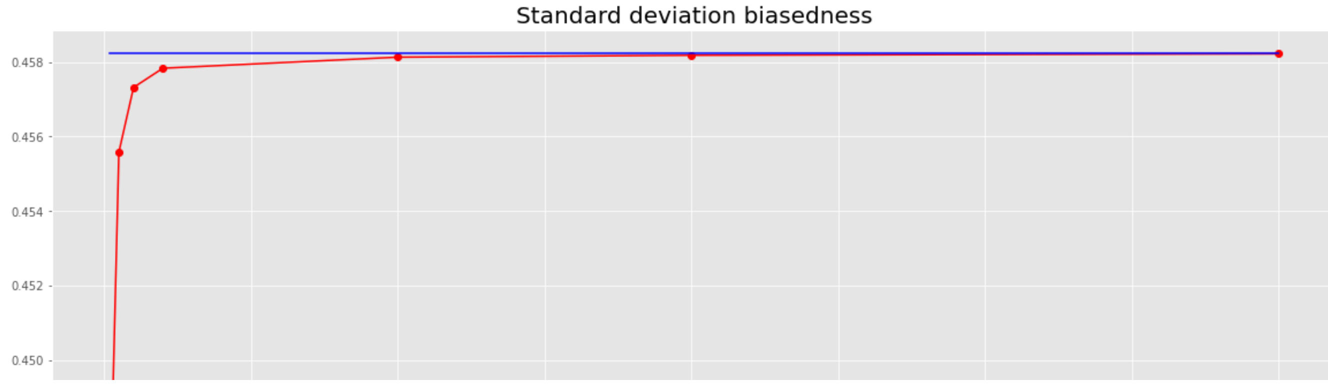

```
mu=0.7
S =1
n=100000 #sample size
n_sim = 10000
s = range(n_sim)

samples = [1,5,10,25,50,100,500,1000,2000,5000,10000,20000]

#simulation iterations
```

```
Sn_s = []
for SIZE in samples:
    size_res = []
    for i in range(n_sim):
        X = rv.bernoulli.rvs(mu, size = SIZE)
        size_res.append(np.var(X, ddof = 1))
    Sn_s.append(np.sum(np.array(size_res))/n_sim)

V_X = 0.21

fig = plt.figure(figsize = (20,8))
gs = GridSpec(1,2)
ax = {}

ax[0] = plt.subplot()

ax[0].plot (samples,
            Sn_s,
            color = 'r',
            label = "E^(S^2)",
            marker ='o')
ax[0].plot (samples,
            V_X*np.ones(12),
            color = 'b', label = "V(X)")
ax[0].set_title("Sample variance unbiasedness", fontsize=20)
ax[0].legend()
```

```
/usr/local/lib/python3.7/dist-packages/numpy/core/fromnumeric.py:3622: RuntimeWa
  **kwargs)
/usr/local/lib/python3.7/dist-packages/numpy/core/_methods.py:226: RuntimeWarnin
  ret = ret.dtype.type(ret / rcount)
```

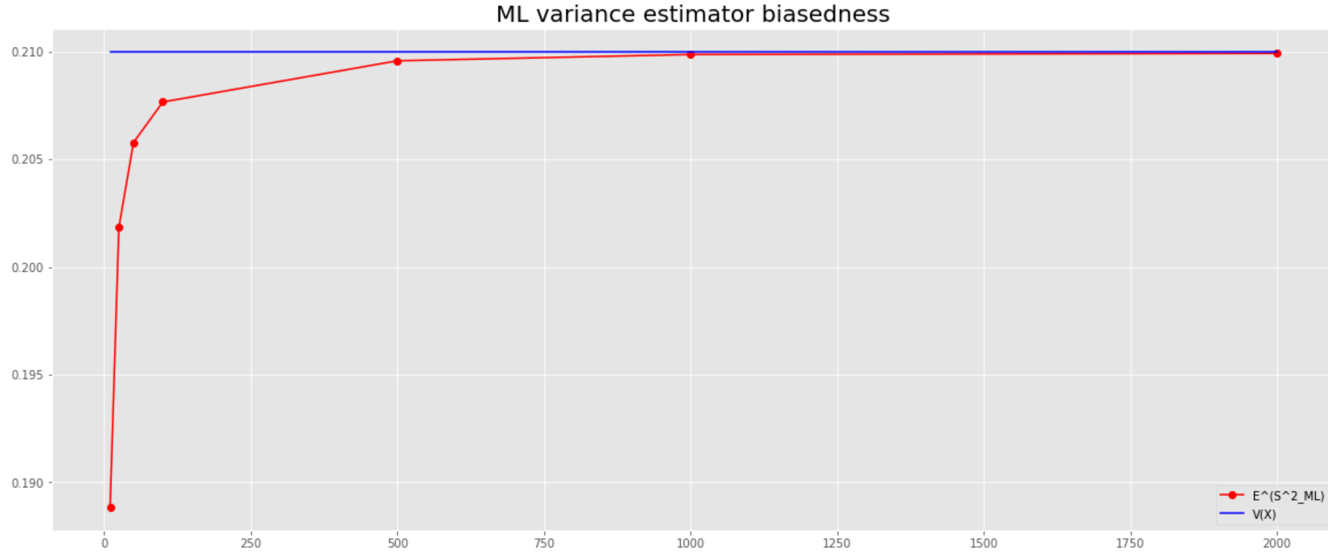
```
mu=0.7
S =1
n=100000 #sample size
n_sim = 10000
s = range(n_sim)

samples = [10,25,50,100,500,1000,2000]
#simulation iterations
Sn_s = []
for SIZE in samples:
    size_res = []
    for i in range(n_sim):
        X = rv.bernoulli.rvs(mu, size = SIZE)
        size_res.append(np.std(X, ddof = 1))
    Sn_s.append(np.sum(np.array(size_res))/n_sim)

S_X = 0.458247

fig = plt.figure(figsize = (20,8))
gs = GridSpec(1,2)
ax = {}

ax[0] = plt.subplot()

ax[0].plot (samples,
            Sn_s,
            color = 'r',
            label = "E^(S)",marker ='o')
ax[0].plot (samples,
            S_X*np.ones(7),
            color = 'b', label = "S(X)")
ax[0].set_title("Standard deviation biasedness", fontsize=20)
ax[0].legend()
```

```
<matplotlib.legend.Legend at 0x7f4eaee9be90>
```




```
mu=0.7
S =1
n=100000 #sample size
n_sim = 10000
s = range(n_sim)

samples = [10,25,50,100,500,1000,2000]

#simulation iterations
Sn_s = []
for SIZE in samples:
    size_res = []
    for i in range(n_sim):
        X = rv.bernoulli.rvs(mu, size = SIZE)
        size_res.append(np.var(X, ddof = 0))
    Sn_s.append(np.sum(np.array(size_res))/n_sim)

V_X = 0.21

fig = plt.figure(figsize = (20,8))
gs = GridSpec(1,2)
ax = {}

ax[0] = plt.subplot()

ax[0].plot (samples,
            Sn_s,
            color = 'r',
            marker ='o',
            label = "E^(S^2_ML)")
ax[0].plot (samples,
            V_X*np.ones(7),
            color = 'b', label = "V(X)")
ax[0].set_title("ML variance estimator biasedness", fontsize=20)
ax[0].legend()
```

```
<matplotlib.legend.Legend at 0x7f4eaececc10>
```

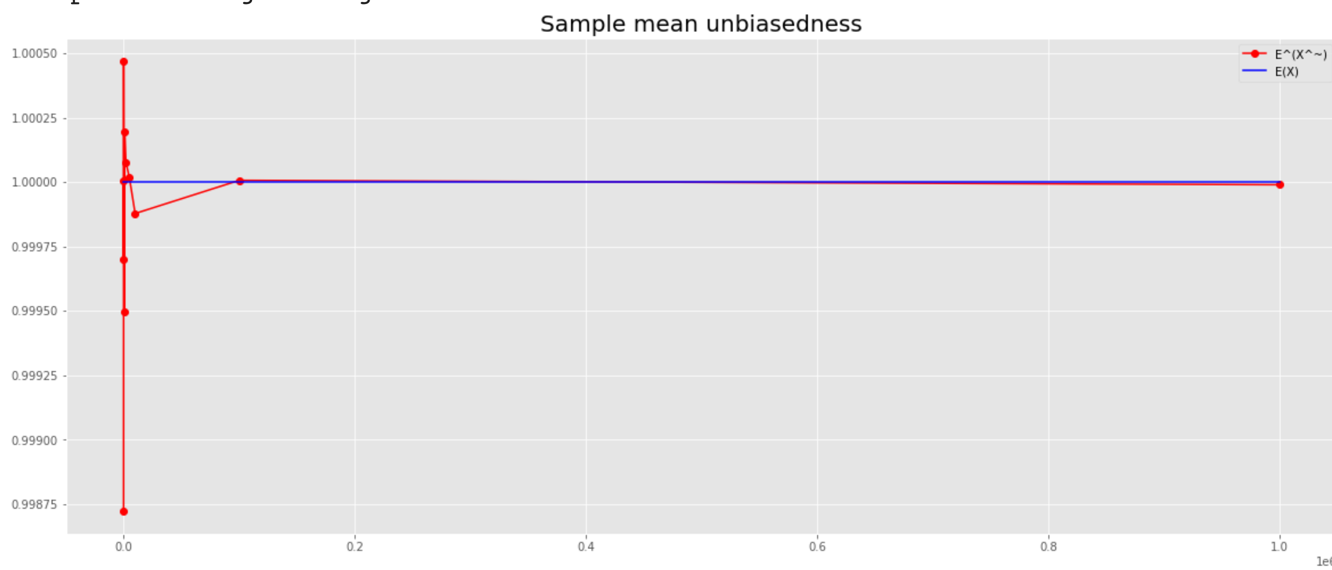



2) For X1, ..., Xn ~ N(μ, σ2) implement a simulation which validates the unbiasedness of the sample mean, the unbiasedness of the sample variance, the biasedness of the sample standard deviation, and the biasedness of the maximum likelihood variance parameter estimator.

```
mu=1
S =0.5
n=100000 #sample size
n_sim = 10000
s = range(n_sim)

samples = [10,25,50,100,500,1000,2000,5000,10000,100000,1000000]

#simulation iterations
Sn_s = []
for SIZE in samples:
    size_res = []
    for i in range(n_sim):
        X = rv.norm.rvs(mu,np.sqrt(S), size = SIZE)
        size_res.append(np.mean(X))
    Sn_s.append(np.sum(np.array(size_res))/n_sim)
```

```
fig = plt.figure(figsize = (20,8))
gs = GridSpec(1,2)
ax = {}

ax[0] = plt.subplot()

ax[0].plot (samples,
            Sn_s,
            color = 'r',
            label = "E^(X^~)",
            marker ='o')
ax[0].plot (samples,
            mu*np.ones(11),
            color = 'b', label = "E(X)")
ax[0].set_title("Sample mean unbiasedness", fontsize=20)
ax[0].legend()
```

```
<matplotlib.legend.Legend at 0x7f4eaec7ca10>
```



