**HOMEWORK 3**

**COMPUTATIONAL METHODS FOR DATA SCIENCE**

**FALL SEMESTER 2022**

D11948002 資料科學博士班一年級 巫哲嘉

1. **Generate Random Variables for Common Distributions.** *(20 points)*
2. Normal

We can use Inverse Method,

Since we know the CDF of normal distribution:

then, we can generate U (0,1) random variable (r.v.) and input the r.v. into inverse function:

.

Therefore, we can generate Normal r.v.

1. Exponential

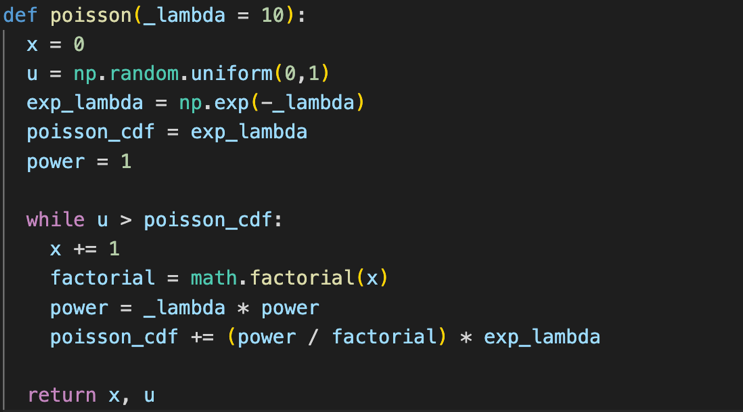
We can also use Inverse Method,

The CDF:

thus, we can generate r.v.:

1. Poisson

Since Poisson is a discrete distribution, we can consider using “Inversion with sequential search”, below is the implementation:



when using sequential search in the inversion method, we need to use the recurrence relation. That is, ,

1. Chi-Square

By the transformation method. By the definition of Chi-Square distribution, it is the summation of k squared Normal random variables. Now, we know how to generate Normal random variables, we could use the idea of transformation to generate Chi-Square random variables.

To generate a random variable of Chi-Square (df=), we can do the following step:

1. We first generate Normal random variables with U (0, 1) like what we do in (a).
2. Square of each random variable.
3. Sum up all of them.

By “Inverse method”,

CDF:

where the PDF

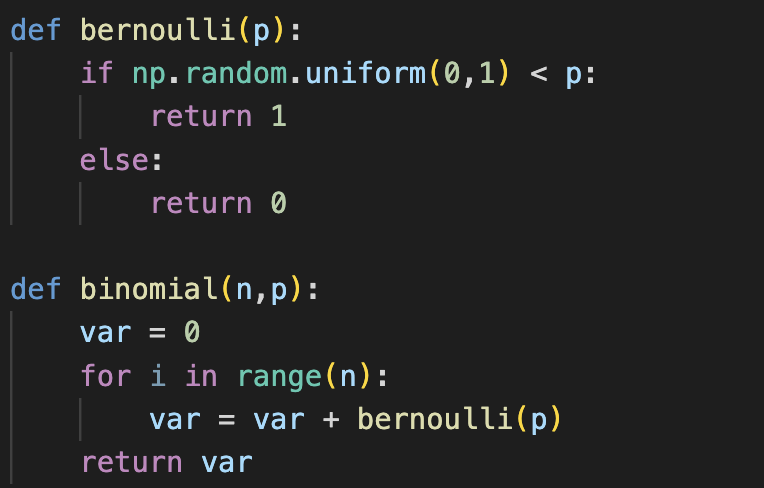
while and is Beta function.

Thus, the r.v. will be:

1. Binomial (*n, p*)

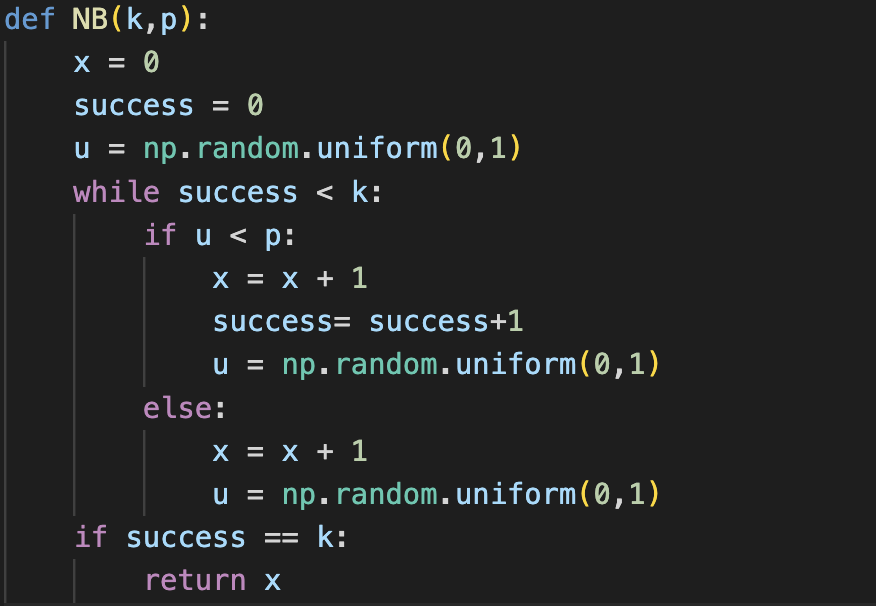
By definition, Binomial r.v. can be calculated by summing up n Bernoulli r.v.

