Reinforcement Learning: Homework #1

Due on March 22, 2020 at 11:59pm

Professor Ziyu Shao

Yiwei Yang
2018533218

Problem 1

Solution

(a) Logistic Distribution

The logistic distribution is given by:

$$X\ Logistic(\mu, s),\ s > 0$$

Here is called the location parameter and s is called the scale parameter. The location parameters makes the PDF slide along X axis, the scale paramter makes the PDF fat or skinny. The PDF function can be described as:

$$f_X(x) = \frac{e^{-\frac{x-\mu}{s}}}{s(1+e^{-\frac{x-\mu}{s}})^2},\ \text{where } -\infty < x < \infty$$

[1]:

```
import numpy as np
from sys import exit
import math
import matplotlib.pyplot as plt
from take_max import take_max
from take_mean import take_mean
from take_min import take_min
from take_variance import take_variance
```

[2]:

```
def uniform ( seed ):
  i4_huge = 2147483647
  seed = int ( seed )
  seed = ( seed % i4_huge )
  if ( seed < 0 ):
    seed = seed + i4_huge
  k = ( seed // 127773 )
  seed = 16807 * ( seed - k * 127773 ) - k * 2836
  if ( seed < 0 ):
    seed = seed + i4_huge
  r = seed * 4.656612875E-10
  return r, seed
```

[3]:

```
def logistic_cdf ( x, a, b ):
  cdf = 1.0 / ( 1.0 + np.exp ( ( a - x ) / b ) )
  return cdf
```

[4]:

```
def logistic_cdf_inv ( cdf, a, b ):
  if ( cdf < 0.0 or 1.0 < cdf ):
    exit ( 'LOGISTIC_CDF_INV - Fatal error!' )
  x = a - b * np.log ( ( 1.0 - cdf ) / cdf )
  return x
```

[5]:

```
def logistic_cdf_test ( ):
  a = 0.0
```

Problem 1 continued on next page. . .

```
    b = 1.0
    seed = 123456789

    for i in range ( 0, 10 ):
      x, seed = logistic_sample ( a, b, seed )
      pdf = logistic_pdf ( x, a, b )
      cdf = logistic_cdf ( x, a, b )
      x2 = logistic_cdf_inv ( cdf, a, b )
      print ( ' %14g  %14g  %14g  %14g' % ( x, pdf, cdf, x2 ) )
    return
```

[6]:
```
def logistic_pdf ( x, a, b ):
  temp = np.exp ( ( a - x ) / b )
  pdf = temp / ( b * ( 1.0 + temp ) ** 2 )
  return pdf
```

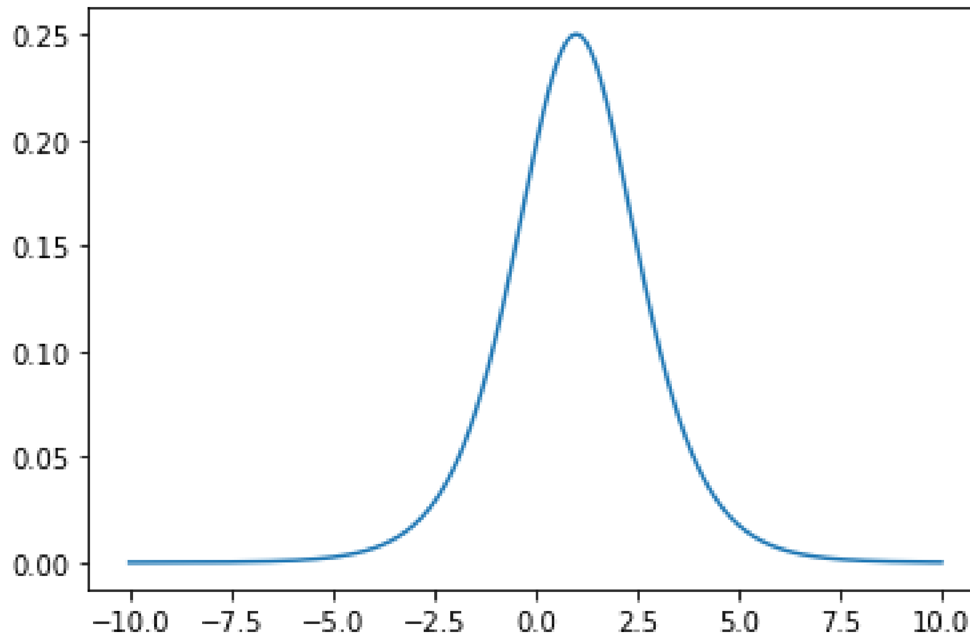
[7]:
```
def logistic_sample ( a, b, seed ):
  cdf, seed = uniform ( seed )
  x = logistic_cdf_inv ( cdf, a, b )
  return x, seed
```

[8]:
```
def logistic_variance ( a, b ):
  variance = ( np.pi * b ) ** 2 / 3.0
  return variance
```

[9]:
```
def logistic_mean ( a, b ):
  mean = a
  return mean
```

[10]:
```
def logistic_draw_picture(  ):
    x_samples = np.arange(-10,10,0.01)
    plt.plot(x_samples,logistic_pdf ( x_samples, 1, 1 ))
    plt.show()
```

[11]:
```
logistic_draw_picture()
```



[12]:
```
def logistic_sample_test ( ):
  nsample = 100000
  seed = 123456789

  print ( '' )
  print ( 'LOGISTIC_SAMPLE_TEST' )
  print ( '  LOGISTIC_MEAN computes the Logistic mean' )
  print ( '  LOGISTIC_SAMPLE samples the Logistic distribution' )
  print ( '  LOGISTIC_VARIANCE computes the Logistic variance.' )

  a = 1.0
  b = 1.0
  randvars = []

  mean = logistic_mean ( a, b )
  variance = logistic_variance ( a, b )

  print ( '' )
  print ( '  PDF parameter A =             %14g' % ( a ) )
  print ( '  PDF parameter B =             %14g' % ( b ) )
  print ( '  PDF mean =                    %14g' % ( mean ) )
  print ( '  PDF variance =                %14g' % ( variance ) )

  x = np.zeros ( nsample )
  for i in range ( 0, nsample ):
    x[i], seed = logistic_sample ( a, b, seed )
    randvars.append(x[i])
```

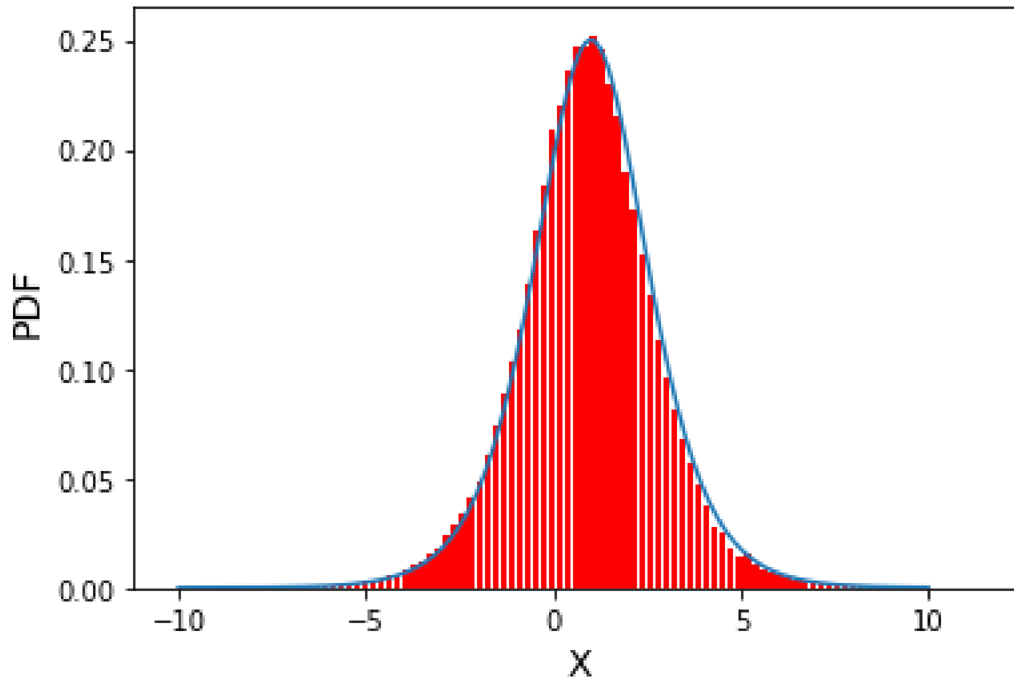
```
    mean = take_mean ( nsample, x )
    variance = take_variance ( nsample, x )
    xmax = take_max ( nsample, x )
    xmin = take_min ( nsample, x )

    x_samples = np.arange(-10,10,0.01)
    plt.plot(x_samples,logistic_pdf ( x_samples, 1, 1 ))
    plt.hist(randvars, bins=100, normed=True,
                 weights=None, cumulative=False, bottom=None,
                 histtype=u'bar', align=u'left', orientation=u'vertical',
                 rwidth=0.8, log=False, color="red", label=None, stacked=False)

    plt.xlabel('X',fontsize=14)
    plt.ylabel('PDF',fontsize=14)
    plt.show()

    print ( '' )
    print ( '  Sample size =     %6d' % ( nsample ) )
    print ( '  Sample mean =     %14g' % ( mean ) )
    print ( '  Sample variance = %14g' % ( variance ) )
    print ( '  Sample maximum =  %14g' % ( xmax ) )
    print ( '  Sample minimum =  %14g' % ( xmin ) )

    print ( '' )
    print ( 'LOGISTIC_SAMPLE_TEST' )
    print ( '  Normal end of execution.' )
    return
```

[15]:

```
logistic_cdf_test ( )
logistic_sample_test ( )
```

```
      -1.27491       0.170712       0.218418       -1.27491
       3.08614      0.0417743       0.956318        3.08614
       1.58215       0.141424       0.829509        1.58215
      0.248046       0.246194       0.561695       0.248046
     -0.342069       0.242827       0.415307      -0.342069
       -2.6479       0.061747      0.0661187        -2.6479
       -1.0586       0.191231       0.257578        -1.0586
      -2.09118      0.0978663       0.109957       -2.09118
      -3.08264       0.041908       0.043829       -3.08264
      0.549268       0.232053       0.633966       0.549268

LOGISTIC_SAMPLE_TEST
  LOGISTIC_MEAN computes the Logistic mean
  LOGISTIC_SAMPLE samples the Logistic distribution
  LOGISTIC_VARIANCE computes the Logistic variance.

  PDF parameter A =                          1
```

```
PDF parameter B =                          1
PDF mean =                                 1
PDF variance =                       3.28987
```




```
Sample size =      100000
Sample mean =            0.991571
Sample variance =         3.25204
Sample maximum =          11.6761
Sample minimum =         -10.0133

LOGISTIC_SAMPLE_TEST
  Normal end of execution.
```