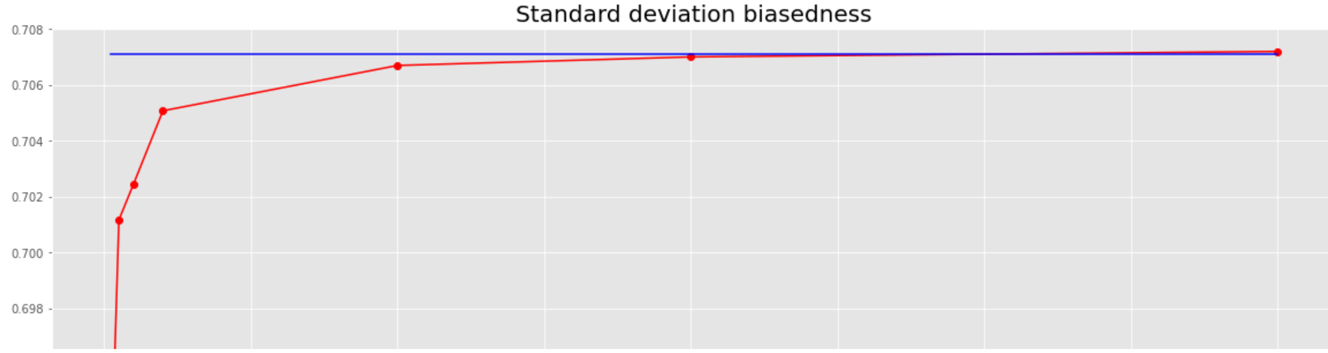

```
mu=1
S =0.5
n=100000 #sample size
n_sim = 10000
s = range(n_sim)

samples = [1,5,10,25,50,100,500,1000,2000,5000,10000,20000]

#simulation iterations
```

```
Sn_s = []
for SIZE in samples:
    size_res = []
    for i in range(n_sim):
        X = rv.norm.rvs(mu,np.sqrt(S),size = SIZE)
        size_res.append(np.var(X, ddof = 1))
    Sn_s.append(np.sum(np.array(size_res))/n_sim)


fig = plt.figure(figsize = (20,8))
gs = GridSpec(1,2)
ax = {}

ax[0] = plt.subplot()

ax[0].plot (samples,
            Sn_s,
            color = 'r',
            label = "E^(S^2)",
            marker ='o')
ax[0].plot (samples,
            S*np.ones(12),
            color = 'b', label = "V(X)")
ax[0].set_title("Sample variance unbiasedness", fontsize=20)
ax[0].legend()
```

```
/usr/local/lib/python3.7/dist-packages/numpy/core/fromnumeric.py:3622: RuntimeWa
  **kwargs)
/usr/local/lib/python3.7/dist-packages/numpy/core/_methods.py:226: RuntimeWarnin
```

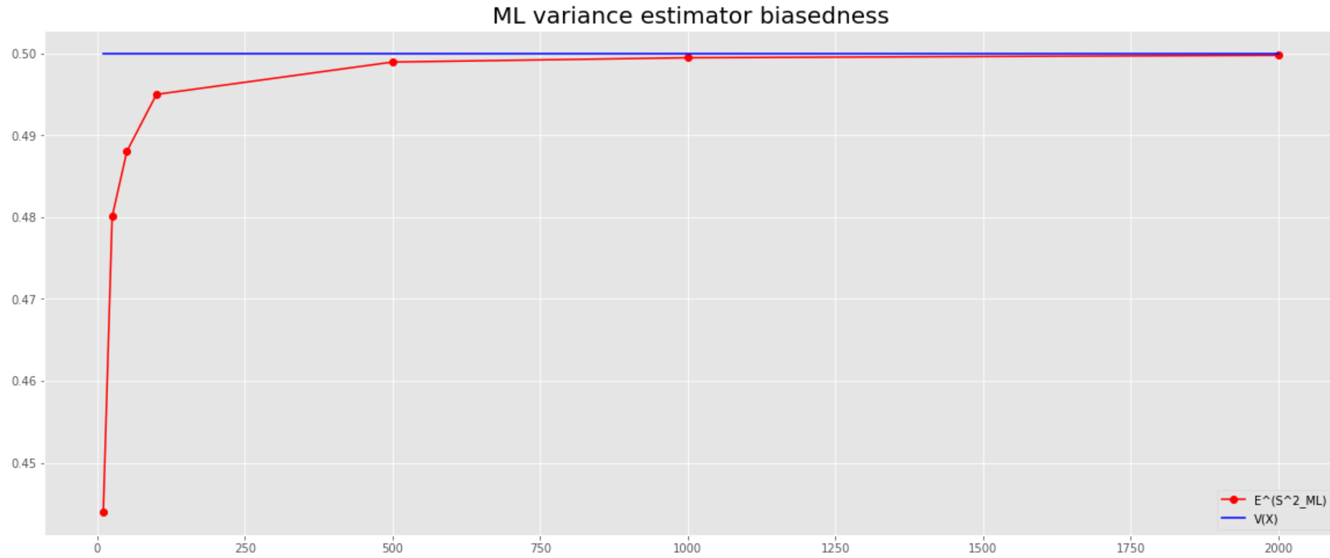
```
mu=1
S =0.5
n=100000 #sample size
n_sim = 10000
s = range(n_sim)

samples = [10,25,50,100,500,1000,2000]
#simulation iterations
Sn_s = []
for SIZE in samples:
    size_res = []
    for i in range(n_sim):
        X = rv.norm.rvs(mu,np.sqrt(S), size = SIZE)
        size_res.append(np.std(X, ddof = 1))
    Sn_s.append(np.sum(np.array(size_res))/n_sim)

S_X = 0.7071

fig = plt.figure(figsize = (20,8))
gs = GridSpec(1,2)
ax = {}

ax[0] = plt.subplot()

ax[0].plot (samples,
            Sn_s,
            color = 'r',
            label = "E^(S)",marker ='o')
ax[0].plot (samples,
            S_X*np.ones(7),
            color = 'b', label = "S(X)")
ax[0].set_title("Standard deviation biasedness", fontsize=20)
ax[0].legend()
```

```
<matplotlib.legend.Legend at 0x7f4eaeb89d90>
```




```
mu=1
S =0.5
n=100000 #sample size
n_sim = 10000
s = range(n_sim)

samples = [10,25,50,100,500,1000,2000]

#simulation iterations
Sn_s = []
for SIZE in samples:
    size_res = []
    for i in range(n_sim):
        X = rv.norm.rvs(mu,np.sqrt(S), size = SIZE)
        size_res.append(np.var(X, ddof = 0))
    Sn_s.append(np.sum(np.array(size_res))/n_sim)


fig = plt.figure(figsize = (20,8))
gs = GridSpec(1,2)
ax = {}

ax[0] = plt.subplot()

ax[0].plot (samples,
            Sn_s,
            color = 'r',
            marker ='o',
            label = "E^(S^2_ML)")
ax[0].plot (samples,
            S*np.ones(7),
            color = 'b', label = "V(X)")
ax[0].set_title("ML variance estimator biasedness", fontsize=20)
ax[0].legend()
```

```
<matplotlib.legend.Legend at 0x7f4eaeaa0810>
```