In other words, we need to do Bernoulli trials with probability of success (p) n times and count the number of successful trials.



1. Negative Binomial

Below is the implementation:



x: the total number of trials.

success: the number of success trials.

If success reaches k (the designated k-th success), then the function will return the total number of trials.

1. Dirichlet (

Let Y be a random vector with components following a standard gamma distribution. The Dirichlet distribution is a distribution over vectors that fulfill the conditions .

Draw k independent random samples from gamma distribution.

Then is Dirichlet-distributed.

Now that we know the probability mass function of Dirichlet distribution, we use the CDF of Dirichlet distribution and repeat the variable generating skills like we did in (c) to generate discrete random variables.

1. **Sampling Problem.** *(10 points)*
2. Estimate using importance sampling with standardized weights

We can estimate the expectation by a known and easily sampled distribution.

Now we can sample from distribution as shown above,

where is the sampling weight and is the estimator.

Suppose:

,

and we already know that:

so,

Hence,

1. Repeat the estimation using rejection sampling

Suppose:

target distribution.

We propose to use:

.

We want to sample from using a proposal PMF that we can sample.

First, we need to find the boundary of k such that:

Owing to , the smallest is .

Define:

Then,

Solving the equation: , we get .

which gives .

1. **Estimation** *(20 points)*
2. Implement this method for *h*=1 and *n* = 2000. What is the effect of increasing *n*? What is the effect of increasing and decreasing *h*? Please comment.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| h\n | 20000 | 25000 | 30000 | 40000 |
| 1 | 3.438 | 3.448 | 3.437 | 3.449 |
| 0.5 | 3.468 | 3.472 | 3.447 | 3.455 |
| 0.2 | 3.380 | 3.330 | 3.340 | 3.303 |
| 2 | 3.143 | 3.127 | 3.149 | 3.150 |
| 3 | 3.134 | 3.148 | 3.135 | 3.133 |

When we fix *h,* increasing or decreasing *n* does not affect the estimate very much. However, when we fix *n,* increasing *h* could drive the estimate closer to 3.14, which is closer to the accurate .

|  |
| --- |
| Python |
| def pi\_estimate(n, h):  x = [0]  y = [0]  t = 0  while t <n - 1:  xt = np.random.uniform(-h, h)  yt = np.random.uniform(-h, h)  try\_x = xt + x[t]  try\_y = yt + y[t]  if -1 <= try\_x <= 1 and -1 <= try\_y <= 1:  x.append(try\_x)  y.append(try\_y)  t += 1  #print(len(x))  return x, y  def cal\_pi(x, y, n):  in\_circle = 0  for i in range(len(x)):  if x[i]\*\*2 + y[i]\*\*2 <= 1:  in\_circle += 1  pi = 4 \* in\_circle / n  return pi  h\_list = [1, 0.5, 0.2, 2, 3]  n\_list = [20000, 25000, 30000, 40000]  for h in h\_list:  for n in n\_list:  x, y = pi\_estimate(n, h)  pi = cal\_pi(x, y, n)  print("n = ", n, "h = ", h, "pi = ", pi) |

1. Explain why this method is flawed. Using the same method to generate candidates, develop the correct approach by referring to the Metropolis-Hastings ratio.

By using this method, the probability of getting a sample near the edge is much less than getting a sample around the center, for if we get a sample out of range, we will resample. The idea of leveraging Metropolis-Hastings (MH) ratio is that when we move to the next point (x, y), we set it as the new center and repeat the uniform sampling by . Afterward, we compute the ratio . Since the probability of any sample in uniform distribution is the same, we can take it as a constant. Hence, . Furthermore, . When , . elsewhere. As a result, when the new sample locates within the target area, we will accept the new sample for . On the other hand, when the new sample falls out of range, we will reject the new point for , and stay at the same location. With MH ratio, if we receive a sample around the edge which is highly likely to breach the boundary, the point will tend to stay near the edge and therefore balance the probability between the center area and the edge area. The distribution will behave more similar to the uniform distribution.

1. Implement your approach from Part (b) and calculate .

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| h\n | 20000 | 25000 | 30000 | 40000 |
| 1 | 3.113 | 3.129 | 3.145 | 3.158 |
| 0.5 | 3.150 | 3.132 | 3.136 | 3.147 |
| 0.2 | 3.189 | 3.069 | 3.142 | 3.125 |
| 2 | 3.137 | 3.14 | 3.136 | 3.145 |
| 3 | 3.113 | 3.113 | 3.141 | 3.150 |

In comparison with the table in (a), the calculated results oscillate around 3.1 with smaller variance than the results in (a). However, the trend is similar that when h < 1, the estimated is farther away from 3.14. When h > 1, the estimated is relatively close to 3.14. Moreover, as n increases, however the size of h changes, the estimated remain stable and quite close to 3.14.