(b)Rayleigh Distribution

The Rayleigh Distribution is given by:

$$X\ Rayleigh(\sigma),\ s > 0$$

Let U N(0, 2) and V N(0, 2) be independent random variables, define

$$X = \sqrt{U^2 + V^2}$$

, then X has a Rayleigh distribution with PDF given below.

$$f_X(x) = \begin{cases} \frac{x}{\sigma^2} e^{\frac{x^2}{2\sigma^2}} & x > 0 \\ 0 & otherwise \end{cases}$$

[1]:
```
import numpy as np
from sys import exit
import matplotlib.pyplot as plt
from take_max import take_max
from take_mean import take_mean
from take_min import take_min
from take_variance import take_variance
```

[2]:
```
def rayleigh_cdf ( x, a ):
  if ( x < 0.0 ):
    cdf = 0.0
  else:
    cdf = 1.0 - np.exp ( - x * x / ( 2.0 * a * a ) )
  return cdf
```

[3]:
```
def rayleigh_cdf_inv ( cdf, a ):
  import numpy as np

  if ( cdf < 0.0 or 1.0 < cdf ):
    print ( '' )
    print ( 'RAYLEIGH_CDF_INV - Fatal error!' )
    print ( '  CDF < 0 or 1 < CDF.' )
    exit ( 'RAYLEIGH_CDF_INV - Fatal error!' )

  x = np.sqrt ( - 2.0 * a * a * np.log ( 1.0 - cdf ) )

  return x
```

[14]:
```
def rayleigh_cdf_test ( ):
  print ( '' )
  print ( 'RAYLEIGH_CDF_TEST' )
  print ( '  RAYLEIGH_CDF evaluates the Rayleigh CDF' )
  print ( '  RAYLEIGH_CDF_INV inverts the Rayleigh CDF.' )
  print ( '  RAYLEIGH_PDF evaluates the Rayleigh PDF' )

  a = 2.0


  print ( '' )
  print ( '  PDF parameter A =             %14g' % ( a ) )

  seed = 123456789

  print ( '' )
  print ( '      X            PDF           CDF            CDF_INV' )
  print ( '' )

  for i in range ( 0, 10 ):
    x, seed = rayleigh_sample ( a, seed )
```

```
      pdf = rayleigh_pdf ( x, a )
      cdf = rayleigh_cdf ( x, a )
      x2 = rayleigh_cdf_inv ( cdf, a )
      print ( ' %14g  %14g  %14g  %14g' % ( x, pdf, cdf, x2 ) )

    print ( '' )
    print ( 'RAYLEIGH_CDF_TEST' )
    print ( '  Normal end of execution.' )
    return
```

[5]:
```
  def uniform ( seed ):
    i4_huge = 2147483647
    seed = int ( seed )
    seed = ( seed % i4_huge )
    if ( seed < 0 ):
      seed = seed + i4_huge
    k = ( seed // 127773 )
    seed = 16807 * ( seed - k * 127773 ) - k * 2836
    if ( seed < 0 ):
      seed = seed + i4_huge
    r = seed * 4.656612875E-10
    return r, seed
```

[6]:
```
  def rayleigh_mean ( a ):
    import numpy as np
    mean = a * np.sqrt ( 0.5 * np.pi )
    return mean
```

[28]:
```
  def rayleigh_pdf ( x, a ):
    pdf = ( x / ( a * a ) ) * np.exp ( - x * x / ( 2.0 * a * a ) )
    return pdf
```

[8]:
```
  def rayleigh_sample ( a, seed ):
    cdf, seed = uniform( seed )
    x = rayleigh_cdf_inv ( cdf, a )
    return x, seed
```

[9]:
```
  def rayleigh_variance ( a ):
    variance = 2.0 * a * a * ( 1.0 - 0.25 * np.pi )
    return variance
```

[31]:
```
  def rayleigh_sample_test ( ):
    import numpy as np
    import platform
    from take_max import take_max
    from take_mean import take_mean
    from take_min import take_min
    from take_variance import take_variance
```

```
nsample = 100000
seed = 123456789

print ( '' )
print ( 'RAYLEIGH_SAMPLE_TEST' )
print ( '  Python version: %s' % ( platform.python_version ( ) ) )
print ( '  RAYLEIGH_MEAN computes the Rayleigh mean' )
print ( '  RAYLEIGH_SAMPLE samples the Rayleigh distribution' )
print ( '  RAYLEIGH_VARIANCE computes the Rayleigh variance.' )

a = 1.0
randvars=[]

mean = rayleigh_mean ( a )
variance = rayleigh_variance ( a )

print ( '' )
print ( '  PDF parameter A =             %14g' % ( a ) )
print ( '  PDF mean =                    %14g' % ( mean ) )
print ( '  PDF variance =                %14g' % ( variance ) )


x = np.zeros ( nsample )
for i in range ( 0, nsample ):
  x[i], seed = rayleigh_sample ( a, seed )
  randvars.append(x[i])

mean = take_mean ( nsample, x )
variance = take_variance ( nsample, x )
xmax = take_max ( nsample, x )
xmin = take_min ( nsample, x )
x_samples = np.arange(0,10,0.01)
plt.plot(x_samples,rayleigh_pdf ( x_samples, a ))
plt.hist(randvars, bins=100, normed=True,
              weights=None, cumulative=False, bottom=None,
              histtype=u'bar', align=u'left', orientation=u'vertical',
              rwidth=0.8, log=False, color="red", label=None, stacked=False)

plt.xlabel('X',fontsize=14)
plt.ylabel('PDF',fontsize=14)
plt.show()

print ( '' )
print ( '  Sample size =     %6d' % ( nsample ) )
print ( '  Sample mean =     %14g' % ( mean ) )
print ( '  Sample variance = %14g' % ( variance ) )
print ( '  Sample maximum =  %14g' % ( xmax ) )
print ( '  Sample minimum =  %14g' % ( xmin ) )
```

Problem 1 continued on next page...

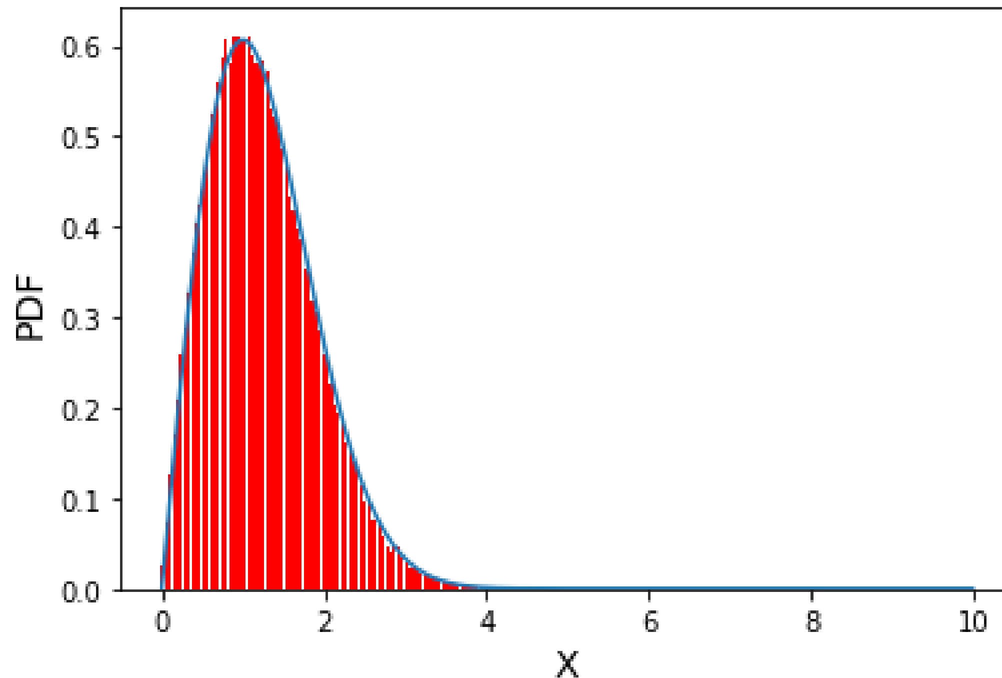
```
    print ( '' )
    print ( 'RAYLEIGH_SAMPLE_TEST' )
    print ( '  Normal end of execution.' )
    return
```

[32]:

```
rayleigh_cdf_test ( )
rayleigh_sample_test ( )
```

```
RAYLEIGH_CDF_TEST
  RAYLEIGH_CDF evaluates the Rayleigh CDF
  RAYLEIGH_CDF_INV inverts the Rayleigh CDF.
  RAYLEIGH_PDF evaluates the Rayleigh PDF

  PDF parameter A =                          2

       X             PDF            CDF            CDF_INV

         1.4041        0.274354       0.218418         1.4041
        5.00465       0.0546538       0.956318        5.00465
        3.76199        0.160346       0.829509        3.76199
         2.5688        0.281479       0.561695         2.5688
        2.07204        0.302877       0.415307        2.07204
       0.739762        0.172712      0.0661187       0.739762
         1.5436        0.286501       0.257578         1.5436
        0.96534        0.214799       0.109957        0.96534
       0.598789        0.143136       0.043829       0.598789
        2.83553        0.259475       0.633966        2.83553

RAYLEIGH_CDF_TEST
  Normal end of execution.

RAYLEIGH_SAMPLE_TEST
  Python version: 3.6.6
  RAYLEIGH_MEAN computes the Rayleigh mean
  RAYLEIGH_SAMPLE samples the Rayleigh distribution
  RAYLEIGH_VARIANCE computes the Rayleigh variance.

  PDF parameter A =                          1
  PDF mean =                           1.25331
  PDF variance =                      0.429204
```

Problem 1 continued on next page...




```
  Sample size =     100000
  Sample mean =            1.25005
  Sample variance =       0.424092
  Sample maximum =         4.62086
  Sample minimum =      0.00574114

RAYLEIGH_SAMPLE_TEST
  Normal end of execution.
```