## (10) Asymptotic estimator properties

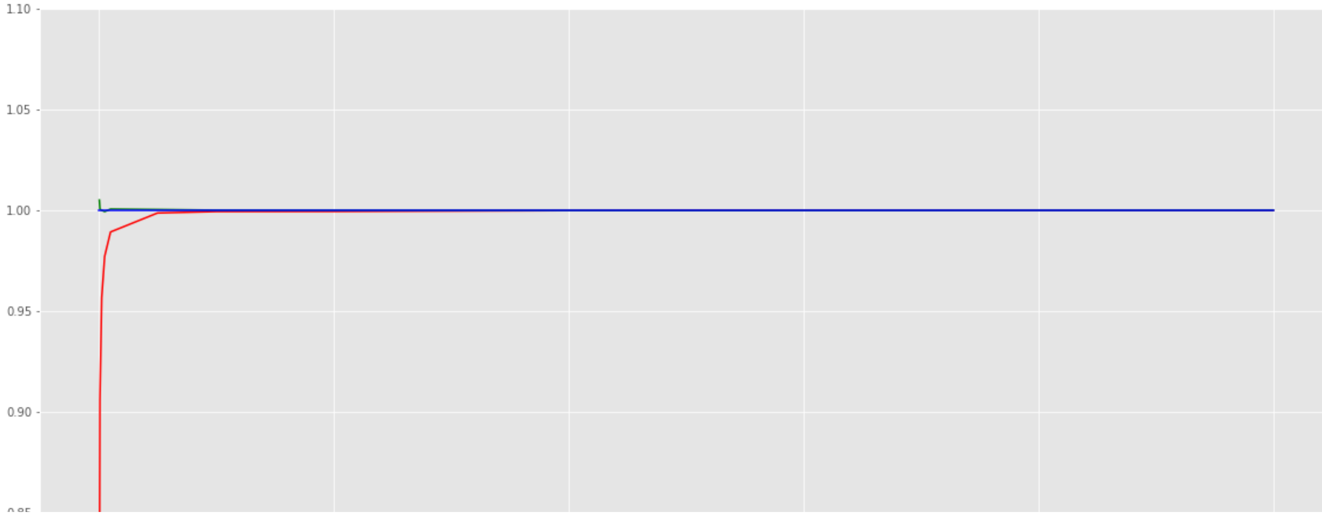
1. Write a simulation that veri es the asymptotic unbiasedness of the maximum likelihood estimator for the variance parameter of a univariate Gaussian distribution. Include a veri cation of the unbiasedness of the sample variance

```
import scipy.stats as rv
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec
import scipy.stats as stats
import numpy as np
import matplotlib.pyplot as plt
import math
plt.style.use('ggplot')
from progressbar import ProgressBar
```

```
mu=0
```

```
S =1
n=100000 #sample size
n_sim = 10000
s = range(n_sim)

samples = [1,5,10,25,50,100,500,1000,2000,5000,10000]

#simulation iterations
Sn_s = []
for SIZE in samples:
    size_res = []
    for i in range(n_sim):
        X = rv.norm.rvs(mu,np.sqrt(S), size = SIZE)
        size_res.append(np.var(X, ddof = 0))
    Sn_s.append(np.sum(np.array(size_res))/n_sim)


S_s = []
for SIZE2 in samples:
    size_res2 = []
    for i in range(n_sim):
        X = rv.norm.rvs(mu,np.sqrt(S), size = SIZE2)
        size_res2.append(np.var(X, ddof = 1))
    S_s.append(np.sum(np.array(size_res2))/n_sim)

fig = plt.figure(figsize = (20,8))
gs = GridSpec(1,2)
ax = {}

ax[0] = plt.subplot()

ax[0].plot (samples,
            Sn_s,
            color = 'r')
ax[0].plot (samples,
            S_s,
            color = 'g')
ax[0].plot (samples,
            1*np.ones(11),
            color = 'b')
ax[0].set_ylim(0.85,1.10)
```

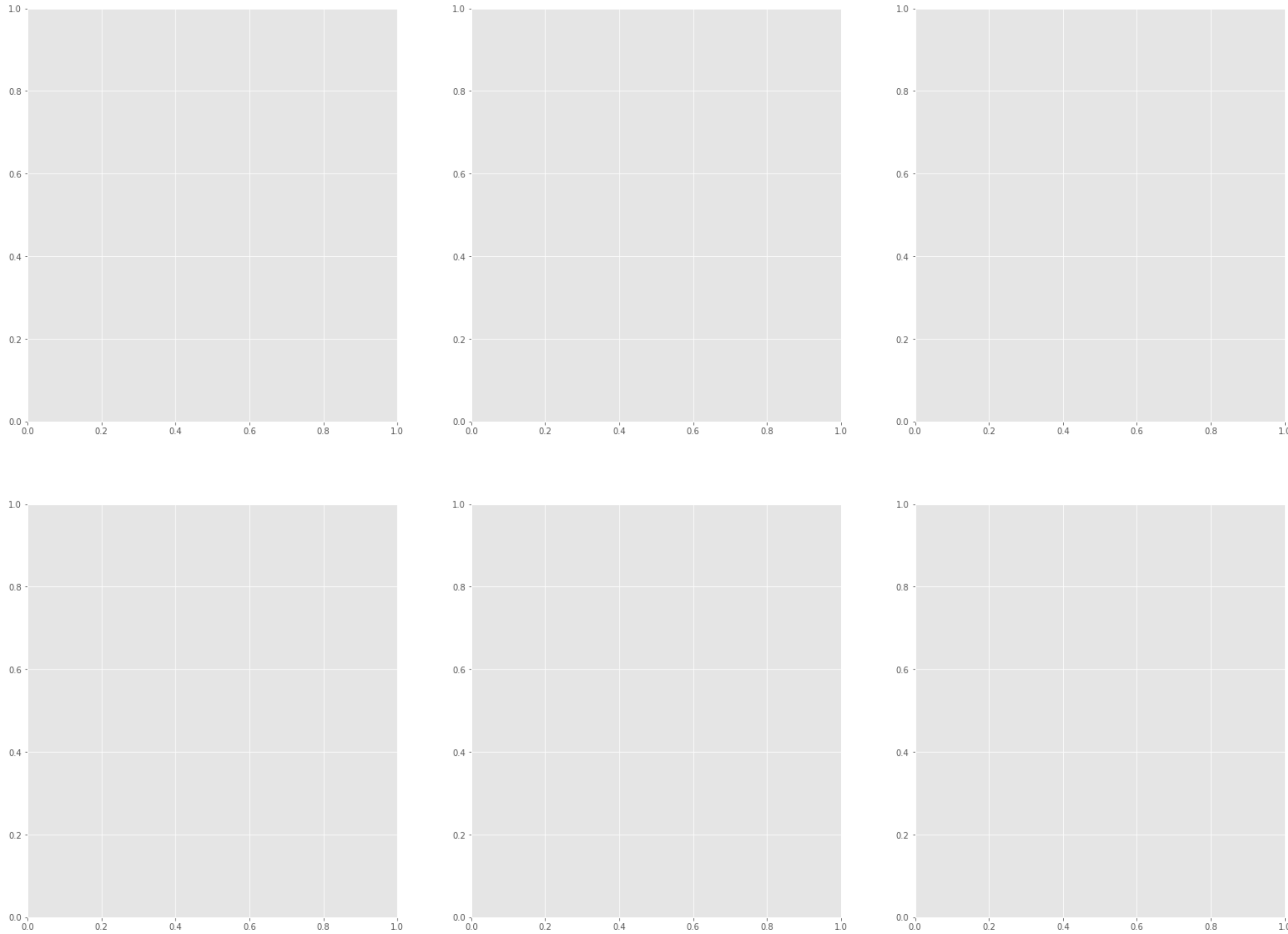
```
/usr/local/lib/python3.7/dist-packages/numpy/core/fromnumeric.py:3622: RuntimeWa
  **kwargs)
/usr/local/lib/python3.7/dist-packages/numpy/core/_methods.py:226: RuntimeWarnin
  ret = ret.dtype.type(ret / rcount)
(0.85, 1.1)
```




## 2) Write a simulation that verifies the asymptotic e ciency of the maximum likelihood estimator for the parameter of a Bernoulli distribution.

```
import scipy.stats as stats
import numpy as np
import matplotlib.pyplot as plt
import math
plt.style.use('ggplot')
from progressbar import ProgressBar


# Initializations

sample_sizes = [100,1000,10000,100000,1000000,1000000]
p = 0.7
repeats = 1000


#Setup the Plot

count_sample_sizes = len(sample_sizes)

rows = math.ceil(count_sample_sizes/3)
if count_sample_sizes < 3:
  cols = count_sample_sizes
else:
  cols = 3
```

```
fig, axs = plt.subplots(rows,cols, figsize=(9*3,math.ceil(count_sample_sizes/3)*10));
```