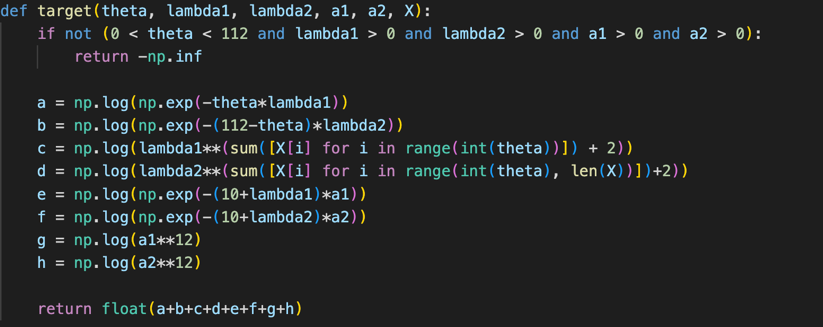
|  |
| --- |
| Python |
| def mh\_pi\_estimate(n, h):  x = [0]  y = [0]  t = 0  while t < n - 1:  xt = np.random.uniform(x[t] - h, x[t] + h)  yt = np.random.uniform(y[t] - h, y[t] + h)  try\_x = xt  try\_y = yt  #print(try\_x)  #print(try\_y)  if -1 <= try\_x <= 1:  x.append(try\_x)  #y.append(try\_y)  else:  x.append(x[t])  #y.append(y[t])  if -1 <= try\_y <= 1:  y.append(try\_y)  else:  y.append(y[t])  t += 1  #print(x[t], y[t])  #print(len(x))  return x, y  h\_list = [1, 0.5, 0.2, 2, 3]  n\_list = [20000, 25000, 30000, 40000]  for h in h\_list:  for n in n\_list:  x, y = mh\_pi\_estimate(n, h)  pi = cal\_pi(x, y, n)  print("n = ", n, "h = ", h, "pi = ", pi) |

1. **Coal-Mining Disaster Analysis** *(35 points)*
2. Estimate the posterior mean and provide a histogram each for and .

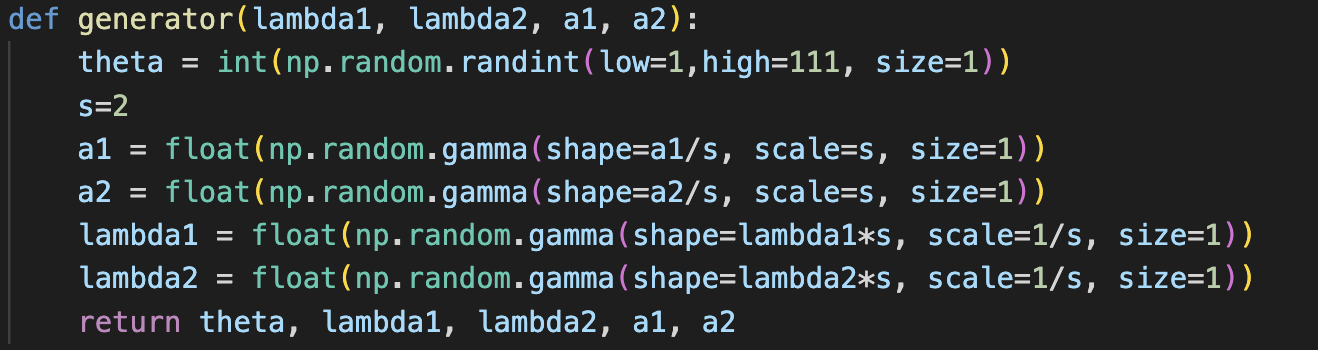
The idea is that we employ Metropolis-Hastings’s algorithm to generate random sample to estimate posterior means and plot histograms for and .

Target Function:

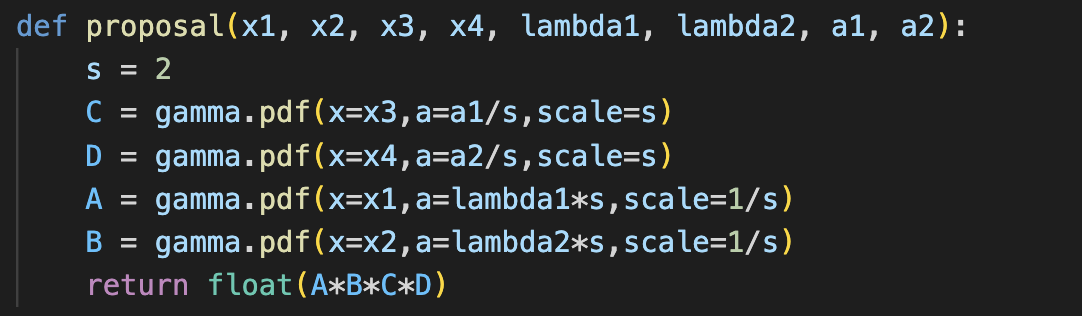
By Bayes’ Theorem, we could obtain the posterior distribution function:



Sample Generator:



Proposal Function:

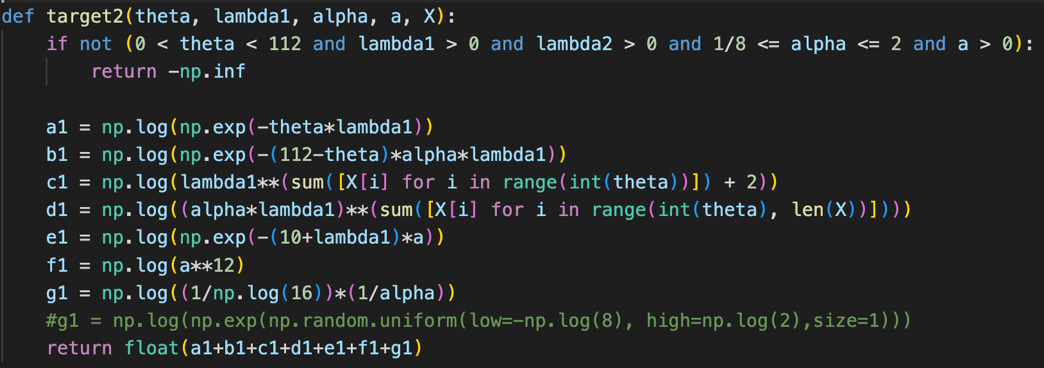


|  |  |  |
| --- | --- | --- |
| Generate and | | |
| We generate with the given information of , , in problem 4.  The , , we generated are summarized as follows:  (, , ) = (85, 2.76, 3.88) | | |
| Python | | |
| #%% 4.a  theta = np.random.randint(low=1,high=111, size=1)  a1 = np.random.gamma(shape=10,scale=0.1,size=1)  a2 = np.random.gamma(shape=10,scale=0.1,size=1)  lambda1 = np.random.gamma(shape=3, scale= 1/a1, size=1)  lambda2 = np.random.gamma(shape=3, scale= 1/a2, size=1)  X1 = np.random.poisson(lam = lambda1, size=theta)  X2 = np.random.poisson(lam = lambda2, size=112-theta)  X = np.concatenate((X1,X2),axis=None)  #%% 4.a  def generator(lambda1, lambda2, a1, a2):  theta = int(np.random.randint(low=1,high=111, size=1))  s=2  a1 = float(np.random.gamma(shape=a1/s, scale=s, size=1))  a2 = float(np.random.gamma(shape=a2/s, scale=s, size=1))  lambda1 = float(np.random.gamma(shape=lambda1\*s, scale=1/s, size=1))  lambda2 = float(np.random.gamma(shape=lambda2\*s, scale=1/s, size=1))  return theta, lambda1, lambda2, a1, a2  def target(theta, lambda1, lambda2, a1, a2, X):  if not (0 < theta < 112 and lambda1 > 0 and lambda2 > 0 and a1 > 0 and a2 > 0):  return -np.inf  a = np.log(np.exp(-theta\*lambda1))  b = np.log(np.exp(-(112-theta)\*lambda2))  c = np.log(lambda1\*\*(sum([X[i] for i in range(int(theta))]) + 2))  d = np.log(lambda2\*\*(sum([X[i] for i in range(int(theta), len(X))])+2))  e = np.log(np.exp(-(10+lambda1)\*a1))  f = np.log(np.exp(-(10+lambda2)\*a2))  g = np.log(a1\*\*12)  h = np.log(a2\*\*12)  return float(a+b+c+d+e+f+g+h)  def proposal(x1, x2, x3, x4, lambda1, lambda2, a1, a2):  s = 2  C = gamma.pdf(x=x3,a=a1/s,scale=s)  D = gamma.pdf(x=x4,a=a2/s,scale=s)  A = gamma.pdf(x=x1,a=lambda1\*s,scale=1/s)  B = gamma.pdf(x=x2,a=lambda2\*s,scale=1/s)  return float(A\*B\*C\*D)  #%%  N = 500000  Y = np.zeros([N,5])  V = np.zeros([N,5])  accept = 1  r=0  theta\_v0, lambda1\_v0, lambda2\_v0, a1\_v0, a2\_v0 = generator(lambda1, lambda2, a1, a2)  V[0] = [theta\_v0, lambda1\_v0, lambda2\_v0, a1\_v0, a2\_v0]  Y[0] = V[0]  for i in range(1,N):  epsilon = 1e-300  theta\_v, lambda1\_v, lambda2\_v, a1\_v, a2\_v = generator(lambda1=Y[i-1,1],lambda2=Y[i-1,2],a1=Y[i-1,3],a2=Y[i-1,4])  V[i] = [theta\_v, lambda1\_v, lambda2\_v, a1\_v, a2\_v]  ratio\_up = np.exp(target(V[i,0], V[i,1], V[i,2], V[i,3], V[i,4], X))\*proposal(x1=Y[i-1,1], x2=Y[i-1,2], x3=Y[i-1,3], x4=Y[i-1,4], lambda1=V[i,1], lambda2=V[i,2],a1=V[i,3],a2=V[i,4])  ratio\_down = np.exp(target(Y[i-1,0], Y[i-1,1], Y[i-1,2], Y[i-1,3], Y[i-1,4], X))\*proposal(x1=V[i,1], x2=V[i,2], x3=V[i,3], x4=V[i,4], lambda1=Y[i-1,1], lambda2=Y[i-1,2],a1=Y[i-1,3],a2=Y[i-1,4])  log\_ratio = np.log(ratio\_up)-np.log(ratio\_down)  rho = min(log\_ratio,1)  U = float(np.random.uniform(low=0,high=1,size=1))  if np.log(U) <= rho:  Y[i] = V[i]  accept = accept+1  else:  Y[i] = Y[i-1]  print('Posterior mean of theta:',Y[:,0].mean())  print('Posterior mean of lambda1:',Y[:,1].mean())  print('Posterior mean of lambda2:',Y[:,2].mean()) | | |
| Estimates of Posterior Means | | |
| After we generate random samples of each parameter, we calculate the sample means of each random sample. The results of each estimate are listed above. | | |
| Histograms of , , | | |
|  |  |  |
| Traceplots of , , | | |
|  |  |  |