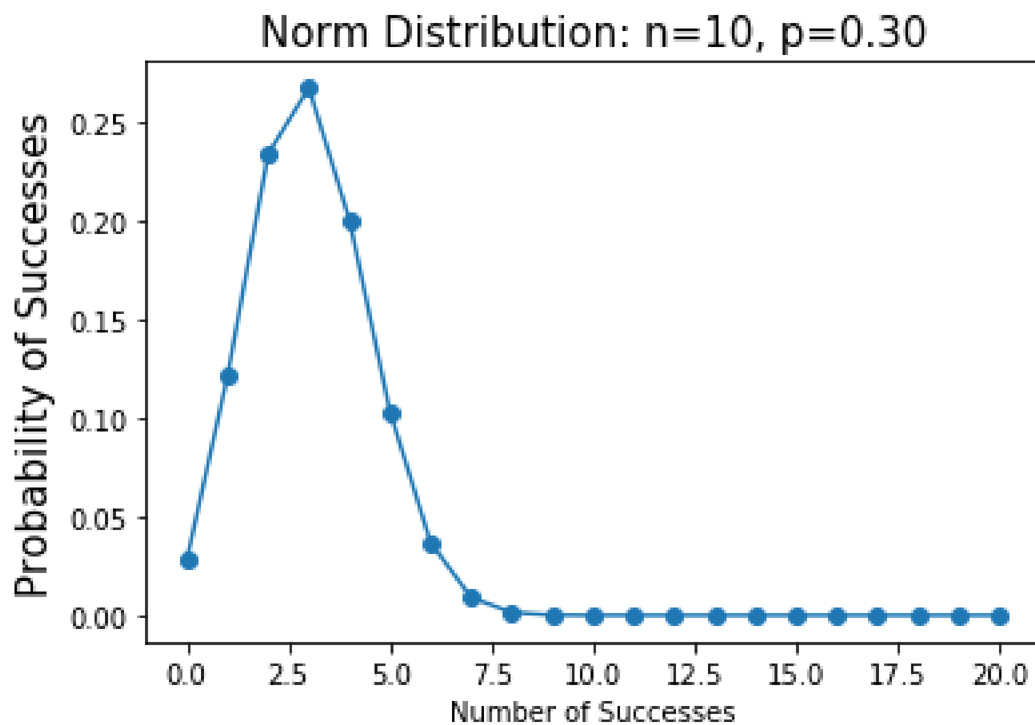
(c) The Box-Muller the Acceptance-Rejection method

[70]:
```
import math
import pandas
import numpy as np
import scipy.stats
from scipy.stats import uniform
from scipy.stats import norm
from scipy.stats import beta
from scipy.stats.mstats import mquantiles
import math
import matplotlib
import matplotlib.pyplot as plt
import pandas as pd
```

```
import seaborn as sns
```

[37]:
```
n = 10
p = 0.3
k = np.arange(0,21)
normal = norm.pdf(k,n,p)
print(normal)
plt.plot(k,binomial,'o-')
plt.title('Norm Distribution: n=%i, p=%.2f' % (n,p), fontsize = 15)
plt.xlabel('Number of Successes')
plt.ylabel('Probability of Successes', fontsize=15)
plt.show()
```

```
[7.06442213e-242 4.91215378e-196 5.10473290e-155 7.92829586e-119
 1.84031612e-087 6.38426299e-061 3.31005236e-039 2.56486621e-022
 2.97030006e-010 5.14092999e-003 1.32980760e+000 5.14092999e-003
 2.97030006e-010 2.56486621e-022 3.31005236e-039 6.38426299e-061
 1.84031612e-087 7.92829586e-119 5.10473290e-155 4.91215378e-196
 7.06442213e-242]
```



The Box-Muller transform

The Box-Muller transform is a method for generating normally distributed random numbers from uniformly distributed random numbers. The Box-Muller transformation can be summarized as follows, sup-

pose u_1 and u_2 are independent random variables that are uniformly distributed between 0 and 1 and let then z_1 and z_2 are independent random variables with a standard normal distribution.

Intuitively, the transformation maps each circle of points around the origin to another circle of points around the origin where larger outer circles are mapped to closely- spaced inner circles and inner circles to outer circles.

[52]:
```
# transformation function
def box_muller(u1,u2):
  z1 = np.sqrt(-2*np.log(u1))*np.cos(2*np.pi*u2)
  z2 = np.sqrt(-2*np.log(u1))*np.sin(2*np.pi*u2)
  return z1,z2
```

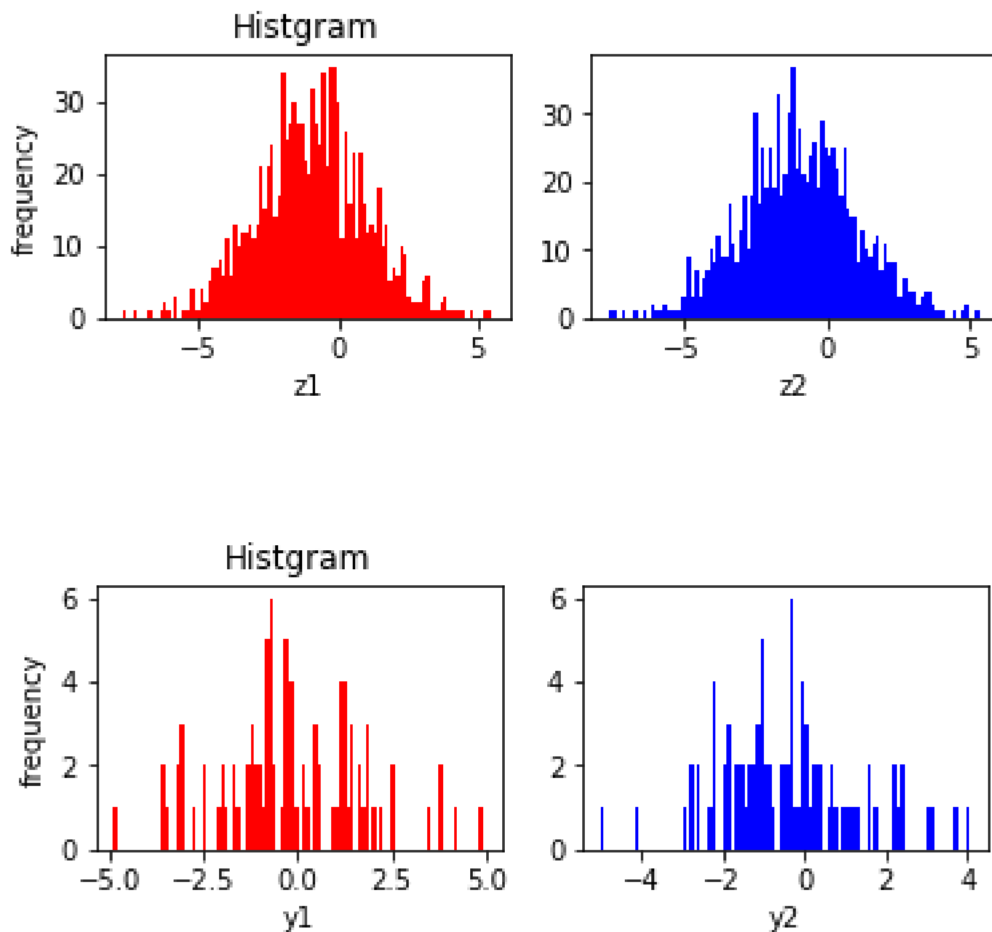
Example in Normal Distribution

If U_1 and U_2 are independent $U(0,1)$ random variables, then

$$z_1 = p\sqrt{2ln(U_1)}cos(2\pi U_2)z_2 = p\sqrt{2ln(U_1)}cos(2\pi U_2)$$

[51]:
```
import numpy as np
import matplotlib.pyplot as plt

#generate from uniform dist
np.random.seed()
N = 1000
z1 = np.random.normal(0, 1.0 ,N)
z2 = np.random.normal(0, 1.0 ,N)
z1 = 2*z1 - 1
z2 = 2*z2 - 1

fig = plt.figure()
ax = fig.add_subplot(2,2,1)
ax.hist(z1,bins=100,color='red')
plt.title("Histgram")
plt.xlabel("z1")
plt.ylabel("frequency")
ax2 = fig.add_subplot(2,2,2)
ax2.hist(z2,bins=100,color='blue')
plt.xlabel("z2")


#discard if z1**2 + z2**2 <= 1
c = z1**2 + z2**2
index = np.where(c<=1)
z1 = z1[index]
z2 = z2[index]
r = c[index]

#transformation
```

```
    y1 = z1*((-2*np.log(r**2))/r**2)**(0.5)
    y2 = z2*((-2*np.log(r**2))/r**2)**(0.5)

    #discard outlier
    y1 = y1[y1 <= 5]
    y1 = y1[y1 >= -5]
    y2 = y2[y2 <= 5]
    y2 = y2[y2 >= -5]

    #plot
    fig = plt.figure()
    ax = fig.add_subplot(2,2,3)
    ax.hist(y1,bins=100,color='red')
    plt.title("Histgram")
    plt.xlabel("y1")
    plt.ylabel("frequency")
    ax2 = fig.add_subplot(2,2,4)
    ax2.hist(y2,bins=100,color='blue')
    plt.xlabel("y2")
    plt.show()
```



Problem 1 continued on next page. . .

Acceptance-Rejection method

To find an explicit formula for F 1(r) for the CDF or to generate $F(x) = P(X \leq (x))$ is not always possible. Even if we can, that may not be the most efficient method for generating a distributed according to $F(x)$. Let us assume the continuous case and that X has CDF and PDF.

The basic idea is to find an alternative probability distribution G, with density g(x), from which we can easily simulate (e.g., inverse-transform etc.). However, we'll also want g(x) 'close' to f(x).

So we assume that $\frac{f(x)}{g(x)}$ is bounded by a constant $c \geq 0$. Hence $sup_x\{\frac{f(x)}{g(x)}\} \leq c$. In practice we want c close to 1 as possible. So Accept-Reject Algorithm can be shown as below:

1. Generate a Y according to G.
2. Generate $R \sim U(0,1)$ independent of Y .
3. If $U \leq \frac{f(Y)}{cg(Y)}$, then set X = Y and accept; otherwise start again and reject.