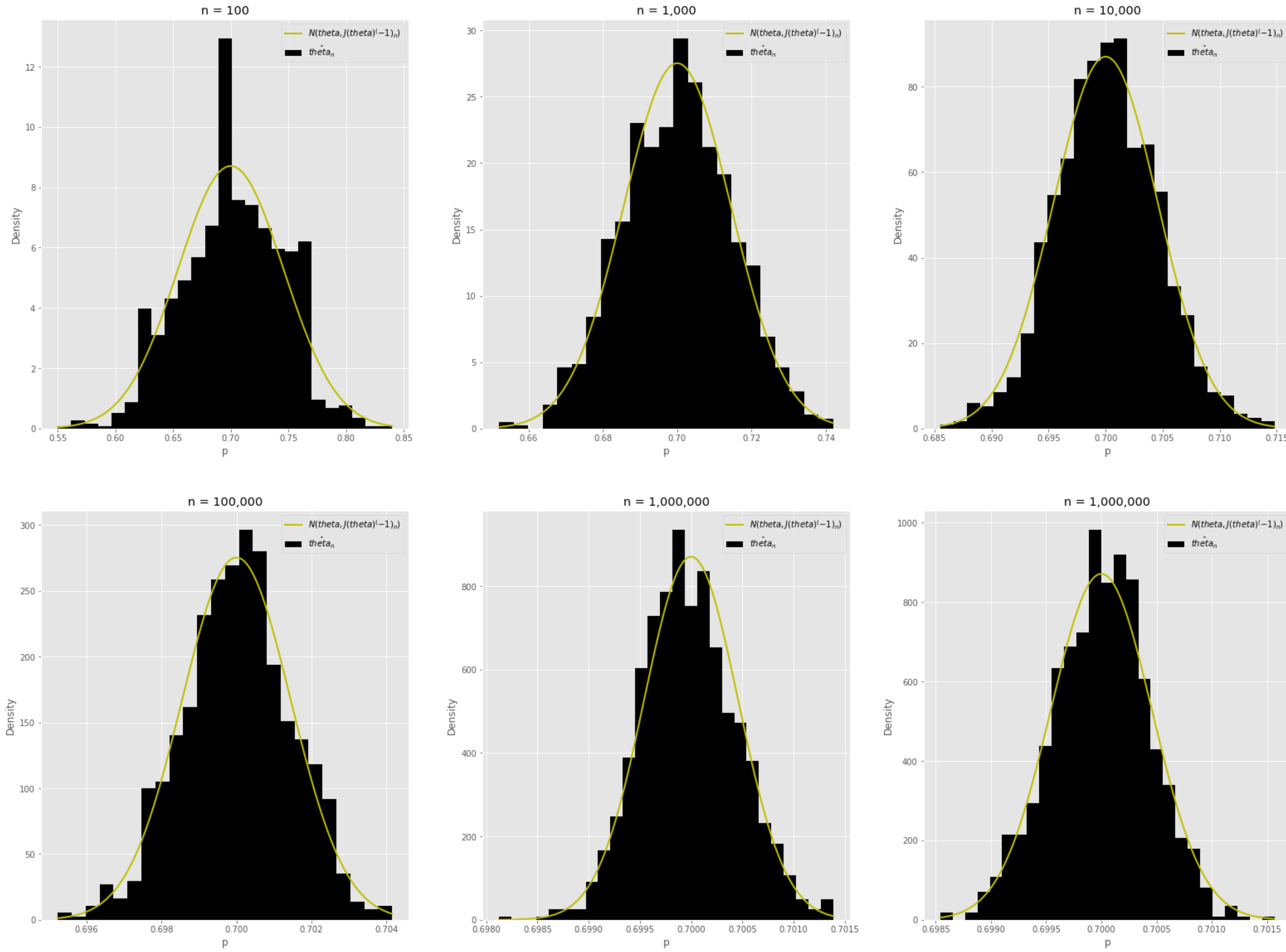


```
#Estimate the Bern parameter and plot PDF of normal dist and the p-estimations

for i in range(0, count_sample_sizes):
  n = sample_sizes[i]
  estimators = np.full(repeats, np.nan)
  pbar = ProgressBar()

  for j in pbar(range(repeats)):
```

```
        bernoulli_sample = stats.bernoulli.rvs(p, size=n)
        estimator = np.mean(bernoulli_sample)
        estimators[j] = estimator

    #define linear space

    x_min = min(estimators)
    x_max = max(estimators)
    x_res = 1000
    x_space = np.linspace(x_min,x_max,x_res)

    EFI = n/p + n/(1-p) # Expected Fisher info
    EFI_power_minus1 = EFI**(-1)

    #define plot labels
    label_norm = '$N(theta, J(theta)^(-1)_{n})$'
    label_estimators = '$\hat{theta}_n$'
    title_string = 'n = ' + '{:,}'.format(n)

    #define respective subplots
    x_ax = math.floor(i/3)
    y_ax = i % 3

    axs[x_ax, y_ax].plot(x_space,
                       stats.norm.pdf(x_space,p,math.sqrt(EFI_power_minus1)),
                       linewidth = 2,
                       color = 'y',
                       label = label_norm);
    axs[x_ax, y_ax].hist(estimators,
                      density=True,
                      bins='auto',
                      color = 'black',
                      label=label_estimators)
    axs[x_ax, y_ax].set_title(title_string)
    axs[x_ax, y_ax].set_xlabel('p')
    axs[x_ax, y_ax].set_ylabel('Density')
    axs[x_ax, y_ax].legend()
```

```
100% (1000 of 1000) |####################| Elapsed Time: 0:00:00 Time:  0:00:00
100% (1000 of 1000) |####################| Elapsed Time: 0:00:00 Time:  0:00:00
100% (1000 of 1000) |####################| Elapsed Time: 0:00:00 Time:  0:00:00
100% (1000 of 1000) |####################| Elapsed Time: 0:00:03 Time:  0:00:03
100% (1000 of 1000) |####################| Elapsed Time: 0:00:28 Time:  0:00:28
100% (1000 of 1000) |####################| Elapsed Time: 0:00:29 Time:  0:00:29
```

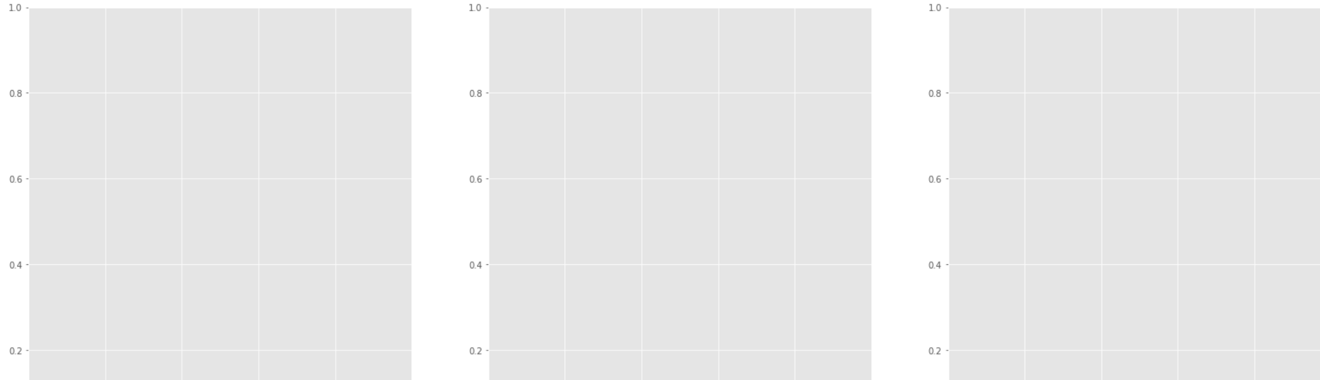
```
fig
```




## 3) Write a simulation that verifies the asymptotic efficiency of the maximum likelihood estimator for the variance parameter of a univariate Gaussian distribution.

```
import scipy.stats as stats
import numpy as np
import matplotlib.pyplot as plt
import math
plt.style.use('ggplot')
from progressbar import ProgressBar
```

```
# Initializations

sample_sizes = [100,1000,10000,100000,1000000,1000000]
p = 0.7
repeats = 1000



#Setup the Plot

count_sample_sizes = len(sample_sizes)

rows = math.ceil(count_sample_sizes/3)
if count_sample_sizes < 3:
  cols = count_sample_sizes
else:
  cols = 3

fig, axs = plt.subplots(rows,cols, figsize=(9*3,math.ceil(count_sample_sizes/3)*10));
```



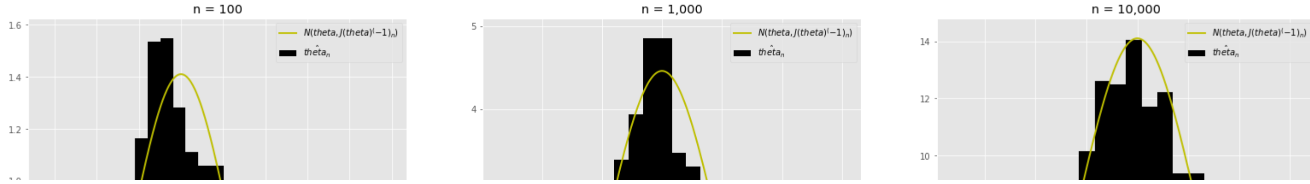
```
mu = 0.7
S  = 2
#Estimate the S_sqr parameter and plot PDF of normal dist

for i in range(0, count_sample_sizes):
  n = sample_sizes[i]
  estimators = np.full(repeats, np.nan)
  pbar = ProgressBar()

  for j in pbar(range(repeats)):
    gaussian_sample = stats.norm.rvs(mu,np.sqrt(S),size=n)
    estimator = np.var(gaussian_sample, ddof = 0)
    estimators[j] = estimator

  #define linear space

  x_min = min(estimators)
  x_max = max(estimators)
  x_res = 1000
  x_space = np.linspace(x_min,x_max,x_res)

  EFI = (2*S**2)/n # Expected Fisher info

  #define plot labels
  label_norm = '$N(theta, J(theta)^(-1)_{n})$'
  label_estimators = '$\hat{theta}_n$'
  title_string = 'n = ' + '{:,}'.format(n)

  #define respective subplots
  x_ax = math.floor(i/3)
  y_ax = i % 3

  axs[x_ax, y_ax].plot(x_space,
                   stats.norm.pdf(x_space,S,math.sqrt(EFI)),
                   linewidth = 2,
                   color = 'y',
                   label = label_norm);
  axs[x_ax, y_ax].hist(estimators,
                  density=True,
                  bins='auto',
                  color = 'black',
```

```
                    color='black',
                    label=label_estimators)
  axs[x_ax, y_ax].set_title(title_string)
  axs[x_ax, y_ax].set_xlabel('p')
  axs[x_ax, y_ax].set_ylabel('Density')
  axs[x_ax, y_ax].legend()
```

```
100% (1000 of 1000) |####################| Elapsed Time: 0:00:00 Time:  0:00:00
100% (1000 of 1000) |####################| Elapsed Time: 0:00:00 Time:  0:00:00
100% (1000 of 1000) |####################| Elapsed Time: 0:00:00 Time:  0:00:00
100% (1000 of 1000) |####################| Elapsed Time: 0:00:05 Time:  0:00:05
100% (1000 of 1000) |####################| Elapsed Time: 0:00:49 Time:  0:00:49
100% (1000 of 1000) |####################| Elapsed Time: 0:00:49 Time:  0:00:49
```

```
print('N(theta, J(theta)^(-1)_n' + ' ~ N( Sigma**2, (2*Sigma**4)/n)')
fig
```

```
N(theta, J(theta)^(-1)_n ~ N( Sigma**2, (2*Sigma**4)/n)
```