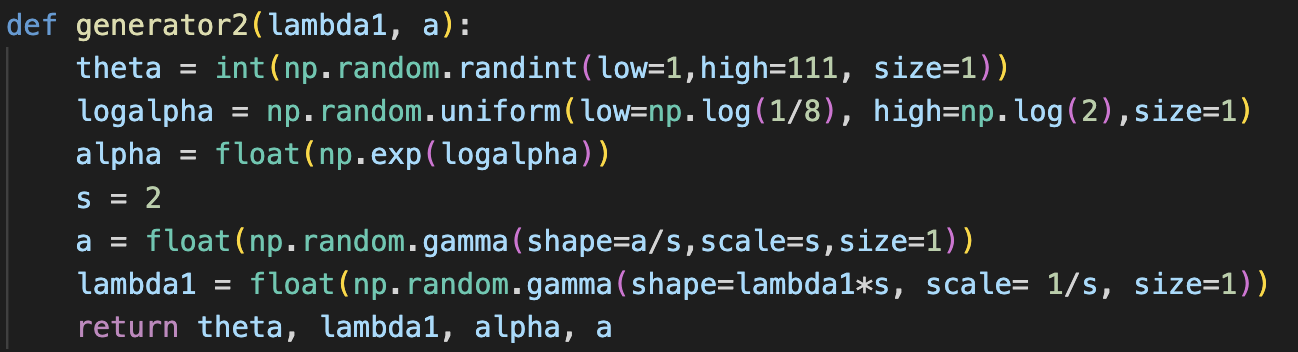
1. Consider a modification that . Assume the same discrete uniform prior for and the same prior , . In addition, assume . Estimate the posterior mean and provide a histogram each for , , and .

Target Function:

By Bayes’ Theorem, we could obtain the posterior distribution function:



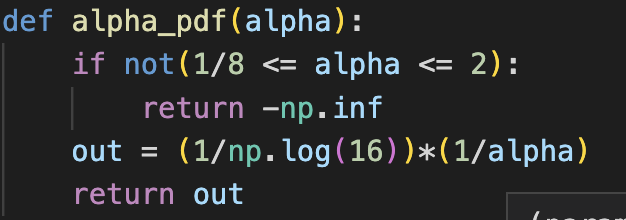
Sample Generator:



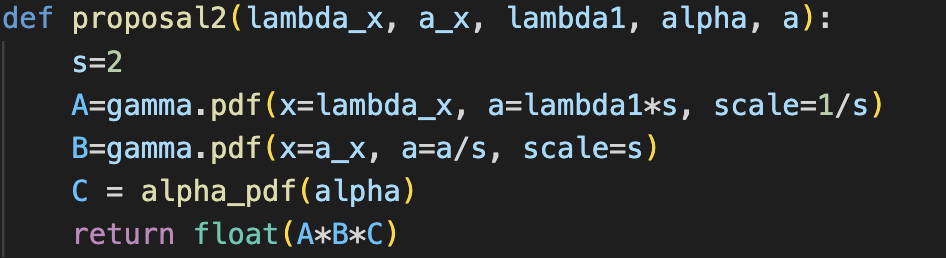
Function:

Since , , we need to derive the probability density function for .

,



Proposal Function:

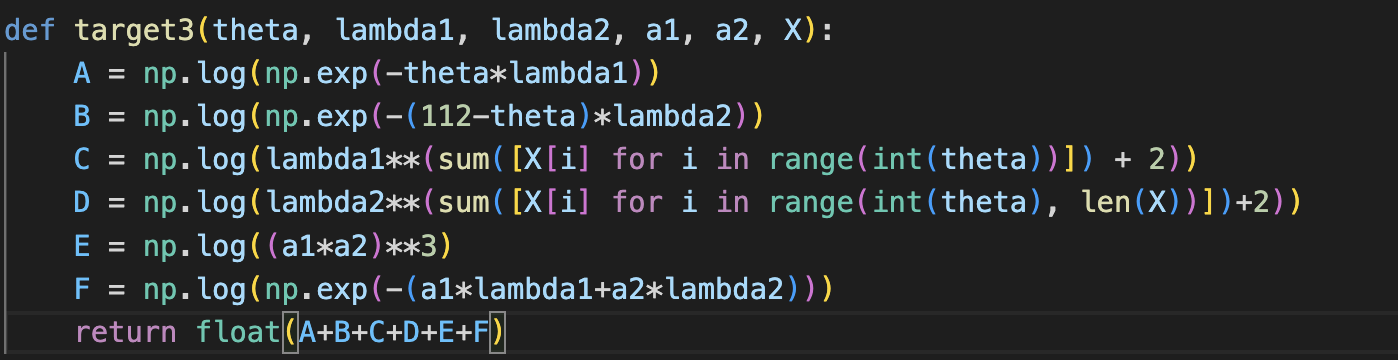


|  |  |  |
| --- | --- | --- |
| Generate , , | | |
| The , , , we generated are summarized as follows:  (, , ) = (64, 2.58, 0.84) | | |
| Python | | |
| logalpha = np.random.uniform(low=np.log(1/8), high=np.log(2),size=1)  alpha = np.exp(logalpha)  theta = np.random.randint(low=1,high=111, size=1)  a = np.random.gamma(shape=10,scale=0.1,size=1)  lambda1 = np.random.gamma(shape=3, scale= 1/a, size=1)  lambda2 = alpha\*lambda1  X1 = np.random.poisson(lam = lambda1, size=theta)  X2 = np.random.poisson(lam = lambda2, size=112-theta)  X = np.concatenate((X1,X2),axis=None)  #%%  def generator2(lambda1, a):  theta = int(np.random.randint(low=1,high=111, size=1))  logalpha = np.random.uniform(low=np.log(1/8), high=np.log(2),size=1)  alpha = float(np.exp(logalpha))  s = 2  a = float(np.random.gamma(shape=a/s,scale=s,size=1))  lambda1 = float(np.random.gamma(shape=lambda1\*s, scale= 1/s, size=1))  return theta, lambda1, alpha, a  def target2(theta, lambda1, alpha, a, X):  if not (0 < theta < 112 and lambda1 > 0 and lambda2 > 0 and 1/8 <= alpha <= 2 and a > 0):  return -np.inf  a1 = np.log(np.exp(-theta\*lambda1))  b1 = np.log(np.exp(-(112-theta)\*alpha\*lambda1))  c1 = np.log(lambda1\*\*(sum([X[i] for i in range(int(theta))]) + 2))  d1 = np.log((alpha\*lambda1)\*\*(sum([X[i] for i in range(int(theta), len(X))])))  e1 = np.log(np.exp(-(10+lambda1)\*a))  f1 = np.log(a\*\*12)  g1 = np.log((1/np.log(16))\*(1/alpha))  #g1 = np.log(np.exp(np.random.uniform(low=-np.log(8), high=np.log(2),size=1)))  return float(a1+b1+c1+d1+e1+f1+g1)  def alpha\_pdf(alpha):  if not(1/8 <= alpha <= 2):  return -np.inf  out = (1/np.log(16))\*(1/alpha)  return out  def proposal2(lambda\_x, a\_x, lambda1, alpha, a):  s=2  A=gamma.pdf(x=lambda\_x, a=lambda1\*s, scale=1/s)  B=gamma.pdf(x=a\_x, a=a/s, scale=s)  C = alpha\_pdf(alpha)  return float(A\*B\*C)  #%%  N = 200000  Y = np.zeros([N,4])  V = np.zeros([N,4])  accept = 1  theta\_v0, lambda1\_v0, alpha\_v0, a\_v0 = generator2(lambda1, a)  V[0] = [theta\_v0, lambda1\_v0, alpha\_v0, a\_v0]  Y[0] = V[0]  for i in range(1,N):  theta\_v, lambda1\_v, alpha\_v, a\_v = generator2(lambda1=Y[i-1,1],a=Y[i-1,3])  V[i] = [theta\_v, lambda1\_v, alpha\_v, a\_v]  ratio\_up = np.exp(target2(V[i,0], V[i,1], V[i,2], V[i,3], X))\*proposal2(lambda\_x=Y[i-1,1], a\_x=Y[i-1,3],lambda1=V[i,1],alpha=V[i,2],a=V[i,3])  ratio\_down = np.exp(target2(Y[i-1,0], Y[i-1,1], Y[i-1,2], Y[i-1,3], X))\*proposal2(lambda\_x=V[i,1], a\_x=V[i,3],lambda1=Y[i-1,1],alpha=Y[i-1,2],a=Y[i-1,3])  log\_ratio = np.log(ratio\_up)-np.log(ratio\_down)  rho = min(log\_ratio,1)  U = float(np.random.uniform(low=0,high=1,size=1))  if np.log(U) <= rho:  Y[i] = V[i]  accept = accept+1  else:  Y[i] = Y[i-1]  print('Posterior mean of theta:',Y[:,0].mean())  print('Posterior mean of lambda1:',Y[:,1].mean())  print('Posterior mean of lambda2:',(Y[:,1]\*Y[:,2]).mean()) | | |
| Estimates of Posterior Means | | |
| After we generate random samples of each parameter, we calculate the sample means of each random sample. The results of each estimate are listed above. | | |
| Histogram of , , | | |
|  |  |  |
| Traceplots of , , | | |
|  |  |  |

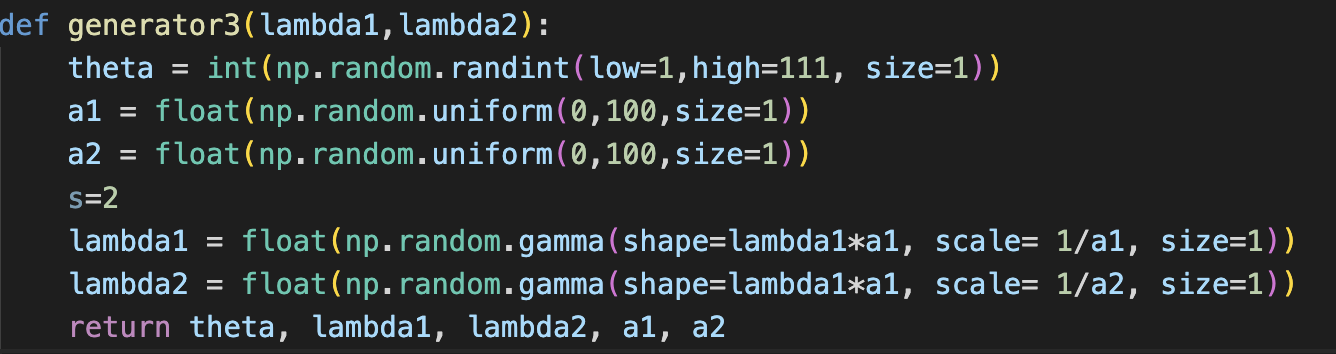
1. Consider a modification that priors , and independently for instead. Assume the same discrete uniform prior for . Estimate the posterior mean and provide a histogram each for , , and .