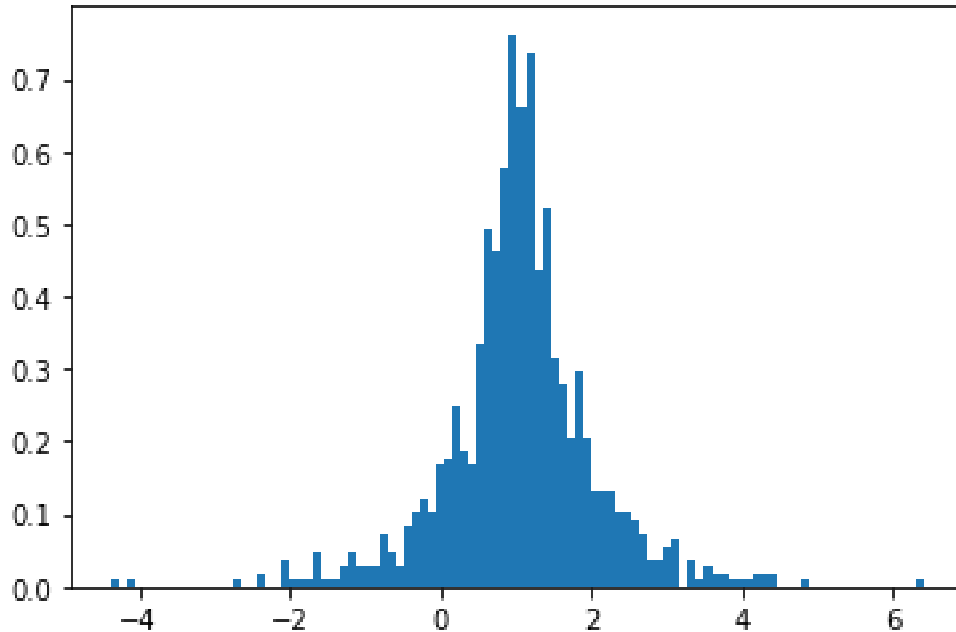
[29]:
```
import numpy as np
from scipy.stats import binom
```

[64]:
```
def rnormal(n, mu, sigma):
 U = np.random.uniform(0,1,n)
 sign = binom.rvs(1, 0.5, size = n)
 sign[sign>0.5] = 1
 sign[sign<0.5] = -1
 y = mu + sign*sigma/np.sqrt(2)*np.log(1-U)
 plt.hist(y,bins=100,normed=1)
```

Example in Normal Distribution

1. Generate Y with an exponential distribution at rate 1; that is, generate U and set $Y = log(U)$
2. Generate U
3. If $U \geq \frac{(Y1)^2}{2}$, set $|Z| = Y$; otherwise go back to 1.
4. Generate U. Set $Z = |Z|$ if $U \leq 0.5$, set $Z = |Z|$ if $U > 0.5$.

[65]:
```
rnormal(1000,1,1)
```



The Difference between The Box-Muller transform and Acceptance-Rejection method

Very often we need to sample from the standard normal distribution. We have seen that we can use the acceptance-rejection method to that end or even the inversion method if we either approximate the normal density or use a numerical method for finding the inverse of the distribution function. But the Cons of it is too time-consuming to find out the result.

The Box–Muller method is a method designed to produce samples from standard normal distribution efficiently. It is based on the following observation. For generating samples from the normal density the Box–Muller algorithm is generally sufficiently efficient.

(d) Beta distribution

Applying the Acceptance-Rejection Method.

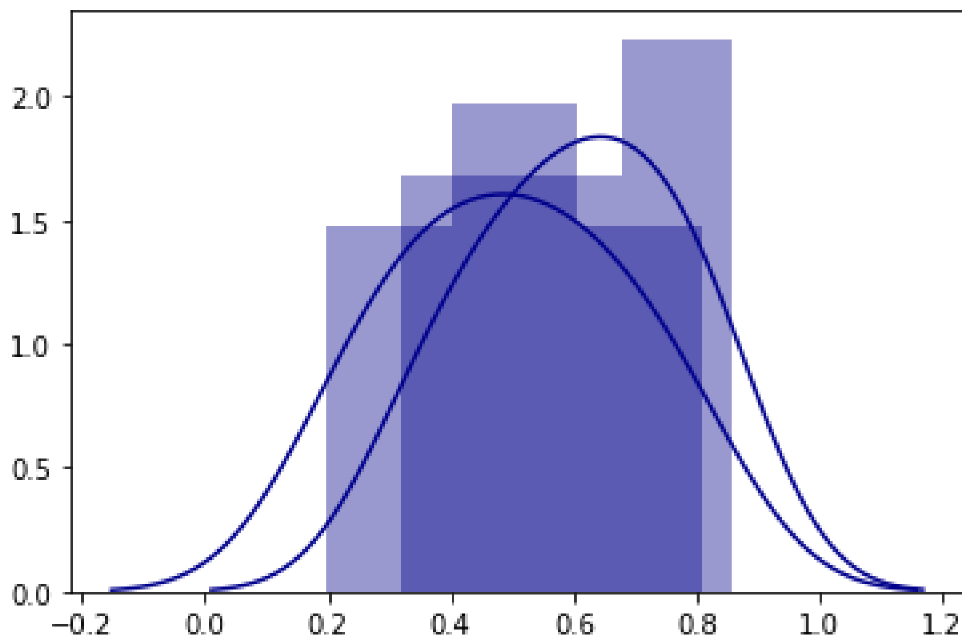
Let us consider a special case of this: $f(x) = bx^n(1x)^n = b(x(1x))^n$. Like $Unif \sim (0,1)$, this has mean $\frac{1}{2}$, but its mass is more concentrated near $\frac{1}{2}$ than near 0 or 1; it has a smaller variance than the uniform. If we use $g(x) = 1, x \in (0,1)$, then $\frac{f(x)}{g(x)} = f(x)$ and as is easily verified, $c = sup_x f(x)$ is achieved at $x = \frac{1}{2}$; $c = b(\frac{1}{4})^n$.

Thus we can generate from $f(x) = b(x(1x))^n$ as follows: 1. Generate U_1 and U_2. 2. If $U_2 4^n (U_1(1U_1))\ldots\ldots n$, then set $X = U_1$; otherwise go back to 1. 3. Get the parameter c in $\frac{f(x)}{cg(x)}$, which is $\frac{(^{2n+1}_{n})}{4_n}$

[81]:
```
        n = 1000
        k = 0
        j = 0
        y = np.zeros(n)
```

```
    while k<n-1:
      u = np.random.uniform(size = 1)
      j = j+1
      x = np.random.uniform(size = 1)
      if x*(1-x) > u:
        k= k+1
        y[k] = x
    p = np.linspace(start = 0.1, stop = 0.9, num=10)
    q_hat = scipy.stats.mstats.mquantiles(y, p)
    ##q_hat = np.quantile(y,p)
    r = beta.ppf(p, 3, 2)
    z = np.sqrt(p*(1-p)/(n*beta.pdf(r, 3, 2)**2))
    print(q_hat)
    print(r)
    print(z)
    temp = np.array([q_hat, r])
    for i in temp:
      sns.distplot(i, hist = True, kde = True, color = 'darkblue')
    plt.show()
```

```
[0.19884839 0.28497432 0.34817953 0.40648894 0.46704451 0.52887576
 0.58302601 0.646068   0.72369341 0.8068061 ]
[0.32046058 0.40829619 0.47627164 0.53494954 0.58857456 0.63953933
 0.68961098 0.7405407  0.7947675  0.85744068]
[0.0113286  0.010457   0.00993545 0.00954212 0.00920814 0.00890172
 0.00860312 0.00829537 0.00795675 0.00754286]
```



Problem 2

Given the following current equation

$$N \sim (0, 1)$$

We would like to evaluate the tail probability $P(I > 8)$ using standard sample average and Importance Sampling.

[1]:
```
import numpy as np
import seaborn as sns
from tqdm import trange
from scipy.stats import norm
import matplotlib.pyplot as plt

# plotting params
%matplotlib inline
plt.rcParams['font.size'] = 10
plt.rcParams['axes.labelsize'] = 10
plt.rcParams['axes.titlesize'] = 10
plt.rcParams['xtick.labelsize'] = 8
plt.rcParams['ytick.labelsize'] = 8
plt.rcParams['legend.fontsize'] = 10
plt.rcParams['figure.titlesize'] = 12
plt.rcParams['figure.figsize'] = (15.0, 8.0)
sns.set_style("white")

# path params
plot_dir = './plots/'
```

(a) Standard Sample Average Estimation

In MC estimation, we approximate an integral by the sample mean of a function of simulated random variables. In more mathematical terms,

$$p(x)\, f(x)\, dx = \mathbb{E}_{p(x)}\left[\, f(x)\right] \approx \frac{1}{N}\, {}_{n=1}^{N} f(x_n)$$

where $x_n \sim \ p(x)$.

A useful application of MC is Standard estimation. In fact, we can cast a probability as an expectation using the indicator function. In our case, given that $A = \{I \mid I > 8\}$, we define $f(x)$ as

$$f(x) = I_A(x) = \begin{cases} 1 & I \geq 8 \\ 0 & I < 8 \end{cases}$$

Replacing in our equation above, we get

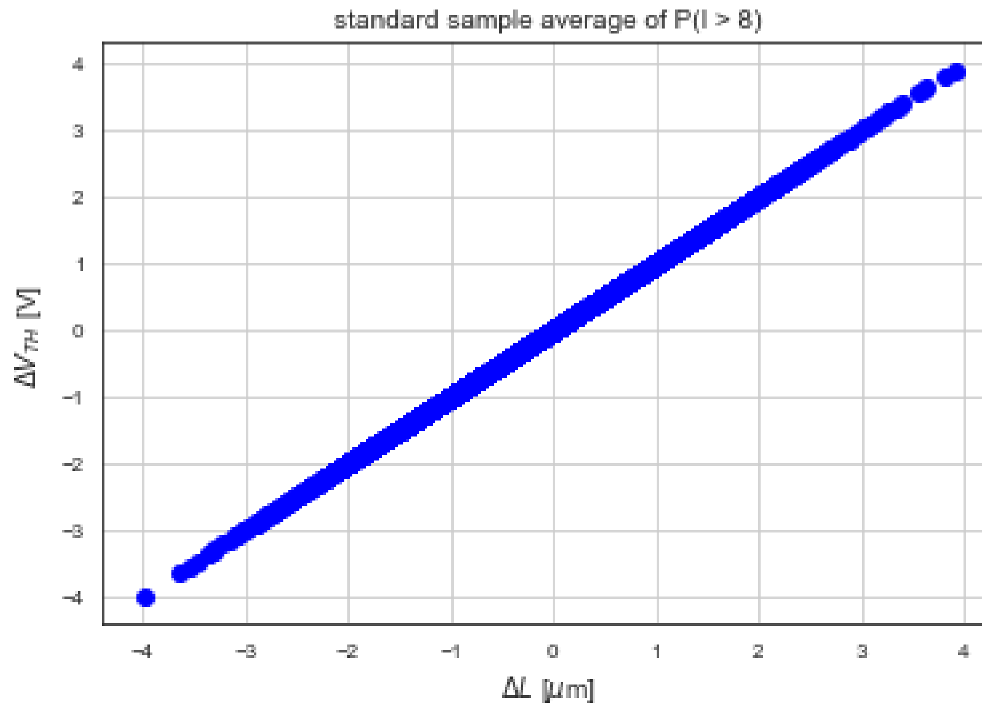
$$p(x)\, f(x)\, dx = \ I(x)\, p(x)\, d(x) = {}_{x \in A}\, p(x)\, d(x) \approx \frac{1}{N}\, {}_{n=1}^{N} I_A(x_n)$$

Problem 2 continued on next page. . .