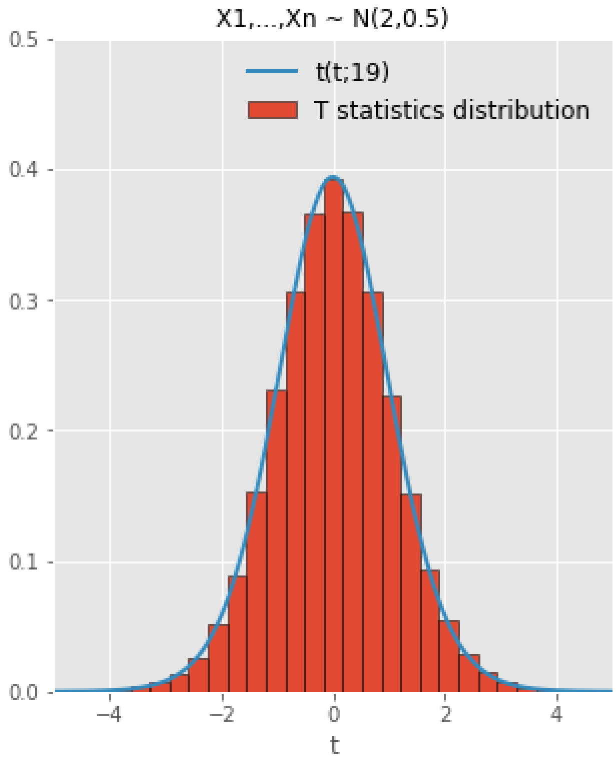

## (11) Confidence intervals

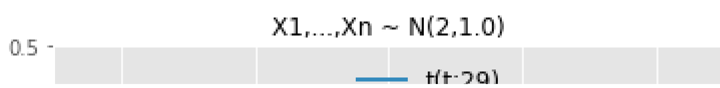
### 1. Write a simulation that veri es that the T statistic is distributed according to a t-distribution with n − 1 degrees of freedom.

```
import scipy.stats as stats
import numpy as np
import warnings
import matplotlib.pyplot as plt
% matplotlib inline
```

```
def tscoredist(mean,variance, n):
  sample= stats.norm.rvs(mean, np.sqrt(variance), size = n)
  return np.sqrt(n)*(np.mean(sample) - mean) / np.sqrt(np.var(sample, ddof = 1))


#Initialisation
n_sim = 100000
t_min = -5
t_max = 5
t_res = 1000
t = np.linspace(t_min,t_max,t_res)
mu = np.array ([2,2,1,0])
S = np.array ([0.5,1,2,3])
n_all = np.array ([20,30,40,50])
_ = plt.figure(num = 2, figsize = (5,6))
for i,n in enumerate(n_all):
  ts = np.full([n_sim, 1], np.nan)
  for j in range(n_sim):
      ts[j] = tscoredist(mu[i], S[i], n)
  _ = plt.hist(ts,density=True,
                bins = np.linspace(t_min,t_max,30),
                edgecolor = 'black', linewidth = .5,
               label = r'T statistics distribution')
  _ = plt.plot(t,
                stats.t.pdf(t,n-1),
```

```
              linewidth = 2,
              label = r't(t;{})'.format(n-1))
_ = plt.title(r'X1,...,Xn ~ N({},{})'.format(mu[i],S[i]), fontsize = 12)
_ = plt.xlabel(r't', fontsize = 12)
_ = plt.xlim(t_min,t_max)
_ = plt.ylim(0,.5)
_ = plt.legend(loc = 'upper right', fontsize = 12, frameon = False)
plt.show()
```



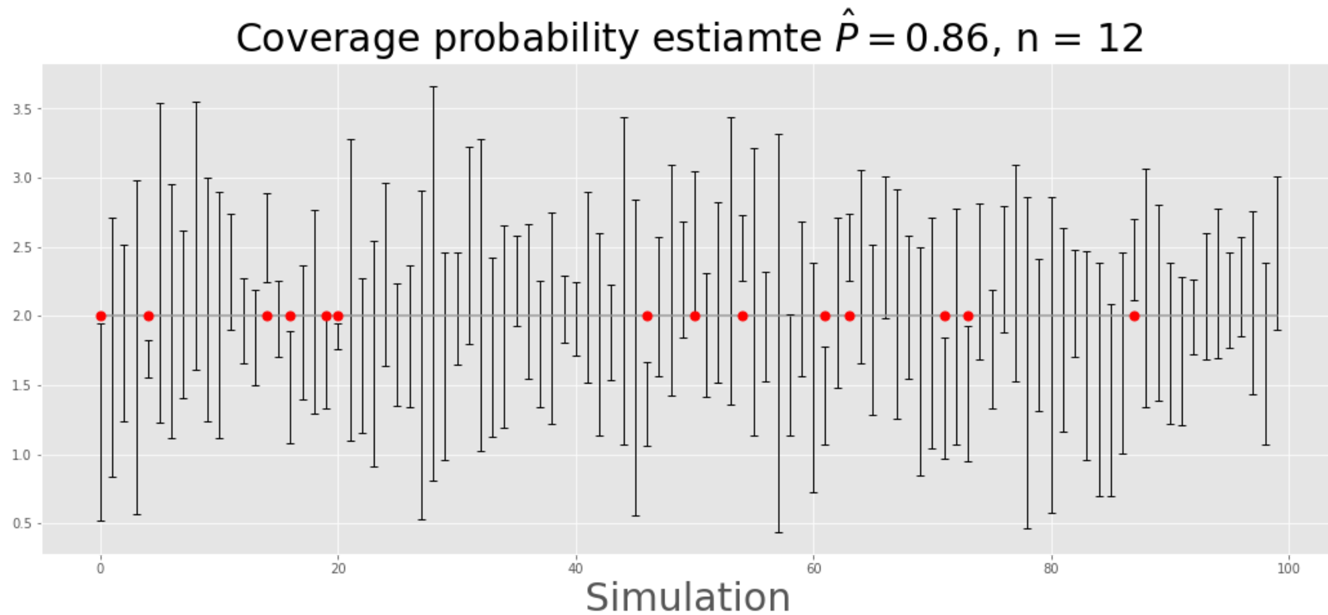

## 2. Write a simulation that veri es that the 95%-con dence interval for the expectation parameter of a Gaussian distribution with unknown variance comprises the true, but unknown, expectation parameter in ≈ 95% of its realizations.


```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec
import scipy.stats as rv
```

```
ıe but unknown exp parameter
# true but unknown var parameter
ɔle size
5 # Confidence level
ɔpf((1+delta)/2,n-1) #from the formula psi^(-1)[(1+delta)/2 with n-1 degrees of freedc
 # no. of simulations
_sim)
full([n_sim,1], np.nan) #sample mean
([n_sim,1], np.nan) #sample std dev
```

```
ull([n_sim,1], np.nan)
([n_sim,2], np.nan) #CI upper n lower boundaries
).full([n_sim,1], np.nan) #Confidence condition


ge(n_sim):
m.rvs(mu,np.sqrt(sigsqr), size = n)
= np.mean(X)
.var(X,ddof = 1)
= t_2*(S[i]/np.sqrt(n))
_bar[i] - gamma[i]
_bar[i] + gamma[i]

C[i,0] and mu <= C[i,1]:
i] = 1

i] = 0

np.argwhere(mu_in_c == False)

on
igure(figsize = (18,7))

.subplot()
s,
mu*np.ones([n_sim,1]),
color = [.6,.6,.6])
bar(s,
    X_bar,
    xerr = None,
    yerr = gamma,
    linestyle ='',
    linewidth = 1,
    capsize = 3,
    color = [0,0,0])
 mu_nin_c[:,0],
2*np.ones([len(mu_nin_c),1]),
ls = '',
marker ='o',
ms = 7,
mfc = 'r',
mec = 'r')
label('Simulation', fontsize = 30)
itle('Coverage probability estiamte $\hat{{P}} = ${0:1.2f}, n = {1:1.0f}'.format(np.me
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:44: MatplotlibDepre
Text(0.5, 1.0, 'Coverage probability estiamte $\\hat{P} = $0.86, n = 12')
```