Target Function:

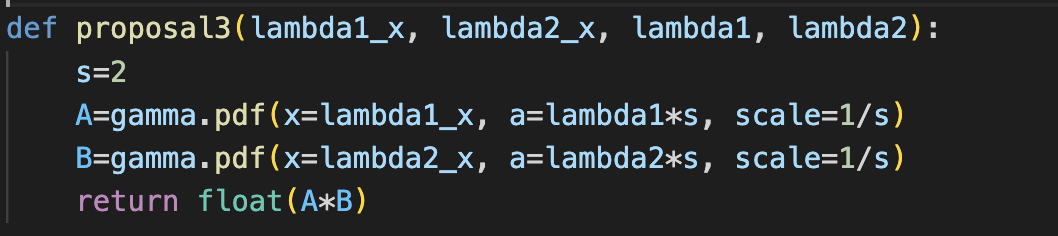
By Bayes’ Theorem, we could obtain the posterior distribution function:



Sample Generator:



Proposal Function:



|  |  |  |
| --- | --- | --- |
| Generate , , | | |
| The , , , we generated are summarized as follows:  (, , ) = (59, 0.292, 0.12) | | |
| Python | | |
| theta = np.random.randint(low=1,high=111, size=1)  a1 = np.random.uniform(0,100,size=1)  a2 = np.random.uniform(0,100,size=1)  lambda1 = np.random.gamma(shape=3, scale= 1/a1, size=1)  lambda2 = np.random.gamma(shape=3, scale= 1/a2, size=1)  X1 = np.random.poisson(lam = lambda1, size=theta)  X2 = np.random.poisson(lam = lambda2, size=112-theta)  X = np.concatenate((X1,X2),axis=None)  #%%  def generator3(lambda1,lambda2):  theta = int(np.random.randint(low=1,high=111, size=1))  a1 = float(np.random.uniform(0,100,size=1))  a2 = float(np.random.uniform(0,100,size=1))  s=2  lambda1 = float(np.random.gamma(shape=lambda1\*a1, scale= 1/a1, size=1))  lambda2 = float(np.random.gamma(shape=lambda1\*a1, scale= 1/a2, size=1))  return theta, lambda1, lambda2, a1, a2  def target3(theta, lambda1, lambda2, a1, a2, X):  A = np.log(np.exp(-theta\*lambda1))  B = np.log(np.exp(-(112-theta)\*lambda2))  C = np.log(lambda1\*\*(sum([X[i] for i in range(int(theta))]) + 2))  D = np.log(lambda2\*\*(sum([X[i] for i in range(int(theta), len(X))])+2))  E = np.log((a1\*a2)\*\*3)  F = np.log(np.exp(-(a1\*lambda1+a2\*lambda2)))  return float(A+B+C+D+E+F)  def proposal3(lambda1\_x, lambda2\_x, lambda1, lambda2):  s=2  A=gamma.pdf(x=lambda1\_x, a=lambda1\*s, scale=1/s)  B=gamma.pdf(x=lambda2\_x, a=lambda2\*s, scale=1/s)  return float(A\*B)  #%%  N = 500000  Y = np.zeros([N,5])  V = np.zeros([N,5])  accept = 1  theta\_v0, lambda1\_v0, lambda2\_v0, a1\_v0, a2\_v0 = generator3(lambda1,lambda2)  V[0] = [theta\_v0, lambda1\_v0, lambda2\_v0, a1\_v0, a2\_v0]  Y[0] = V[0]  for i in range(1,N):  theta\_v, lambda1\_v, lambda2\_v, a1\_v, a2\_v = generator3(lambda1=Y[i-1,1],lambda2=Y[i-1,2])  V[i] = [theta\_v, lambda1\_v, lambda2\_v, a1\_v, a2\_v]  ratio\_up = np.exp(target3(V[i,0], V[i,1], V[i,2], V[i,3], V[i,4], X))\*proposal3(lambda1\_x=Y[i-1,1],lambda2\_x=Y[i-1,2],lambda1=V[i,1],lambda2=V[i,2])  ratio\_down = np.exp(target3(Y[i-1,0], Y[i-1,1], Y[i-1,2], Y[i-1,3], Y[i-1,4], X))\*proposal3(lambda1\_x=V[i,1],lambda2\_x=V[i,2],lambda1=Y[i-1,1],lambda2=Y[i-1,2])  log\_ratio = np.log(ratio\_up)-np.log(ratio\_down)  rho = min(log\_ratio,1)  U = float(np.random.uniform(low=0,high=1,size=1))  if np.log(U) <= rho:  Y[i] = V[i]  accept = accept+1  else:  Y[i] = Y[i-1]  print('Posterior mean of theta:',Y[:,0].mean())  print('Posterior mean of lambda1:',Y[:,1].mean())  print('Posterior mean of lambda2:',Y[:,2].mean()) | | |
| Estimates of Posterior Means | | |
| After we generate random samples of each parameter, we calculate the sample means of each random sample. The results of each estimate are listed above. | | |
| Histogram of , , | | |
|  |  |  |
| Traceplots of , , | | |
|  |  |  |

1. Using the standard setup, derive the conditional distributions necessary to carry out Gibbs sampling for the change-point model.