[14]:
```
def monte_carlo_proba(num_simulations, num_samples, verbose=True, plot=False):

    if verbose:
        print("===========================================")
        print("{} Monte Carlo Simulations of size {}".format(num_simulations,␣
↪num_samples))
        print("===========================================\n")

    num_samples = int(num_samples)
    num_simulations = int(num_simulations)

    probas = []
    for i in range(num_simulations):
        mu_1, sigma_1 = 0, 1

        I = np.random.normal(mu_1, sigma_1, num_samples)

        true_condition = np.where(I >= 8)
        false_condition = np.where(I < 8)
        num_true = true_condition[0].shape[0]
        proba = num_true / num_samples
        probas.append(proba)
        if plot:
            if i == (num_simulations - 1):
                plt.scatter(I[true_condition], I[true_condition], color='r')
                plt.scatter(I[false_condition], I[false_condition], color='b')
                plt.xlabel(r'$\Delta L$ [$\mu$m]')
                plt.ylabel(r'$\Delta V_{TH}$ [V]')
                plt.title("standard sample average of P(I > 8)")
                plt.grid(True)
                plt.savefig(plot_dir + 'monte_carlo_{}.pdf'.format(num_samples),␣
↪format='pdf', dpi=300)
                plt.show()


    mean_proba = np.mean(probas)
    std_proba = np.std(probas)

    if verbose:
        print("Probability Mean: {:0.5f}".format(mean_proba))
        print("Probability Std: {:0.5f}".format(std_proba))

    return probas
```

[16]:
```
probas = monte_carlo_proba(10, 10000, plot=True)
```

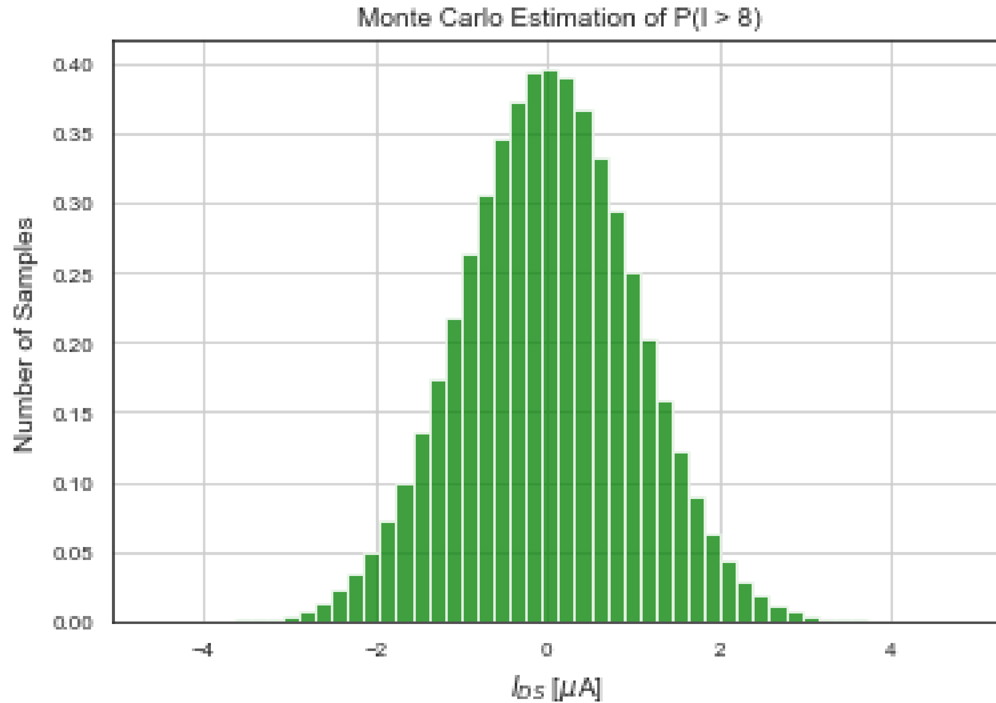
```
===========================================
10 Monte Carlo Simulations of size 10000
===========================================
```




```
Probability Mean: 0.00000
Probability Std: 0.00000
```

[17]:
```
def MC_histogram(num_samples, plot=True):

    num_samples = int(num_samples)

    mu_1, sigma_1 = 0, 1

    I = np.random.normal(mu_1, sigma_1, num_samples)


    if plot:
        n, bins, patches = plt.hist(I, 50, normed=1, facecolor='green', alpha=0.75)
        plt.ylabel('Number of Samples')
        plt.xlabel(r'$I_{DS}$ [$\mu$A]')
        plt.title("Monte Carlo Estimation of P(I > 275)")
        plt.grid(True)
        plt.savefig(plot_dir + 'mc_histogram_{}.pdf'.format(num_samples),␣
 ↪format='pdf', dpi=300)
        plt.show()
```

[18]:
```
MC_histogram(1e6)
```

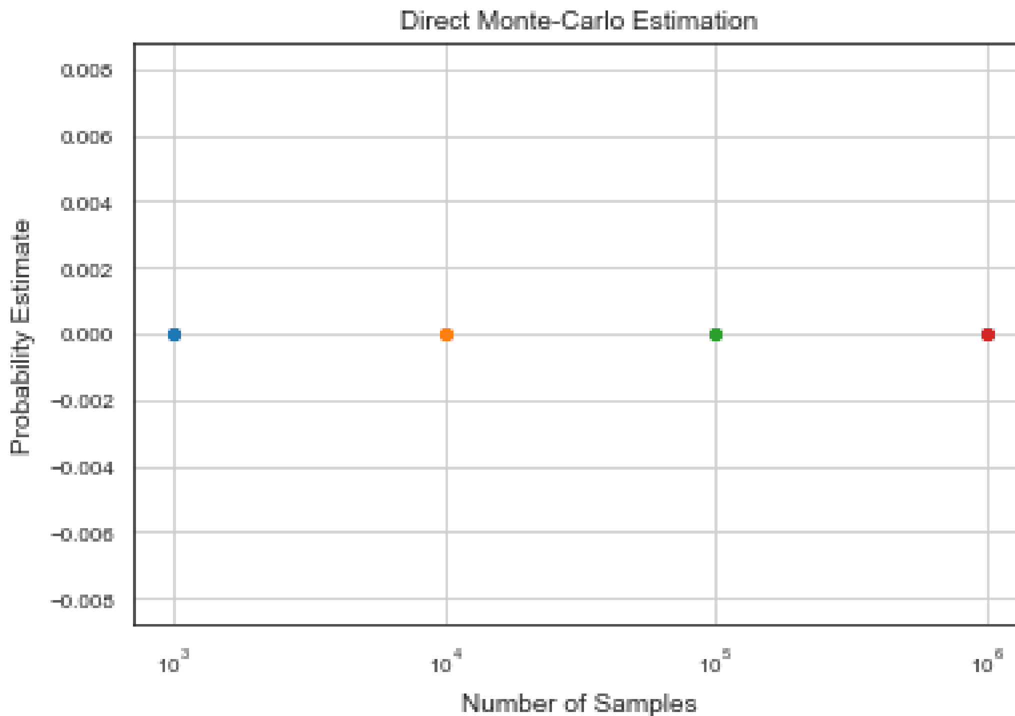


[19]:
```
num_samples = [1e3, 1e4, 1e5, 1e6]
num_repetitions = 25

total_probas = []
for i, num_sample in enumerate(num_samples):
    print("Iter {}/{}".format(i+1, len(num_samples)))
    probas = monte_carlo_proba(num_repetitions, num_sample, verbose=False)
    total_probas.append(probas)
```

```
Iter 1/4
Iter 2/4
Iter 3/4
Iter 4/4
```

[20]:
```
y_axis = np.asarray(total_probas)
x_axis = np.asarray(num_samples)

for x, y in zip(x_axis, y_axis):
    plt.scatter([x] * len(y), y, s=12)

plt.xscale('log')
plt.title("Direct Monte-Carlo Estimation")
plt.ylabel("Probability Estimate")
plt.xlabel('Number of Samples')
plt.grid(True)
```

```
plt.show()
```



(b) Importance Sampling

With importance sampling, we try to reduce the variance of our Monte-Carlo integral estimation by choosing a better distribution from which to simulate our random variables. It involves multiplying the integrand by 1 (usually dressed up in a "tricky fashion") to yield an expectation of a quantity that varies less than the original integrand over the region of integration. Concretely,

$$\mathbb{E}_{p(x)}[f(x)] = f(x)\, p(x)\, dx = f(x)\, p(x)\, \frac{q(x)}{q(x)}\, dx = \frac{p(x)}{q(x)} \cdot f(x)\, q(x)\, dx = \mathbb{E}_{q(x)}[f(x) \cdot \frac{p(x)}{q(x)}]$$

Thus, the MC estimation of the expectation becomes:

$$\mathbb{E}_{q(x)}[f(x) \cdot \frac{p(x)}{q(x)}] \approx \frac{1}{N} \,{}_{n=1}^{N} w_n \cdot f(x_n)$$

where $w_n = \dfrac{p(x_n)}{q(x_n)}$

In our current example above, we can alter the mean and/or standard deviation of ΔL and ΔV_{TH} in the hopes that more of our sampling points will fall in the failure region (red area). For example, let us define 2 new distributions with altered σ^2.

- $\Delta \hat{N} \sim N(0, 1^2)$

[24]:
```
def importance_sampling(num_simulations, num_samples, verbose=True, plot=False):

    if verbose:
        print("===================================================")
        print("{} Importance Sampling Simulations of size {}".format(num_simulations,␣
↪num_samples))
        print("===================================================\n")

    num_simulations = int(num_simulations)
    num_samples = int(num_samples)

    probas = []
    for i in range(num_simulations):
        mu_1, sigma_1 = 0, 1
        mu_1_n, sigma_1_n = 0, 2

        # setup pdfs
        old_pdf_1 = norm(mu_1, sigma_1)
        new_pdf_1 = norm(mu_1_n, sigma_1_n)

        I = np.random.normal(mu_1_n, sigma_1_n, num_samples)

        # calculate f
        true_condition = np.where(I >= 8)

        # calculate weight
        num = old_pdf_1.pdf(I)
        denum = new_pdf_1.pdf(I)
        weights = num / denum

        # select weights for nonzero f
        weights = weights[true_condition]

        # compute unbiased proba
        proba = np.sum(weights) / num_samples
        probas.append(proba)

        false_condition = np.where(I < 8)

    mean_proba = np.mean(probas)
    std_proba = np.std(probas)

    if verbose:
        print("Probability Mean: {}".format(mean_proba))
        print("Probability Std: {}".format(std_proba))

    return probas
```

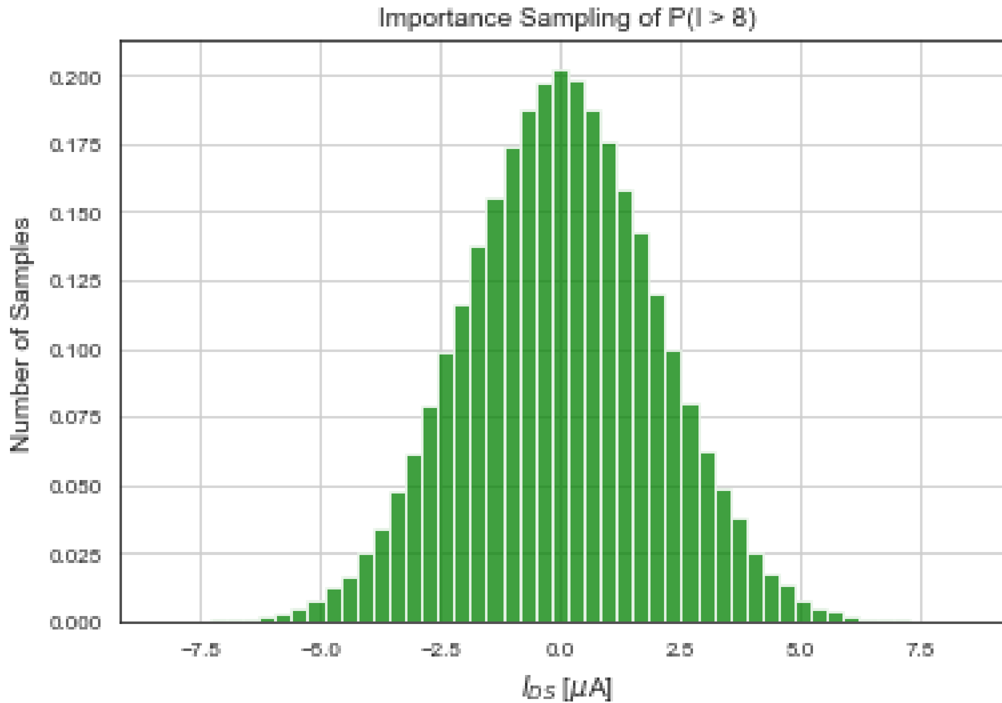
Problem 2 continued on next page. . .

[25]:
```
probas = importance_sampling(10, 10000, plot=True)
```

```
====================================================
10 Importance Sampling Simulations of size 10000
====================================================

Probability Mean: 1.822090729222658e-15
Probability Std: 3.7775832252893025e-15
```

[26]:
```
def IS_histogram(num_samples, plot=True):

    num_samples = int(num_samples)

    mu_1_n, sigma_1_n = 0, 2

    I = np.random.normal(mu_1_n, sigma_1_n, num_samples)

    if plot:
        n, bins, patches = plt.hist(I, 50, normed=1, facecolor='green', alpha=0.75)
        plt.ylabel('Number of Samples')
        plt.xlabel(r'$I_{DS}$ [$\mu$A]')
        plt.title("Importance Sampling of P(I > 8)")
        plt.grid(True)
        plt.savefig(plot_dir + 'is_histogram_{}.pdf'.format(num_samples),␣
 ↪format='pdf', dpi=300)
        plt.show()
```

[27]:
```
IS_histogram(1e5)
```



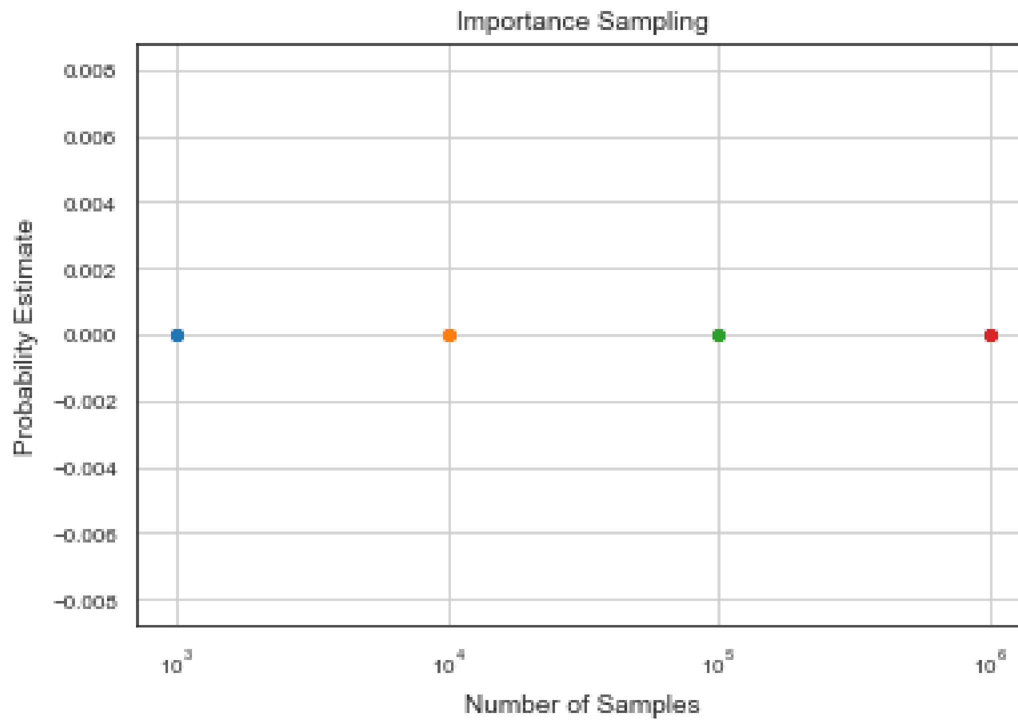
[28]:
```
num_samples = [1e3, 1e4, 1e5, 1e6]
num_repetitions = 25

total_probas = []
for i, num_sample in enumerate(num_samples):
    print("Iter {}/{}".format(i+1, len(num_samples)))
    probas = importance_sampling(num_repetitions, num_sample, verbose=False)
    total_probas.append(probas)
```

```
Iter 1/4
Iter 2/4
Iter 3/4
Iter 4/4
```

[29]:
```
y_axis = np.asarray(total_probas)
x_axis = np.asarray(num_samples)

for x, y in zip(x_axis, y_axis):
    plt.scatter([x] * len(y), y, s=12)

plt.xscale('log')
plt.title("Importance Sampling")
plt.ylabel("Probability Estimate")
plt.xlabel('Number of Samples')
plt.grid(True)
```

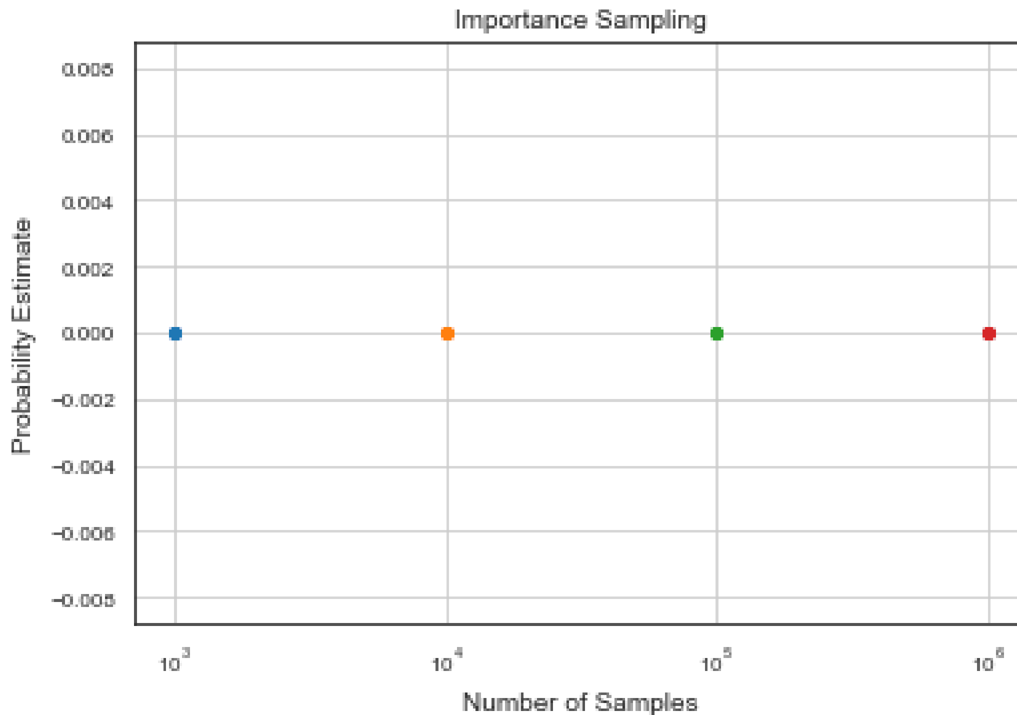
Problem 2 continued on next page. . .

```
plt.savefig(plot_dir + 'imp_sampling_convergence_speed.pdf', format='pdf', dpi=300)
plt.show()
```



Problem 2 continued on next page. . .

Correctness



The probability for $P(X > 8)$ is nearly 0 under the precision of 4 digits after 0.

Problem 3

Proof (a) To simplify the problem first. Denote p as $\frac{1}{2}$. Solving it using Markov chians. Let the state 0 be the start state, then with probability 0.5 you will get heads and move to state 1; with probability 0.5 you will get tails and stay in state 0. Once in state 1, you will either get heads (probability 0.5) and remain in state 1 or you will get tails (probability 0.5) and move to state 2 which is the accepting state. As a traslation matrix, we have:

	0	1	2
0	0.5	0.5	0
1	0.5	0	0.5
2	0	0	1

If we let $\phi(i)$ be the expected number of flips to go from state i to state 2, then from this transition matrix we must have:

$$\phi(0) = 0.5 * (1 + \phi(0)) + 0.5 * (1 + \phi(1)) + 0 * (1 + \phi(2))$$
$$\phi(1) = 0.5 * (1 + \phi(0)) + 0 * (1 + \phi(1)) + 0.5 * (1 + \phi(2))$$
$$\phi(2) = 0 * (1 + \phi(0)) + 0 * (1 + \phi(1)) + 1 * \phi(2)$$

So, obviously, $\phi(2) = 0$ and plugging this into the first two equations gives the same system to solve as in the other answers. Solving for $\phi(2) = 0$ gives $\phi(0) = 4$.

If the probability turns into p, similarly we have:

	0	1	2
0	p	$1-p$	0
1	p	0	$1-p$
2	0	0	1

$$\phi(0) = p*(1+\phi(0)) + (1-p)*(1+\phi(1)) + 0*(1+\phi(2))$$
$$\phi(1) = p*(1+\phi(0)) + 0*(1+\phi(1)) + (1-p)*(1+\phi(2))$$
$$\phi(2) = 0*(1+\phi(0)) + 0*(1+\phi(1)) + 1*\phi(2)$$

Solve for $\phi(2)$ we get $E(N) = \frac{1+p}{p^2} = \frac{1}{p} + \frac{1}{p^2}$

(b) Now, we use N as the random variable until the first HH occurs. We have the following expectations:

$$E(N) = E(E(N|p)) = E(\frac{1}{p}) + E(\frac{1}{p^2})$$

Assume p is an unknown variable that is described as Beta distribution whose α and β are greater than 2. We have:

$$E(\frac{1}{p^2}) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} {}_0^1 p^{\alpha-3}(1-p)^{\beta=1}dp = \frac{\Gamma(\alpha+\beta)\Gamma(\alpha-2)\Gamma(\beta)}{\Gamma(\alpha)\Gamma(\beta)\Gamma(\alpha)\Gamma(\alpha+\beta-2)} = \frac{(\alpha+\beta-1)(\alpha+\beta-2)}{(\alpha-1)(\alpha-2)}$$

So the answer is:

$$E(N) = \frac{(\alpha+\beta-1)(\alpha+\beta-2)}{(\alpha-1)(\alpha-2)} + \frac{\alpha+\beta-1}{\alpha-1}$$

Problem 4

Proof we define the pivot as

$$h(X_1, ...X_n, \mathbf{p}) = \frac{\bar{X} - \mathbf{p}}{\frac{\delta}{\sqrt{N}}} h(X_1, ...X_n, \mathbf{p}) \sim N(0,1)$$

Therefore:

$$P(z(\frac{\epsilon}{2}) \leq \frac{\bar{X} - \mathbf{p}}{\frac{\delta}{\sqrt{N}}} \leq (1 - \frac{\epsilon}{2})) = 1 - \epsilon$$

the $(z(\epsilon)$ is defined as ${}_{-\infty}^{z(\epsilon)} \phi(x)dx = \epsilon$, and $\phi(x)$ is the density function of the standard normal distribution because normal distribution is symmetric about 0. So there's an equation that $z(\frac{\epsilon}{2}) = -z(1-\frac{\epsilon}{2})$ and $-z(1-\frac{\epsilon}{2}) \leq \frac{\bar{X}-\mathbf{p}}{\frac{\delta}{\sqrt{N}}} \leq z(1-\frac{\epsilon}{2}) \equiv \bar{X} - z(1-\frac{\epsilon}{2})\frac{\delta}{\sqrt{N}} \leq \mathbf{p} \leq \bar{X} + z(1-\frac{\epsilon}{2})\frac{\delta}{\sqrt{N}}$. So:

$$P\{\bar{X} - z(1-\frac{\epsilon}{2})\frac{\delta}{\sqrt{N}} \leq \mathbf{p} \leq \bar{X} + z(1-\frac{\epsilon}{1})\frac{\delta}{\sqrt{N}}\} = 1 - \epsilon$$

Therefore, the $1 = \epsilon$ confidence interval for $\mathbf{p}$ is

$$[\bar{X} - z(1-\frac{\epsilon}{2})\frac{\delta}{\sqrt{N}}, \bar{X} + z(1-\frac{\epsilon}{1})\frac{\delta}{\sqrt{N}}] = 1 - \epsilon$$

And the role of δ: population standard deviation; his involves forming a simple random sample from the population. The role of N: the size of the simple random sample; Nice samples, which exhibit no strong skewness or have any outliers, along with a large enough sample size, allow us to invoke the central limit theorem. As a result, we are justified in using a table of z-scores, even for populations that are not normally distributed.

Problem 5

Proof For a known prior PDF f_Θ which is $Unif(0,1)$, we can get the conditional PMF of the data (n_1 is the number of ones, n_0 is the number of zeros) given of the Bernoulli: $p_{X|\Theta}(x|\theta) = \theta^{n_1}(1-\theta)^{n_0}$

$$
\begin{aligned}
f_{\Theta|X}(\theta|x) &= \frac{f_\Theta(\theta)p_{X|\Theta}(x|\theta)}{p_X(x)} \\
&= \frac{f_\Theta(\theta)p_{X|\Theta}(x|\theta)}{{}_u\, f_\theta(u)p_{X|\theta}(x|u)du} \\
&= \frac{\theta^{n_1}(1-\theta)^{n_0}}{{}_u\, u_{n_1}(1-u)^{n_0}du} \\
&= \frac{\theta^{n_1}(1-\theta)^{n_0}}{\beta(n_1+1, n_0+1)} \\
&= \frac{\theta^{n_1}(1-\theta)^{n_0}}{\beta(n_1+1, n_0+1)}
\end{aligned}
$$

The point estimates for the parameter Θ:

$$
\begin{aligned}
E(\Theta) &= {}_0^1\, \theta f_{\Theta|X}(\theta|x)d\theta \\
&= \frac{{}_0^1\, \theta^{n_1+1}(1-\theta)^{n_0}d\theta}{\beta(n_1+1, n_0+1)} \\
&= \frac{\beta(2+x, 1+n-x)}{\beta(1+x, 1+n-x)}
\end{aligned}
$$

Similarly, $E(X)$ has the $X \sim Beta(n, 0.5)$

$$
\begin{aligned}
P(X=k) &= C_n^k \frac{1}{2}^n \quad (k = 0, ...n) \\
E(X) &= \frac{1}{2}
\end{aligned}
$$

Then we can calculate the corresponding function.

$$
\begin{aligned}
E(X^2) &= E[E[\Theta^2|X]] = E[2X^2] = {}_0^1\, 2x^2dx = \frac{2}{3} \\
VarX &= E(X^2) - E(X)^2 = \frac{5}{12} \\
E(\Theta|X) &= E[XE[X|\Theta]] = E[\Theta * \Theta] = {}_0^1 (\frac{\beta(2+x, 1+n-x)}{\beta(1+x, 1+n-x)})^2dx \\
Cov(\Theta, X) &= E[\Theta|X] - E[\Theta]E[X] = {}_0^1 (\frac{\beta(2+x, 1+n-x)}{\beta(1+x, 1+n-x)})^2dx - \frac{\beta(2+x, 1+n-x)}{2\beta(1+x, 1+n-x)} \\
L(\Theta|X) &= (X - \frac{1}{2})\frac{{}_0^1 (\frac{\beta(2+x,1+n-x)}{\beta(1+x,1+n-x)})^2dx - \frac{\beta(2+x,1+n-x)}{2\beta(1+x,1+n-x)}}{\frac{5}{12}} + \frac{\beta(2+x, 1+n-x)}{\beta(1+x, 1+n-x)}
\end{aligned}
$$

Problem 6

Proof (a) First, for a random variable p with a prior beta distribution, we have $p(\mathbf{p}) = \frac{1}{B(\alpha,\beta)}\mathbf{p}^{\alpha-1}(1-\mathbf{p})^{\beta-1}$ for $\mathbf{p} \in [0,1]$ and $B(\alpha,\beta) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha\beta)}$ where Γ is the Gamma function. The mode of the beta distribution is $\frac{\alpha-1}{\alpha+\beta-2}$ According to the MAP Estimation, we have:

$$\begin{aligned}
\hat{\mathbf{p}}_{MAP} &= \arg\max_{\mathbf{p}} Pr(\mathbf{p}|X) \\
&= \arg\max_{\mathbf{p}} \frac{Pr(X|\mathbf{p})Pr(\mathbf{p})}{PrX} \\
&= \arg\max_{\mathbf{p}} Pr(X|\mathbf{p})Pr(\mathbf{p}) \\
&= \arg\max_{\mathbf{p}} {}_{x_i \in X} Pr(x_i|\mathbf{p})Pr(\mathbf{p})
\end{aligned}$$

We can caluculate the argmax for the logarithm then.

$$\begin{aligned}
\arg\max_{\mathbf{p}} Pr(\mathbf{p}|X) &= \arg\max_{\mathbf{p}} logPr(\mathbf{p}|X) \\
&= \arg\max_{\mathbf{p}} log {}_{x_i \in X} Pr(x_i|\mathbf{p})Pr(\mathbf{p}) \\
&= \arg\max_{\mathbf{p}} \Sigma_{x_i \in X}\{logPr(x_i|\mathbf{p})\} + logPr(\mathbf{p})
\end{aligned}$$

Because from beyes' therom, we have: $Pr(x_i|\mathbf{p}) = Bernoulli(x_i|\mathbf{p}) = \mathbf{p}^{x_i}(1-\mathbf{p})^{1-x_i}$ and $Pr(\mathbf{p}) = Beta(\mathbf{p}|\alpha,\beta) = \frac{1}{B(\alpha,\beta)}\mathbf{p}^{\alpha-1}(1-\mathbf{p})^{\beta-1}$, we can deduct that:

$$\begin{aligned}
\mathscr{L} &= logPr(\mathbf{p}|X) \\
&= log\{{}_i Bernoulli(x_i|\mathbf{p})\}Beta(\mathbf{p}|\alpha,\beta) \\
&= \Sigma_i logBernoulli(x_i|\mathbf{p}) + logBeta(\mathbf{p}|\alpha,\beta)
\end{aligned}$$

We can get $\hat{\mathbf{p}}_{MAP} = \arg\max_{\mathbf{p}} \mathscr{L} = \arg\max_{\mathbf{p}} \Sigma_i logBernoulli(x_i|\mathbf{p}) + logBeta(\mathbf{p}|\alpha,\beta)$ and $\frac{\partial}{\partial \mathbf{p}}\mathscr{L} = 0$

$$\begin{aligned}
\frac{\partial}{\partial \mathbf{p}}\mathscr{L} &= \frac{\partial}{\partial \mathbf{p}}\Sigma_i logBernoulli(x_i|\mathbf{p}) + \frac{\partial}{\partial \mathbf{p}} logBeta(\mathbf{p}|\alpha,\beta) \\
&= \frac{1}{\mathbf{p}}\Sigma_{i=1}^n x_i - \frac{1}{1-\mathbf{p}}\Sigma_{i=1}^n(1-x_i) + \frac{\partial}{\partial \mathbf{p}} log\{\frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha\beta)}\mathbf{p}^{\alpha-1}(1-\mathbf{p})^{\beta-1}\} \\
&= \frac{1}{\mathbf{p}}\Sigma_{i=1}^n x_i - \frac{1}{1-\mathbf{p}}\Sigma_{i=1}^n(1-x_i) + \frac{\partial}{\partial \mathbf{p}} log\frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha\beta)} + \frac{\partial}{\partial \mathbf{p}}\mathbf{p}^{\alpha-1}(1-\mathbf{p})^{\beta-1} \\
&= \frac{1}{\mathbf{p}}\Sigma_{i=1}^n x_i - \frac{1}{1-\mathbf{p}}\Sigma_{i=1}^n(1-x_i) + 0 + \frac{\partial}{\partial \mathbf{p}}(\alpha-1)\frac{\partial}{\partial \mathbf{p}} log\mathbf{p} + (beta-1)\frac{\partial}{\partial \mathbf{p}}(1-\mathbf{p}) \\
&= \frac{1}{\mathbf{p}}\Sigma_{i=1}^n x_i - \frac{1}{1-\mathbf{p}}\Sigma_{i=1}^n(1-x_i) + \frac{\alpha-1}{\mathbf{p}} - \frac{\beta-1}{1-\mathbf{p}} \\
&= 0
\end{aligned}$$

Finally, we get:

$$\mathbf{p}[\Sigma_{i=1}^{n}(1-x_i)+\beta-1] = (1-\mathbf{p})[\Sigma_i+\alpha-1]$$
$$\mathbf{p}[\Sigma_{i=1}^{n}(1-x_i)+\Sigma_{i=1}^{n}x_i+\beta-1+\alpha-1] = \Sigma_i x_i+\alpha-1$$
$$\mathbf{p}[\Sigma_{i=1}^{n}+\alpha+\beta-2] = \Sigma_i x_i+\alpha-1$$
$$\mathbf{p}[n+\alpha+\beta-2] = \Sigma_i x_i+\alpha-1$$
$$\hat{\mathbf{p}}_{MAP} = \frac{\Sigma_i x_i+\alpha-1}{n+\beta+\alpha-2}$$

(b) We can say that the constant p is unknown constant but from a uniform distribution on the interval $(0,\mathbf{p})$. First, we can get the pdf of each observation:

$$f(x|\mathbf{p}) = \begin{cases} \frac{1}{\mathbf{p}}\ 0 \le x \le \mathbf{p} \\ 0 \quad otherwise \end{cases}$$

So, we can easily get the likelihood func.

$$f(x|\mathbf{p}) = \begin{cases} \frac{1}{\mathbf{p}^n}\ 0 \le x_i \le \mathbf{p}(i=1,...,n) \\ 0 \quad otherwise \end{cases}$$

It's clear that for all $\mathbf{p} \ge x_i$, MLE of $\mathbf{p}$ can be at most $\frac{1}{\mathbf{p}^n}$. Because MLE is a decreasing function, we can use $max(X_1,...Xn)$ to estimate $\mathbf{p}$.

$$\mathbf{p} = max(x_1,...,x_n)$$
$$\hat{\mathbf{p}}_{MLE} = max(X_1,...,X_n) = \frac{k}{n}$$

(c) By definition, the Beta function is $B(\alpha,\beta) = {}_0^1 x^{\alpha-1}(1-x)^{\beta-1}\,dx = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha+\beta)}$

$$\mathbf{p} = E[X] = \frac{{}_0^1 x^{\alpha}(1-x)^{\beta-1}\,dx}{B(\alpha,\beta)} = \frac{B(\alpha+1,\beta)}{B(\alpha,\beta)} = \frac{\Gamma(\alpha+1)\Gamma(\beta)}{\Gamma(\alpha+\beta+1)}\frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} = \frac{\alpha}{\alpha+\beta}$$

Applying the identity $\Gamma(t+1) = t\Gamma(t)$. We get for each $\sigma^2 = \frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}$

$$\sigma^2+\mathbf{p}^2 = E[X^2] = \frac{B(\alpha+2,\beta)}{B(\alpha,\beta)} = \frac{\alpha(\alpha+1)}{(\alpha+\beta)(\alpha+\beta+1)}$$

According to the MSE theorm, we have:

$$\mathbf{p} = E(\bar{X}) = E(\frac{X_1+...X_n}{n}) = \frac{E(X_1+...E(X_n))}{n}$$

So $\bar{X}$ is an unbiased estimator.

$$\hat{\mathbf{p}}_{MMSE} = MSE_{\bar{X}} = E(\bar{X}-\mathbf{p})^2 = Var(\bar{X}) = \frac{\sigma^2}{n} = \frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)n}$$

Problem 7

Proof First, we can solve the problem applying the $E(K) = \Sigma_{k=1}^{\infty} Pr(K \geq k)$. Then The problem can be denoted as finding the expectations of average number of people required to find a pair with same birthday. So we can denoted case X as the random variable "The first person to have the same birthday".

We know that the probability of $k-1$ people to have no matches is

$$P(K \geq 1) = 1$$
$$P(K \geq 365+2) = 0$$
$$P(K \geq k+1) = \frac{365!}{(365-k)!(365)^k}$$

$$\frac{365}{365}\frac{365-1}{365}\frac{365-2}{365}...\frac{365-k+2}{365} = \frac{365!}{(366-k)!365^k}$$

Then the probability of a match on the k-th person is

$$P(K = k) = (k-1)\frac{365!}{(366-k)!365^k}$$

If using $k = 1$ and starting sum from $i = 1$ yields:

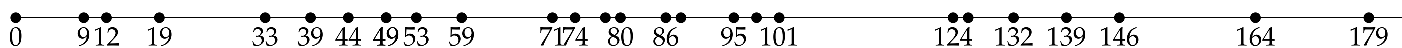
$$E[X] = 1 + \Sigma_{k=1}^{\infty}\frac{365!}{(365-k)!365^k}$$

As for the integral form:

$$E(X) = 1 + _{k=1}^{\infty}\frac{365!}{(365-k)!365^k}$$

Problem 8

Proof



The question equals to picking a number from the given range, is the number in a short or long interfal, which is called size-based sampling. For the bus waiting problem, the expected length of an interarrival time, which contains a fixed time t is larger than the average interval lenth between buses.

Consider a fixed t ≤ 0. The time last train came is $E(S_{N_t+1} - S_{N_t}) = E(S_{N_t+1}) - E(S_{N_t})$

For $E(S_{N_t+1})$, condition on N_t. Consider

$$E(S_{N_t+1}|N_t = k) = E(S_{N_t}|N_t = k) = E(S_{k+1}) = \frac{k+1}{\lambda}$$

The second part is just the $k+1$ version of the above equation that occurs after time t, so it's independent of N_t. So, it's obvious that $E(S_{N_t+1})|N_t = \frac{N_t+1}{\lambda}$. According to the production law:

$$E(S_{N_t+1}) = E(E(S_{N_t+1}|N_t)) = E(\frac{N_t+1}{\lambda}) = \frac{\lambda t+1}{\lambda} = t + \frac{1}{\lambda}$$

Problem 8 continued on next page. . .

Applying the same theory we can easily get the $E(S_{N_t})$

$$
\begin{aligned}
E(S_{N_t}|N_t = k) &= \frac{tk}{k+1} \\
E(S_{N_t}|N_t) = \frac{tN_t}{1+N_t}) &= t - \frac{t}{N_t+1} \\
E(S_{N_t}) = E(E(S_{N_t}|N_t)) = E(t - \frac{t}{N_t+1}) &= t - tE(\frac{1}{N_t+1})
\end{aligned}
$$

So,

$$
\begin{aligned}
E(\frac{1}{Nt+1}) &= \Sigma_{k=0}^{\infty}(\frac{1}{k+1})\frac{e^{-\lambda t}(\lambda t)^k}{k!} \\
&= \frac{e^{-\lambda t}}{\lambda t}\Sigma_{k=0}^{\infty}\frac{(\lambda t)^{k+1}}{(k+1)!} \\
&= \frac{e^{-\lambda t}}{-\lambda t}\lambda t\Sigma_{k=0}^{\infty}\frac{(\lambda t)^k}{k!} \\
&= \frac{e^{-\lambda t}}{\lambda t}(e^{\lambda t} - 1) \\
&= \frac{1 - e^{-\lambda t}}{\lambda t}
\end{aligned}
$$

In conclusion, $E(S_{N_t}) = t - \frac{1}{\lambda} + \frac{e^{-\lambda t}}{\lambda}$, and thus $E(S_{N_t+1} - S_{N_t}) = (t + \frac{1}{\lambda}) - (t - \frac{1}{\lambda} + \frac{e^{\lambda t}}{\lambda}) = \frac{2 - e^{e^{-\lambda t}}}{\lambda}$

Problem 9

Proof

$$
\frac{\partial \pi(x)}{\partial H(a)} = \frac{\partial \frac{e^{H(x)}}{{}^{k}_{y=1} e^{H(y)}}}{\partial H(a)}
$$

$$
\begin{aligned}
\frac{\partial e^{H(x)}}{\partial H(a)} &= \begin{cases} e^{H(x)} & x = a \\ 0 & x \neq a \end{cases} \\
\frac{\partial {}^{k}_{y=1} e^{H(y)}}{\partial H(a)} &= \frac{\partial(e^{H(1)} + e^{H(2)} + \cdots + e^{H(k)})}{\partial H(a)} = e^{H(a)} + (k-1) \times 0 = e^{H(a)}
\end{aligned}
$$

So, it's obvious that:

$$
\frac{\partial \pi(x)}{\partial H(a)} = \frac{\partial \frac{e^{H(x)}}{{}^{k}_{y=1} e^{H(y)}}}{\partial H(a)} = \frac{\frac{\partial e^{H(x)}}{\partial H(a)} \times {}^{k}_{y=1} e^{H(y)} - \frac{\partial {}^{k}_{y=1} e^{H(y)}}{\partial H(a)} \times e^{H(x)}}{({}^{k}_{y=1} e^{H(y)})^2} = \frac{\frac{\partial e^{H(x)}}{\partial H(a)} \times {}^{k}_{y=1} e^{H(y)} - e^{H(a)} \times e^{H(x)}}{({}^{k}_{y=1} e^{H(y)})^2}
$$

If $x = a$:

$$
\frac{\partial \pi(x)}{\partial H(a)} = \frac{e^{H(x)} \times {}^{k}_{y=1} e^{H(y)} - e^{H(a)} \times e^{H(x)}}{({}^{k}_{y=1} e^{H(y)})^2} = \frac{e^{H(x)}}{{}^{k}_{y=1} e^{H(y)}} - \frac{e^{H(a)}}{{}^{k}_{y=1} e^{H(y)}} \times \frac{e^{H(x)}}{{}^{k}_{y=1} e^{H(y)}} = \pi(x) - \pi(a) \times \pi(x) = \pi(x)(1 - \pi(a))
$$

Problem 9 continued on next page...

If $x \neq a$:

$$\frac{\partial \pi(x)}{\partial H(a)} = \frac{-e^{H(a)} \times e^{H(x)}}{({}^{k}_{y=1} e^{H(y)})^2} = -\frac{e^{H(a)}}{{}^{k}_{y=1} e^{H(y)}} \times \frac{e^{H(x)}}{{}^{k}_{y=1} e^{H(y)}} = -\pi(a) \times \pi(x)$$

In conclusion, $\frac{\partial \pi(x)}{\partial H(a)} = \pi(x)(1_{\{x=a\}} - \pi(a))$

Reference

1. http://www.columbia.edu/~ks20/4703-Sigman/4703-07-Notes-ARM.pdf
2. https://github.com/AnmolPanchal/Box-mueller-Method-Acceptance-Rejection-Algorithm-Binomial-Distribution-
3. https://www.win.tue.nl/~marko/2WB05/lecture8.pdf
4. http://ib.berkeley.edu/labs/slatkin/eriq/classes/guest_lect/mc_lecture_notes.pdf