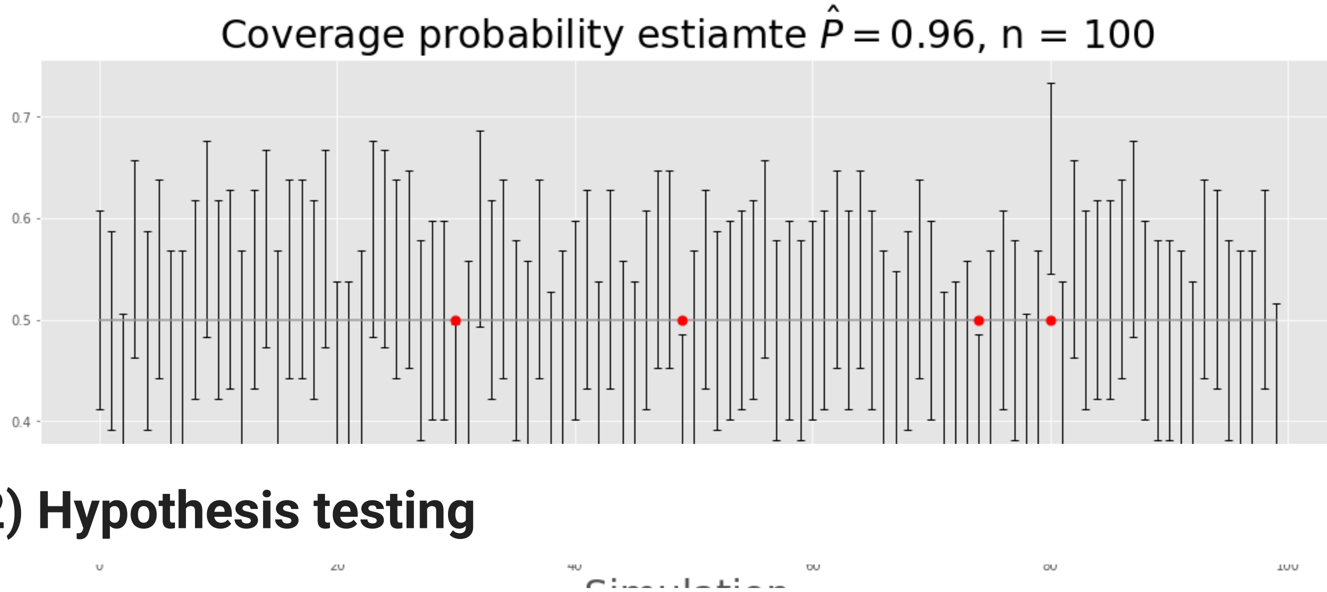



3. Write a simulation that veri es that the approximate 95%-con dence interval for the expectation parameter of a Bernoulli distribution comprises the true, but unknown, expectation parameter in ≈ 95% of its realizations.

```
 = 0.5 # True but unknown exp parameter ~ N(0,1)
= 100 # sample size
lta = 0.95 # Confidence level
delta = rv.norm.ppf((1+delta)/2,0,1) # from formula ~N(0,1)
sim = np.int(1e2)  #simulations
= range(n_sim)
_hat = np.full([n_sim,1], np.nan)
inv = np.full([n_sim,1], np.nan) # 1/expected fisher info
= np.full([n_sim,2], np.nan) # Confidence interval
_in_c = np.full([n_sim,1], np.nan) # Confidence condition

imulation
 i in range(n_sim):
X = rv.bernoulli.rvs(mu, size = n)
mu_hat[i] = np.mean(X)
J_inv[i] = mu_hat[i]*(1-mu_hat[i])/n
C[i,0] = mu_hat[i] - np.sqrt(J_inv[i])*z_delta
```

```
:[i,i] = mu_hat[i] + np.sqrt(J_inv[i])*z_delta

if mu >= C[i,0] and mu <= C[i,1]:
  mu_in_c[i] = 1
else:
  mu_in_c[i] = 0

ases where confidence interval (CI) is not covering mu
_nin_c = np.argwhere(mu_in_c == False)

isualization
g = plt.figure(figsize = (18,7))
 = {}
[0] = plt.subplot()
[0].plot ( s,
          mu*np.ones([n_sim,1]),
          color = [.6,.6,.6])
[0].errorbar( s,
              mu_hat,
              xerr = None,
              yerr = np.sqrt(J_inv)*z_delta,
              linestyle = '',
              linewidth = 1,
              capsize = 3,
              color = [0,0,0]
              )
[0].plot( mu_nin_c[:,0],
         mu*np.ones([len(mu_nin_c),1]),
         ls = '',
         marker ='o',
         ms = 7,
         mfc = 'r',
         mec = 'r')
[0].set_xlabel('Simulation', fontsize = 30)
[0].set_title('Coverage probability estiamte $\hat{{P}} = ${0:1.2f}, n = {1:1.0f}'.for
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:42: MatplotlibDepre
Text(0.5, 1.0, 'Coverage probability estiamte $\\hat{P} = $0.96, n = 100')
```



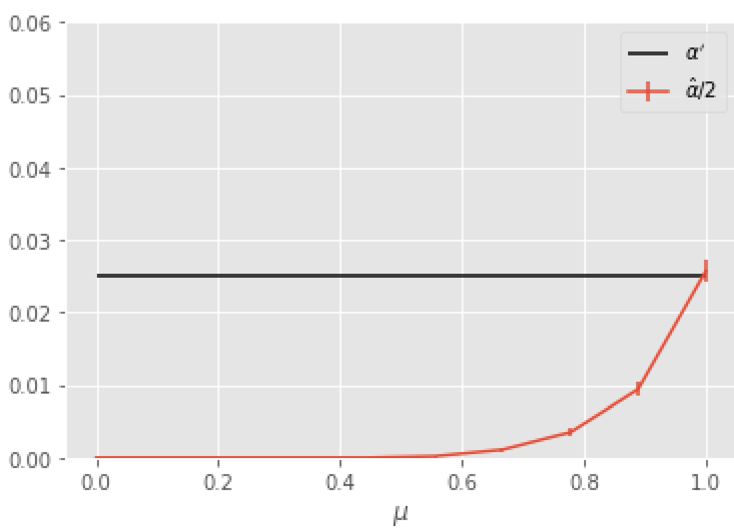

## (12) Hypothesis testing

1. By means of simulation, show that a two-sided T test with simple null hypothesis Θ0 := {μ0} of signi cance level α 0 is exact.

```
repeats = 10000
alpha  = 0.05
sample_size = 25

mu_0 = 1
theta = np.linspace(0,mu_0, 10)
alpha_est = np.full([10,2], np.nan)   # the outputs for thetas
t_stats = np.full(repeats, np.nan)


sigmasqr = 2
df = sample_size-1
c_alpha_prime  = rv.t.ppf(1 - alpha/2, df)  #Critical value for  two tailed test - al

for k, mu in np.ndenumerate(theta):
  test_result = np.full(repeats,np.nan)
  for s in range(repeats):
    X = rv.norm.rvs(mu, np.sqrt(sigmasqr),size=sample_size)
    mean_of_sample = np.mean(X)
    std_dev = np.sqrt(np.var(X, ddof = 1))
    t_stats[s] = np.sqrt(sample_size)*((mean_of_sample - mu_0)/std_dev)

    if t_stats[s] >= c_alpha_prime:  # reject null hypothesis
      test_result[s] = 1
    else:
```

```
      test_result[s] = 0

  # test size
  alpha_est[k[0],0] = np.mean(test_result)
  alpha_est[k[0],1] = np.std(test_result, ddof = 1)/ np.sqrt(repeats)


plt.hlines(alpha/2, theta[0], theta[-1], label = r'$\alpha^\prime$')
#plt.plot(theta,alpha_est[:,0], label = r'$\hat{\alpha}$' )
plt.errorbar(theta ,alpha_est[:,0], alpha_est[:,1],
                       label = r'$\hat{\alpha}/2$')
plt.xlabel('$\mu$')
plt.ylim(0, 0.06)
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7f4ea5565f10>
```




## 2) By means of simulation, demonstrate that the δ-confidence interval-based test for the expectation parameter of univariate Gaussian distribution is of significance level $\alpha' = 1 - \delta$.

```
repeats = 1000
n=25
mu_0 = 1
S = 2
delta = 0.95 #Confidence level
t_delta = rv.t.ppf((1+delta)/2,n-1)
C = np.full([repeats,2], np.nan) #Confidence boundaries
result = np.full([repeats,1], np.nan)
```

```
for i in range(repeats):
  X = rv.norm.rvs(mu_0,np.sqrt(S), size = n)
  X_bar = np.mean(X)
  S = np.sqrt(np.var(X, ddof = 1))
  C[i,0] = X_bar - t_delta*(S/np.sqrt(n))
  C[i,1] = X_bar + t_delta*(S/np.sqrt(n))

  if mu_0 >= C[i,0] and mu_0 <= C[i,1]:
    result[i] = 0
  else:
    result[i] = 1

alphaprime = np.mean(result)
print(" Significance level :", alphaprime, "1 - delta :", 1-delta , "are equal")
```

```
 Significance level : 0.038 1 - delta : 0.050000000000000044 are equal
```

## (13) Conjugate inference

### 1. For n = 10, implement batch and recursive Bayesian estimation for the Beta-Binomial model. Compare the results based on identical samples.

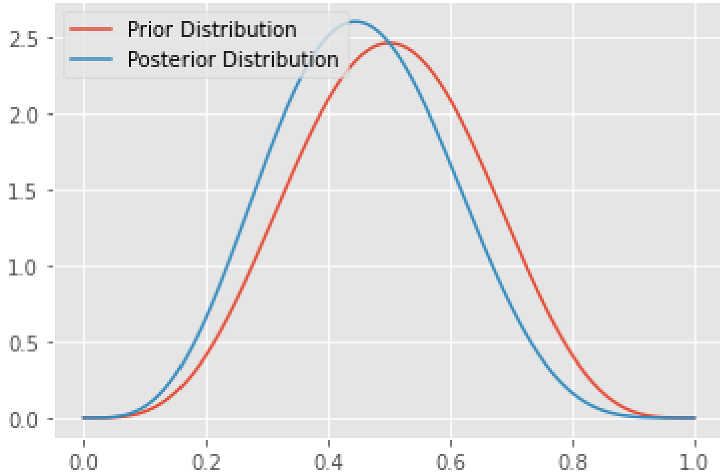
```
#Load in required modules
import pandas as pd
from scipy.stats import beta
from scipy.stats import binom
import numpy as np
import matplotlib.pyplot as plt


class beta_dist:
    def __init__(self, a = 1, b = 1):
        self.a = a
        self.b = b

    #Get the beta pdf
    def get_pdf(self):
        x = np.linspace(0, 1, 1000)
        fx = beta.pdf(x, self.a, self.b)
        dens_dict = {'x': x, 'fx': fx}
        return(dens_dict)

    #Update parameters:
    def update_beta_params(self, n, num_successes):
```

```
    def update_beta_params(self, n, num_successes):
        self.old_a = self.a
        self.old_b = self.b
        self.a = self.a + num_successes
        self.b = self.b + n - num_successes
```

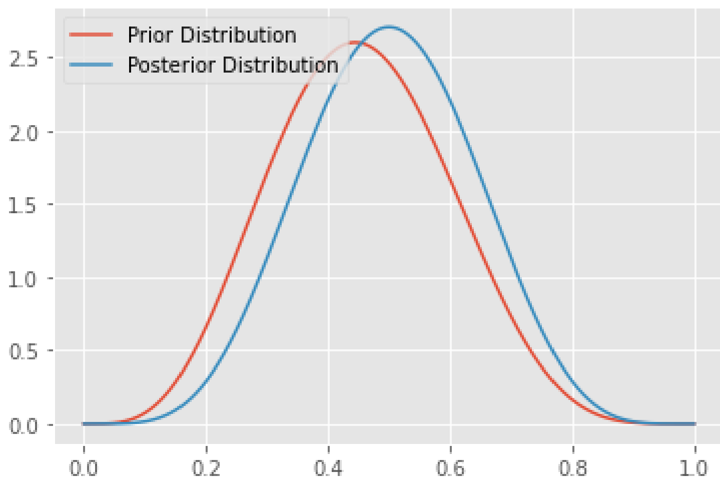
## Recursive Implementation

```
success_list = [0,1,0,0,1,0,1,0,1,0] #Observations  n=10

#Updation Description when done recursively:

#Prior = (5,5)
#Update formula: a+x,b+n-x
#Update 1:   5+0,5+1-0  =  5 6  beta+
#Update 2:   5+1,6+1-1  =  6 6  alpha+
#Update 3:   6+0,6+1-0  =  6 7  beta+
#Update 4:   6+0,7+1-0  =  6 8  beta+
#Update 5:   6+1,8+1-1  =  7 8  alpha+
#Update 6:   7+0,8+1-0  =  7 9  beta+
#Update 7:   7+1,9+1-1  =  8 9  alpha+
#Update 8:   8+0,9+1-0  =  8 10 beta+
#Update 9:   8+1,10+1-1 =  9 10 alpha+
#Update 10:  9+0,10+1-0 =  9 11 beta+

dist = beta_dist(a = 5, b = 5)
prior = dist.get_pdf()
for x in success_list:
  dist.update_beta_params( n = 1, num_successes = x)
  posterior = dist.get_pdf()
  print("The updated hyperparameters are:")
  print(dist.a,dist.b)

  #Plot prior and posterior
  plt.plot(prior['x'], prior['fx'])
  plt.plot(posterior['x'], posterior['fx'])
  plt.legend(['Prior Distribution', 'Posterior Distribution'], loc='upper left')
  plt.show()
  prior = posterior
```

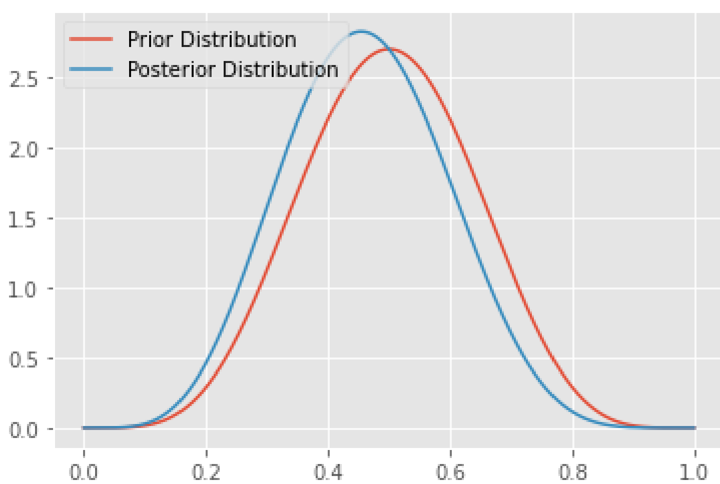
```
The updated hyperparameters are:
5 6
```



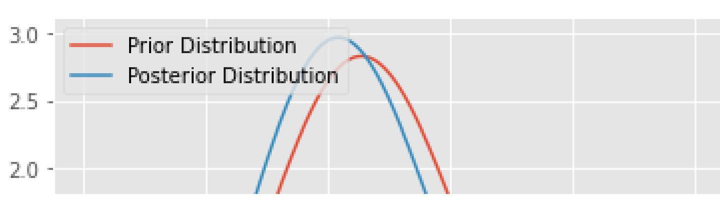

```
The updated hyperparameters are:
6 6
```

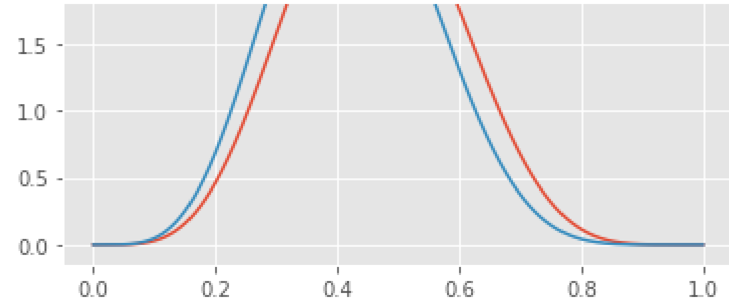



```
The updated hyperparameters are:
6 7
```

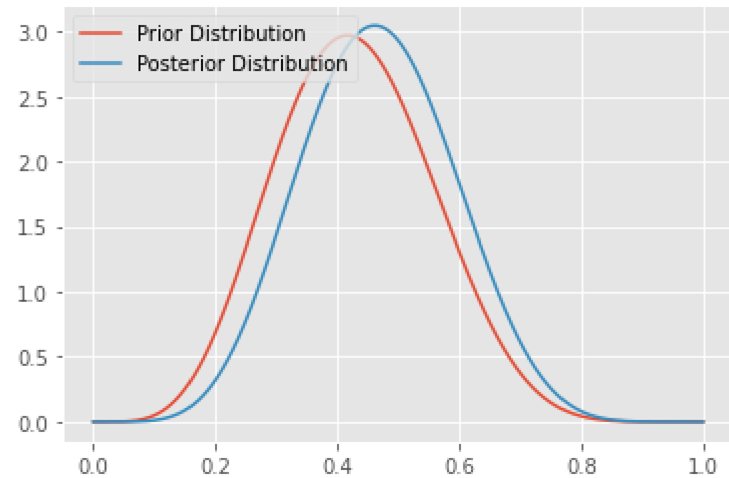



```
The updated hyperparameters are:
6 8
```

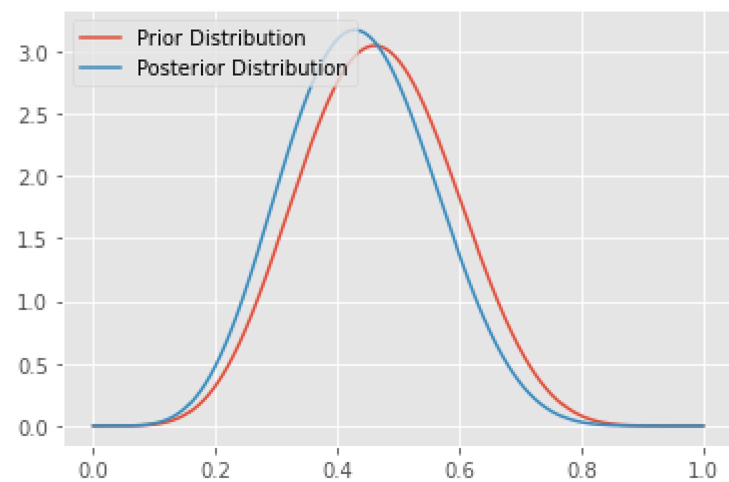




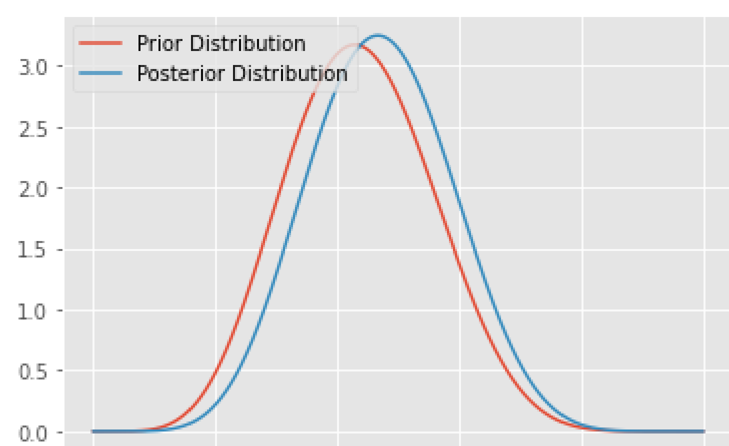
```
The updated hyperparameters are:
7 8
```



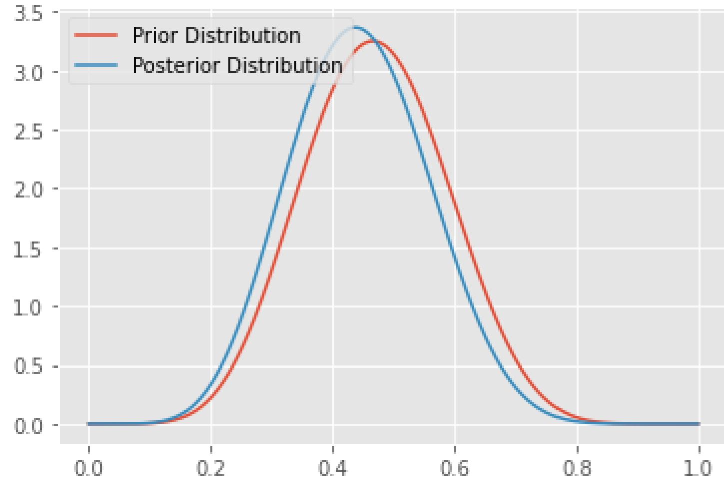

```
The updated hyperparameters are:
7 9
```



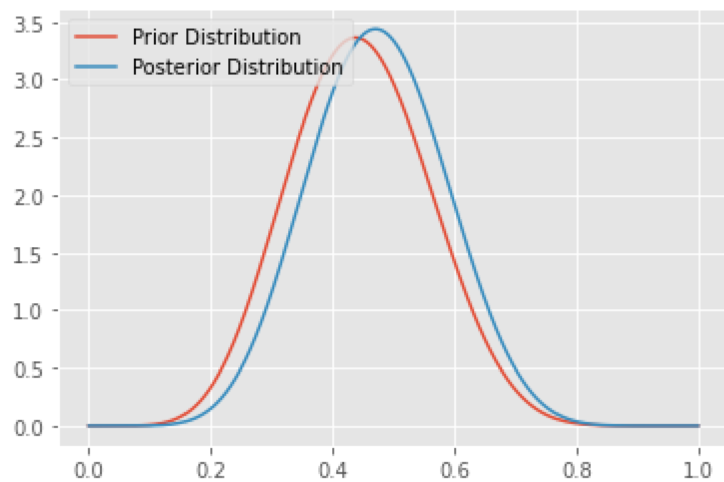

```
The updated hyperparameters are:
8 9
```



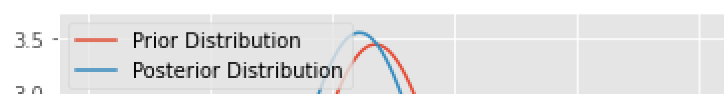
```
The updated hyperparameters are:
8 10
```




```
The updated hyperparameters are:
9 10
```




```
The updated hyperparameters are:
9 11
```

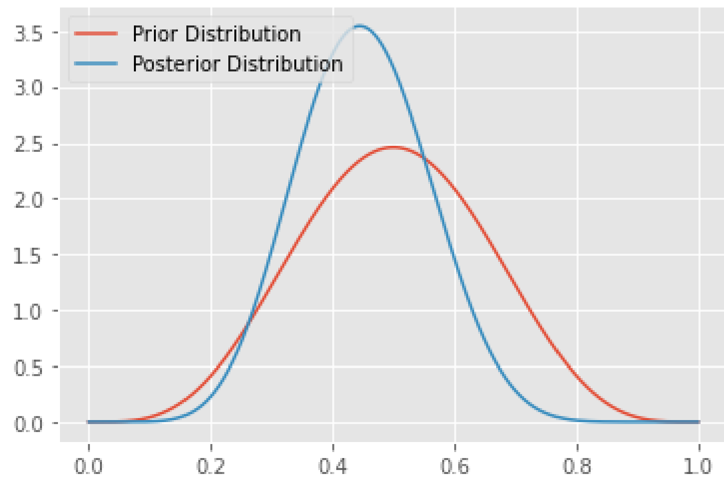



## Batch Implementation

```
dist = beta_dist(a = 5, b = 5)
prior = dist.get_pdf()

success_list = [0,1,0,0,1,0,1,0,1,0]

#Updation Description when processed in one batch:
#Prior = (5,5)
#Update formula: a+x,b+n-x
#                where x=sum of all the x's and n= total no. of trials
#                i.e   x=sum[0,1,0,0,1,0,1,0,1,0] = 4   & n = 10

#Update o/p:     5+4,5+10-4  =  9,11
```

```
#Obtain data to update hyperparameters

#Update hyperparameters
dist.update_beta_params(n = len(success_list), num_successes = sum(success_list))
posterior = dist.get_pdf()
print("The updated hyperparameters are:")
print(dist.a, dist.b)

#Plot prior and posterior
plt.plot(prior['x'], prior['fx'])
plt.plot(posterior['x'], posterior['fx'])
plt.legend(['Prior Distribution', 'Posterior Distribution'], loc='upper left')
plt.show()
```

```
The updated hyperparameters are:
9 11
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:17: DeprecationWarn
```

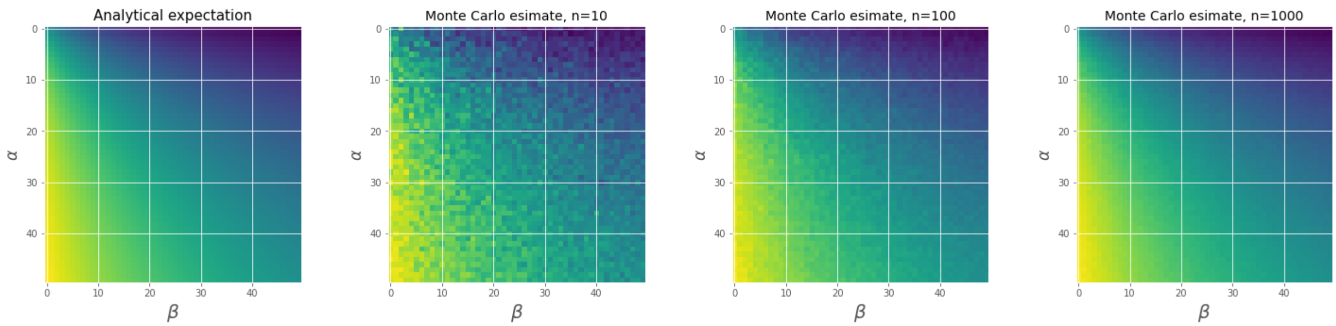



**Conclusion: Batch and recursive bayesian implementation yeilds similar results for beta binomial model. **

## (14) Numerical methods

1. Estimate the expected value of a Beta(α, β) for varying values of α and β by means of Monte Carlo integration by using a Beta distribution random number generator. Compare the results to the true expected values.

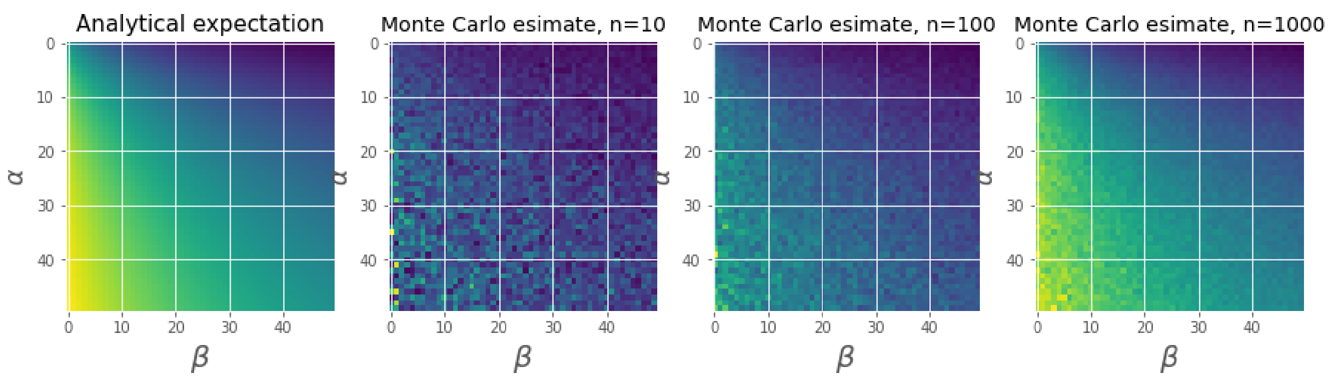
```
import scipy.stats as rv
import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec
import numpy as np




#Initialisation
ns = np.array([10,100,1000]) #diff values of n
aspace = np.linspace(1,10,50) #alpa space
bspace = np.linspace(1,10,50) #beta space
Exp = np.full([50,50,len(ns)+1], np.nan) #Store analytical, numerical expectation wit

#Parameter space iterations
for i, alpha in np.ndenumerate(aspace):
  for j, beta in np.ndenumerate(bspace):

    #analytical expectation
    Exp[i,j,0] = alpha / (alpha + beta)

    #Monte Carlo Estimate sample size iterations
    for k, n in np.ndenumerate(ns):

      #Monte Carlo Estimate
      Exp[i,j,k[0]+1] = np.mean(rv.beta.rvs(alpha, beta, size = n))


#Visualization
fig = plt.figure(figsize = (25,5))
gs = GridSpec(1,4)
id = 0
ax = {}
subplotlab = ['Analytical expectation','Monte Carlo esimate, n={}']

for i in range(4):
  ax[i] = plt.subplot(gs[i])
  ax[i].imshow(Exp[:,:,i], cmap='viridis')
  ax[i].xticks = np.linspace(1,10,10)
  ax[i].yticks = np.linspace(1,10,10)

  if i == 0:
      ax[i].set_title(subplotlab[0], fontsize = 15)
  else:
      ax[i].set_title(subplotlab[1].format(ns[i-1]), fontsize = 14)

  ax[i].set_xlabel(r'$\beta$',fontsize = 20)
  ax[i].set_ylabel(r'$\alpha$',fontsize = 17)
```




# 2. Estimate the expected value of a Beta(α, β) for varying values of α and β by means of Monte Carlo integration using an importance sampling scheme and a uniform random number generator.

```
#Initialisation
ns = np.array([10,100,1000]) #diff values of n
aspace = np.linspace(1,10,50) #alpa space
bspace = np.linspace(1,10,50) #beta space
Exp = np.full([50,50,len(ns)+1], np.nan) #Store analytical, numerical expectation wit


#Parameter space iterations
for i, alpha in np.ndenumerate(aspace):
  for j, beta in np.ndenumerate(bspace):

    #analytical expectation
    Exp[i,j,0] = alpha / (alpha + beta)

    #Monte Carlo Estimate sample size iterations
    for k, n in np.ndenumerate(ns):

      #Importance sampling Mone Carlo Estimate
      X = rv.uniform.rvs(size=n)
      I_hat_n = 1/n * np.sum(X*(rv.beta.pdf(X,alpha,beta)))
      Exp[i,j,k[0]+1] = I_hat_n

#Visualization
fig = plt.figure(figsize = (15,5))
gs = GridSpec(1,4)
id = 0
ax = {}
subplotlab = ['Analytical expectation','Monte Carlo esimate, n={}']
```

```
for i in range(4):
  ax[i] = plt.subplot(gs[i])
  ax[i].imshow(Exp[:,:,i], cmap='viridis')
  ax[i].xticks = np.linspace(1,10,10)
  ax[i].yticks = np.linspace(1,10,10)

  if i == 0:
      ax[i].set_title(subplotlab[0], fontsize = 15)
  else:
      ax[i].set_title(subplotlab[1].format(ns[i-1]), fontsize = 14)

  ax[i].set_xlabel(r'$\beta$',fontsize = 20)
  ax[i].set_ylabel(r'$\alpha$',fontsize = 17)
```




## 3) Use an acceptance-rejection algorithm to sample random numbers from Beta(2, 6).

```
yspace = np.linspace(0,1,1000) #Random varialbe space

#target density parameters
alpha = 2
beta = 6

#Proposal density paramters
mu = 0
sigsqr = 1
c = 8

#acceptance rejection algorithm
n=10000
Y = np.full([n,1],np.nan)
```