By Bayes’ Theorem,

We can derive the following full conditional distributions:

1. Implement the Gibbs sampler. Use a suite of convergence diagnostics to evaluate the convergence and mixing of your sampler.

|  |  |  |
| --- | --- | --- |
| Generate , , | | |
| The , , we generated are summarized as follows:  (, , ) = (62, 0.657, 3.918) | | |
| Python | | |
| theta = np.random.randint(low=1,high=111, size=1)  a1 = np.random.gamma(shape=10,scale=0.1,size=1)  a2 = np.random.gamma(shape=10,scale=0.1,size=1)  lambda1 = np.random.gamma(shape=3, scale= 1/a1, size=1)  lambda2 = np.random.gamma(shape=3, scale= 1/a2, size=1)  X1 = np.random.poisson(lam = lambda1, size=theta)  X2 = np.random.poisson(lam = lambda2, size=112-theta)  X = np.concatenate((X1,X2),axis=None)  #%% Posterior Theta Generator  class Theta(object):  def \_\_init\_\_(self, lambda1, lambda2, X):  self.lambda1 = lambda1  self.lambda2 = lambda2  self.X = X  def theta\_up(self, theta):  A = np.exp(theta\*(self.lambda2 - self.lambda1))  B = (self.lambda1/self.lambda2)\*\*(sum([self.X[i] for i in range(int(theta))]))  return float(A\*B)  def theta\_pmf(self, theta):  A = self.theta\_up(theta)  B = sum([self.theta\_up(theta = i) for i in range(1,112)])  out = A/B  return float(out)  def theta\_cdf(self, theta):  out = sum([self.theta\_pmf(theta = i) for i in range(1,theta+1)])  return out  def generator(self):  U = np.random.uniform(low=0, high=1, size=1)  t = 1  while self.theta\_cdf(t) < U:  t = t + 1  return t  #%% Gibbs Sampler  theta\_y0 = int(np.random.randint(low=1,high=111, size=1))  a1\_y0 = float(np.random.gamma(shape=10,scale=0.1,size=1))  a2\_y0 = float(np.random.gamma(shape=10,scale=0.1,size=1))  lambda1\_y0 = float(np.random.gamma(shape=3, scale= 1/a1, size=1))  lambda2\_y0 = float(np.random.gamma(shape=3, scale= 1/a2, size=1))  ThetaGenerator = Theta(lambda1, lambda2, X)  ThetaGenerator.generator()  N = 1000  Y = np.zeros([N, 5])  Y[0] = [theta\_y0, lambda1\_y0, lambda2\_y0, a1\_y0, a2\_y0]  for t in range(1, N):  ThetaGenerator = Theta(lambda1 = Y[t-1,1], lambda2 = Y[t-1,2], X = X)  Y[t,0] = ThetaGenerator.generator()  Y[t,1] = np.random.gamma(shape=(sum([X[i] for i in range(int(Y[t,0]))])+3), scale = 1/(Y[t,0]+Y[t-1,3]), size=1)  Y[t,2] = np.random.gamma(shape=(sum([X[i] for i in range(int(Y[t,0]), len(X))])+3), scale=1/(112-Y[t,0]+Y[t-1,4]), size=1)  Y[t,3] = np.random.gamma(shape = 13, scale = 1/(10+Y[t,1]), size=1)  Y[t,4] = np.random.gamma(shape = 13, scale = 1/(10+Y[t,2]), size=1)  #%% Traceplots in 4.e  plt.plot(range(N), Y[:,0], label='theta')  # plt.plot(range(N), Y[:,1], label='lambda1')  # plt.plot(range(N), Y[:,2], label='lambda2')  plt.ylabel("Estimate")  plt.xlabel("Steps")  plt.legend()  plt.show() | | |
| Traceplots of , , | | |
|  |  |  |

1. Construct density histogram and a table of summary statistics for the approximate posterior distributions of θ, λ1 and λ2

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Table of Summary Statistics | | | | | | |
| Parameter | Sample Mean | | Sample Median | Min | | Max |
|  | 62.32 | | 62 | 58 | | 111 |
|  | 0.73 | | 0.72 | 0.43 | | 3.55 |
|  | 3.67 | | 3.66 | 0.97 | | 4.91 |
| Histogram of , , | | | | | | |
|  | |  | | |  | |

1. **Unbiased Cross-Validation** *(15 points)*
2. Show that can be simplified into , where and .

the Gaussian kernel.

the kernel density estimate, where *h* denotes the width of the kernel, often called kernel bandwidth.

leave-one-out cross-validation

|  |
| --- |
| Note: |

1. Find the functional form of C
2. **Bootstrapping Practice** *(Bonus 10 points)*
3. Show that and , where .

The asterisk (\*) in the formulas indicates that these are estimations of the mean and variance of the bootstrap estimate of the statistic. is the bootstrap sample and the empirical distribution of observed data.

1. By Taylor Series Expansion, find and . You can show the first two terms only of the expansion only.

We generate B bootstrap samples by sampling with replacement from the original sample. Then, is the statistic of interest calculated for the original sample, and is the statistic of interest calculated for the *b*th bootstrap sample. We can approximate the mean and variation of the original sample with the following functions: