EE511 Project5

Name: Yijing Jiang

Email: yijingji@usc.edu

USC ID: 5761477714

Question 1

Explane:

- Simulate a single-server queue system with nonhomogeneous Poisson process. The system can be changed by three actions:
  - Job arrives. Arrival is a poisson distribution, and the arrival rate (lambda) is a function to the current time.
  - Job departures. Job is finished after service-time. Service-time is an exponential distribution with rate 25 jobs/h.
  - Server takes a break. If there is no job in line when the server finished the last job, then the server will have a break. The time of the break is uniformly distributed on (0,0.3)
- By using discrete event simulator, consider following kinds of event:
  - The server is on work:
    - Next arrival is ealier than next departure, which means a job comes when the server still works on the previous one. So the arriving job needs to wait in line.
    - Next arrival is later than next departure, which means before next job comes the server has finished the previous one. Then there will be two cases:
      - If there is no jobs after the previous one(because the next job hasn't come yet), the server will have a break time.
      - If there still have jobs after the previous one, then the server just continues to work on the next one.
  - The server is on break:
    - Next arrival is during the break, which means during the break there will be one more job waiting in line, and once the break is end, the server need to process this job.
    - Next arrival is after the break, which means the next event is the break being finished. When the break is finished, there also will be two possible cases:
      - If there is no job in line, that is, no job arrives during the break time, then the server can start another break time.

- If there is jobs in line, then the server should start work.

Code:

```
1   import numpy as np
2
3   #get param of arrival at time t, return the param lambda
4   def getLambda(t):
5       t = t%10
6       if t >= 0.0 and t < 5.0: return 4+3*t
7       else: return 4+3*(10-t)
8   #get the next arrival time tA. t is current time
9   def getArrive(t):
10      u1 = np.random.uniform(0,1)
11      t = t - np.log(u1)/20
12      while np.random.uniform(0,1) > getLambda(t)/19:
13          u1 = np.random.uniform(0,1)
14          t = t - np.log(u1)/20
15      return t
16  #get the next departure time tD. t is current time
17  def getDeparture(t):
18      u = np.random.uniform(0,1)
19      td = - np.log(1-u)/(25)
20      return t + td
21  #get the next open time tR. t is current time
22  def getBreak(t):
23      tr = np.random.uniform(0,0.3)
24      return t + tr
25
26  #Interested break time
27  B_array = np.zeros((100,)) #total break time
28  for i in range(100): #update B_array[i]
29      #State initialize
30      NA = 0 #arrival number
31      ND = 0 #departure number
32      n = 0 #current number in system(in service and in line)
33      t = 0 #current time. t<=100
34      #Var initialize
35      tA = getArrive(t) #next arrive time
36      tD = 200 #next departure time, set 200 as infinity
37      tR = 200 #next open time, set 200 as infinity
38      while t <= 100:
39          if tR == 200: #open
40              if tA <= tD: #custom arrive
41                  t = tA
42                  NA = NA + 1
43                  n = n + 1
44                  tA = getArrive(t)
45                  if tD == 200: tD = getDeparture(t)
46              else: #departure
47                  t = tD
```

```
                    ND = ND + 1
                    n = n - 1
                    if n > 0: #continue work
                        tD = getDeparture(t)
                    else: #start break
                        tD = 200
                        tR = getBreak(t)
                        B_array[i] = B_array[i] + tR - t
            else: #close
                if tA <= tR: #arrive before break end
                    t = tA
                    NA = NA + 1
                    n = n + 1
                    tA = getArrive(t)
                else: #arrive after break end
                    t = tR
                    if n == 0: #no customer in line
                        tR = getBreak(t)
                        B_array[i] = B_array[i] + tR - t
                    else: #has customer in line, start work
                        tR = 200
                        tD = getDeparture(t)
print(np.mean(B_array))
```

Result:

- Repeat the experiment for 100 times, and record the total break time in each experiment. Get all of the break time as following:

```
[49.81900052 52.94516956 51.43361125 52.55903591 53.18379304 50.52564426
 50.82726128 47.71986117 56.59383522 53.72548006 51.61171189 52.03573425
 52.36976977 55.05536623 51.83945197 52.03472625 49.15312543 53.37216644
 53.77800077 51.12017452 50.95619582 54.08120884 52.2412005  52.73557901
 55.18804143 52.10559602 50.59096904 51.94923542 51.00573183 51.39532503
 53.02926267 52.32742375 50.83149484 48.41927926 51.21041486 50.83857264
 50.33677911 48.54483635 56.16254953 52.82146238 51.01343197 49.29262617
 51.42951399 51.38511861 48.82775167 52.63879933 50.05943638 53.38990951
 52.6138719  50.26892826 49.90129418 48.25391011 53.28596678 50.96707179
 53.61626677 50.25266329 52.43915589 52.59655448 55.27202282 49.84115177
 52.35230202 50.51290054 52.07363269 52.2573753  52.7982822  51.8689957
 48.75583423 53.80405598 51.97504416 50.51160099 51.90590787 53.12311772
 53.17242062 50.39888444 51.84702261 55.62991763 52.61326762 54.63590705
 53.48577143 48.35548906 53.26079436 54.11814793 51.25577952 51.31375296
 48.25134972 50.80349847 52.75614541 48.64476492 54.00684266 46.83154351
 53.09186601 52.04835799 52.15877188 52.77306599 51.97693056 51.82404671
```

- And compute the expected amount of time that the server is on break in the first 100 hours:

```
51.516811612958556
```

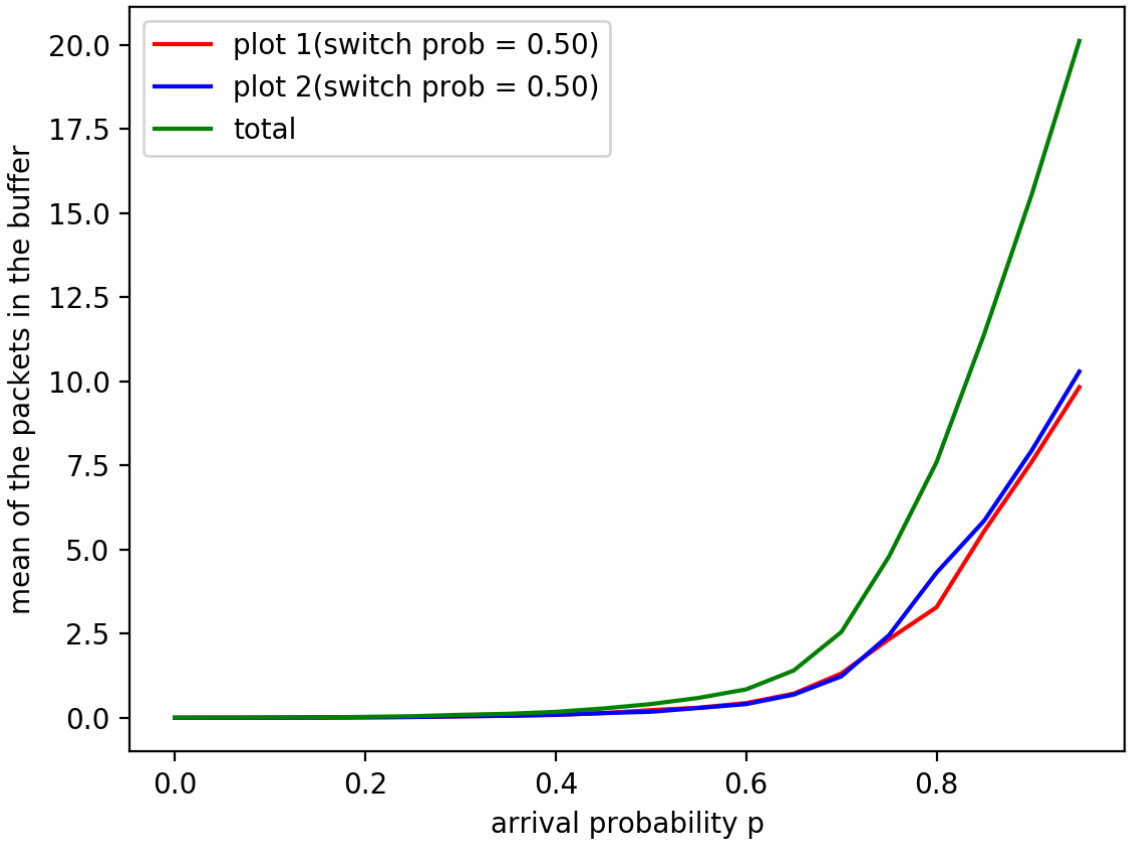
Question 2.a

Explane:

- Model the 2*2 HOL switch system is the following assumptions:
  - Buffer:
    - packages arrive at port i with the probability p in a time slot
    - packages' arrival will happen at the end of the slot
  - Switch:
    - if packages are different, the the switch attempt will success
    - if packages are the same, then the conflict will happen. And the the probability that switch the package in port 1 is 0.5, the probability that switch the package in port 2 is 0.5.
    - packages' switch will happen at the end of the slot.
- And define that the state of the system is the number of packages in the buffer at the middle of one slot.

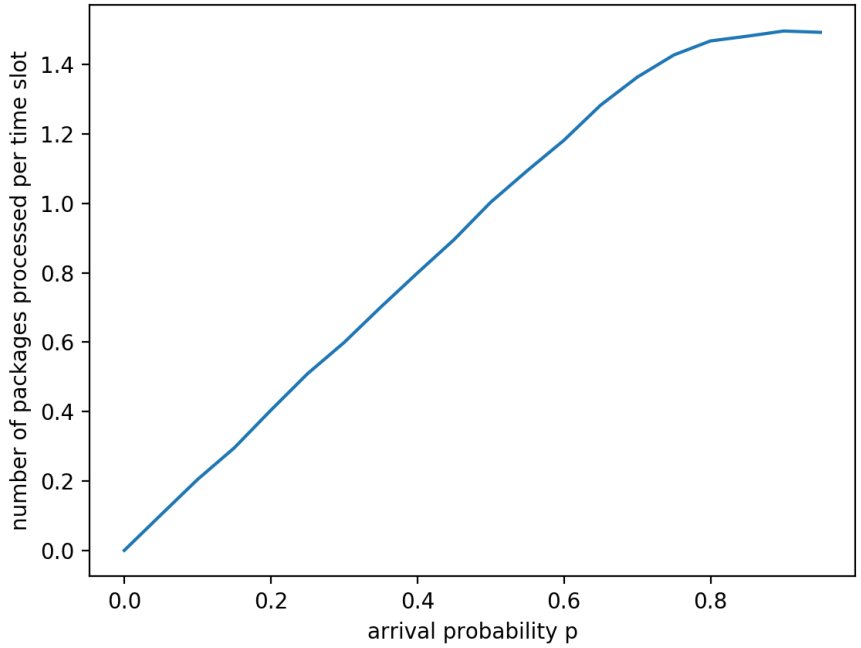
Results:

- **Plot of the mean of the number of packets in buffer in each slot at plot 1 and plot 2 and both plots (as a function of the arrival probability p):**



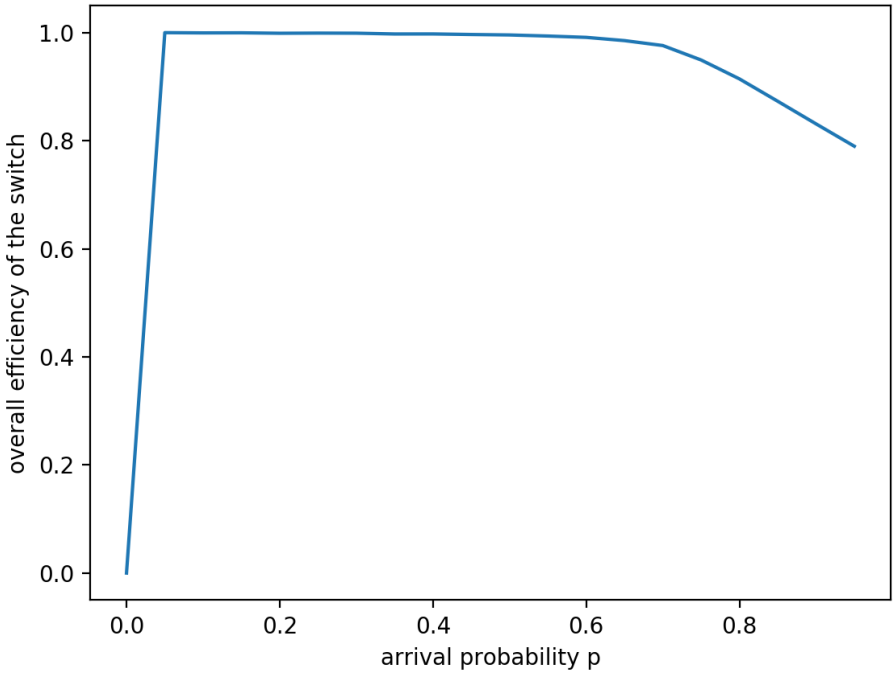
- when the switch prob are r1 = r2 = 0.5, the number of packets in buffer at plot 1 and plot 2 are similar.
- when the arrival probability p is increasing, the number of packets in buffer will increase too. And when p is greater than around 0.7, the number of packets in buffer will increase quickly.

- **Plot of the mean of the number of packets processed by switch in each slot (as a function of the arrival probability p):**



- when the arrival prob p is increasing, the number of packets processed in each slot is increasing too. And approximately, it gradually close to 1.5 when p > 0.8.

- **95% confidence interval for the overall efficiency of the switch**



```
p =  0.0 : [0.0, 0.0] 0.0
p =  0.05 : [1.0, 1.0] 1.0
p =  0.1 : [1.0, 1.0] 0.9995582706766918
p =  0.15000000000000002 : [1.0, 1.0] 0.999725975975976
p =  0.2 : [0.9786455630513752, 1.0] 0.9989154981343062
p =  0.25 : [0.9803921568627451, 1.0] 0.9992020889355437
p =  0.30000000000000004 : [0.9833189655172413, 1.0] 0.999025372245832
p =  0.35000000000000003 : [0.9833333333333333, 1.0] 0.997562235794866
p =  0.4 : [0.9862966133942161, 1.0] 0.9976207637519259
p =  0.45 : [0.9789242336610758, 1.0] 0.9965837483280517
p =  0.5 : [0.9722157320872274, 1.0] 0.9957366647733746
p =  0.55 : [0.9729605752541532, 1.0] 0.9937673540315882
p =  0.6000000000000001 : [0.9657051282051282, 1.0] 0.991289177787541
p =  0.65 : [0.9462838192114094, 1.0] 0.9852968014785988
p =  0.7000000000000001 : [0.9319129060074735, 1.0] 0.9761737001103842
p =  0.75 : [0.8838709677419355, 1.0] 0.9494492341126309
p =  0.8 : [0.8470479971129556, 0.9743387096774194] 0.914476798523401
p =  0.8500000000000001 : [0.8022033898305085, 0.9387230101187156] 0.8730896376672024
p =  0.9 : [0.7691576086956522, 0.8837707641196013] 0.8312352071713349
p =  0.9500000000000001 : [0.737079349393729, 0.8542410714285714] 0.789832647180878
```

- arrival probability p : 95% confidence interval, mean of efficiency
- overall efficiency = total number of switched / total number of arrival
- when the arrival prob is small (less than 0.6), efficiency is close to 1.0, which means almost all of the arrival packets can be switched after 200 slots when the number of arrival is small.
- when the arrival prob is bigger than around 0.6, efficiency will be worse, which means when there are more arrivals, the system will have more probability of facing conflicts and more packets will be remained in buffer, therefore the efficiency will be decreasing.

Code:

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

p = np.arange(0,1,0.05) #for different p (size = 20)
r1 = 0.50 #prob of switching plot 1
r2 = 0.50 #prob of switching plot 2
packets_mean = [] #mean of packets in different arrive_p
packets_mean_1 = []
packets_mean_2 = []
switch_mean = [] #switched packets per slot
effi_mean = []
for arrive_p in p:
    each_buffer_p = []
    each_buffer1_p = []
    each_buffer2_p = []
    each_switch = []
    effi_list = []
    for j in range (200):

        line1 = [] #line in plot 1
        line2 = [] #line in plot 2
        buffer_num = [] #num of packets in buffer
        buffer_num_1 = []
        buffer_num_2 = []
        switch_num = [] #num of packets switched
```

```
26          total_num = 0 #number of all arrival packets
27          total_switch = 0 #number of all switched packets
28          #100 slot
29          for slot in range(100):
30              #save state in this slot
31              buffer_num.append(len(line1) + len(line2))
32              buffer_num_1.append(len(line1))
33              buffer_num_2.append(len(line2))
            #packets arrive: random number in [0,1]; n<p, arrive;
    n>p not arrive
35              #plot1
36              arrive_plot1 = np.random.uniform(0,1)
37              if arrive_plot1 <= arrive_p:
38                  num = np.random.uniform(0,1) #0 or 1
39                  if num <= 0.5: line1.append(0)
40                  else: line1.append(1)
41                  total_num = total_num + 1
42              #plot2
43              arrive_plot2 = np.random.uniform(0,1)
44              if arrive_plot2 <= arrive_p:
45                  num = np.random.uniform(0,1) #0 or 1
46                  if num <= 0.5: line2.append(0)
47                  else: line2.append(1)
48                  total_num = total_num + 1
49              #switch
50              #both are empty: switch = 0
51              if len(line1) == 0 and len(line2) == 0:
52                  switch_num.append(0)
53              #line1 is empty and line2 is non empty:switch=1
54              elif len(line1) == 0 and len(line2) != 0:
55                  line2.pop(0)
56                  switch_num.append(1)
57                  total_switch = total_switch + 1
58              #line1 is nonempty and line2 is empty:switch=1
59              elif len(line1) != 0 and len(line2) == 0:
60                  line1.pop(0)
61                  switch_num.append(1)
62                  total_switch = total_switch + 1
63              #both are not empty
64              else:
65                  plot1 = line1[0]
66                  plot2 = line2[0]
67                  #different: switch = 2
68                  if(plot1 != plot2):
69                      line1.pop(0)
70                      line2.pop(0)
71                      switch_num.append(2)
72                      total_switch = total_switch + 2
73
                    #same: random number in [0,1]; n<r1, input1 s
    witch; n>r1, input2 switch; switch = 1;
74                  else:
```

```
                        switch = np.random.uniform(0,1)
                        if switch <= r1: line1.pop(0)
                        else: line2.pop(0)
                        switch_num.append(1)
                        total_switch = total_switch + 1
            each_buffer_p.append(np.mean(np.array(buffer_num)))

            each_buffer1_p.append(np.mean(np.array(buffer_num_1))
)

            each_buffer2_p.append(np.mean(np.array(buffer_num_2))
)
            each_switch.append(np.mean(np.array(switch_num)))
            if total_num == 0: efficiency = 0
            else: efficiency = total_switch/total_num
            effi_list.append(efficiency)
        mean_each_buffer_p = np.mean(np.array(each_buffer_p))
        mean_each_buffer1_p = np.mean(np.array(each_buffer1_p))
        mean_each_buffer2_p = np.mean(np.array(each_buffer2_p))
        mean_each_switch = np.mean(np.array(each_switch))
        mean_effi = np.mean(np.array(effi_list))
        #compute the mean of packets number in this p
        packets_mean.append(mean_each_buffer_p)
        packets_mean_1.append(mean_each_buffer1_p)
        packets_mean_2.append(mean_each_buffer2_p)
        #compute the mean of switch number per slot in this p
        switch_mean.append(mean_each_switch)
        #efficiency c.i.
        ci = [np.percentile(effi_list,(100-95)/
2),np.percentile(effi_list,(100+95)/2)]
        print("p = ", arrive_p, ":",ci,mean_effi)
        effi_mean.append(mean_effi)

#plt.subplot(121)

#plt.plot(p, packets_mean_1,'r', label = 'plot 1(switch prob
= 0.50)')

#plt.plot(p, packets_mean_2,'b', label = 'plot 2(switch prob
= 0.50)')
#plt.plot(p,packets_mean,'g', label = 'total')
#plt.xlabel('arrival probability p')
#plt.ylabel('mean of the packets in the buffer')
#plt.legend(loc='upper left')
#plt.subplot(122)
plt.plot(p, effi_mean)
plt.xlabel('arrival probability p')
plt.ylabel('overall efficiency of the switch')
plt.show()
```

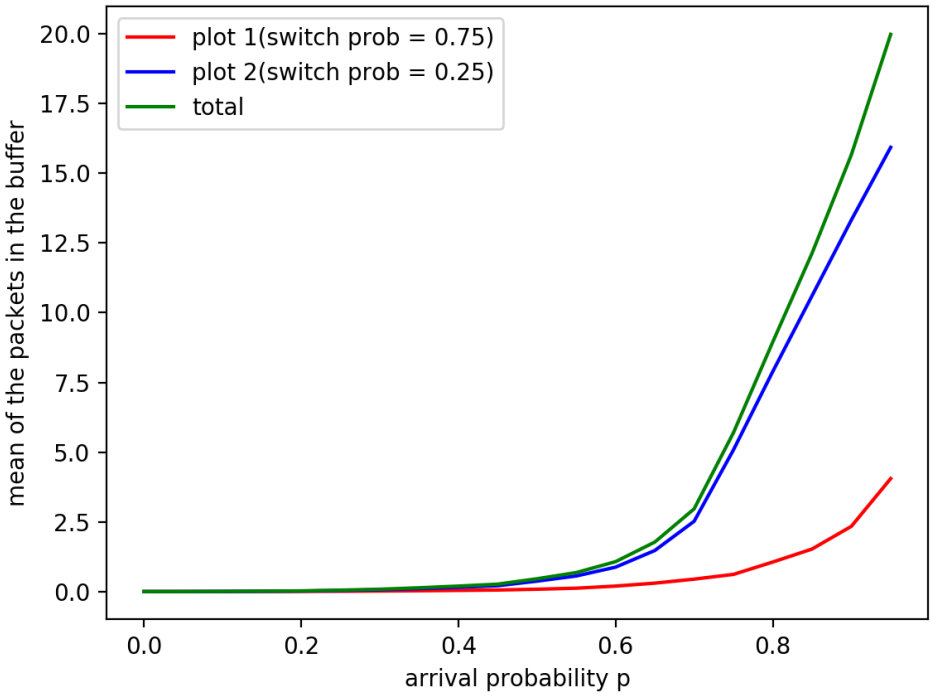
Question 2.b

Explane:

- Repeat all the steps in a with new switch probability: r1 = 0.75, r2 = 0.25

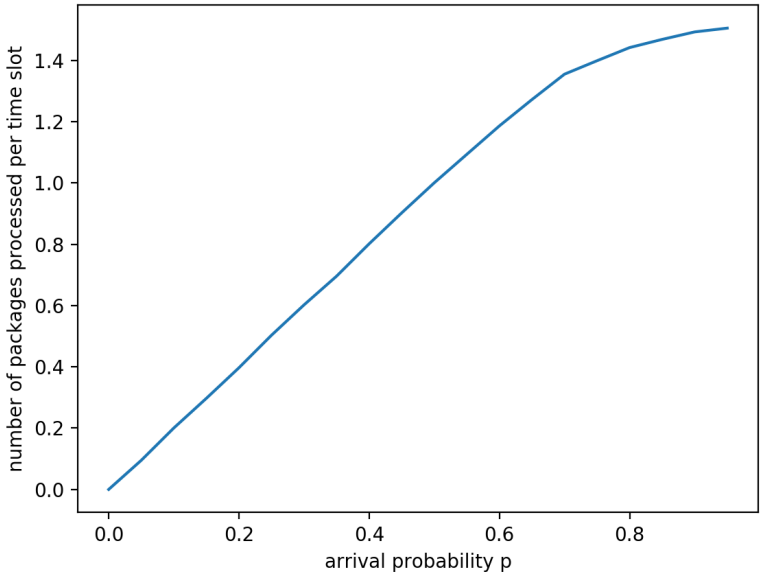
Results:

- **Plot of the mean of the number of packets in buffer in each slot at plot 1 and plot 2 and both plots (as a function of the arrival probability p):**



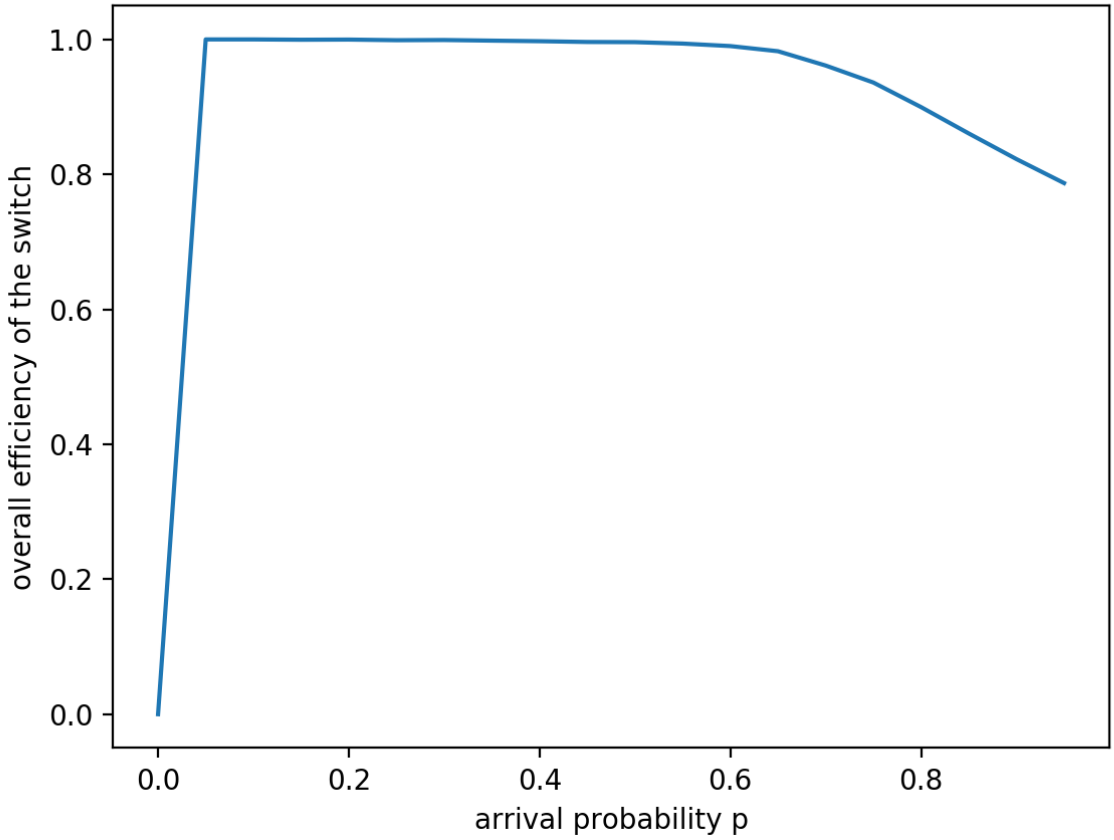
  - when the switch prob are r1 = 0.75, r2 = 0.25, the number of packets in buffer at plot 1 and plot 2 are different. Plot 1 is much smaller than plot 2. The plot with higher switch probability can have more number of packets switched, thus will left less number of packets in buffer.
  - when the arrival probability p is increasing, the number of packets in buffer will increase too.

- **Plot of the mean of the number of packets processed by switch in each slot (as a function of the arrival probability p):**



- when the arrival prob p is increasing, the number of packets processed in each slot is increasing too. And approximately, it gradually close to 1.5 when p > 0.8.

- **95% confidence interval for the overall efficiency of the switch**




```
p =  0.0 : [0.0, 0.0] 0.0
p =  0.05 : [1.0, 1.0] 1.0
p =  0.1 : [1.0, 1.0] 1.0
p =  0.15000000000000002 : [1.0, 1.0] 0.9995244507655994
p =  0.2 : [1.0, 1.0] 0.9997620380739082
p =  0.25 : [0.9795701259227094, 1.0] 0.9988345685549774
p =  0.30000000000000004 : [0.9835997267759563, 1.0] 0.9991127598416163
p =  0.35000000000000003 : [0.9848304473304473, 1.0] 0.9981955428863027
p =  0.4 : [0.984700956937799, 1.0] 0.997359305155359
p =  0.45 : [0.9752765752765752, 1.0] 0.9962116981402772
p =  0.5 : [0.98, 1.0] 0.9959468483122432
p =  0.55 : [0.973208252895753, 1.0] 0.9938468627078811
p =  0.6000000000000001 : [0.9605979130657448, 1.0] 0.9901478405405417
p =  0.65 : [0.9383695652173913, 1.0] 0.9825016132488983
p =  0.7000000000000001 : [0.9005233177418608, 1.0] 0.9613100978240979
p =  0.75 : [0.865224358974359, 0.9932897091722594] 0.9361432724203758
p =  0.8 : [0.8281213980293735, 0.9568703703703704] 0.8997367850607637
p =  0.8500000000000001 : [0.7998571428571429, 0.9259485094850949] 0.8606703692498283
p =  0.9 : [0.7646099744245524, 0.8779932006306661] 0.8226844655440019
p =  0.9500000000000001 : [0.7408989097582038, 0.8494623655913979] 0.7870754688974745
```

- overall efficiency = total number of switched / total number of arrival
- when the arrival prob is small (less than 0.6), efficiency is close to 1.0, which means almost all of the arrival packets can be switched after 200 slots when the number of arrival is small.

- when the arrival prob is bigger than around 0.6, efficiency will be worse, which means when there are more arrivals, the system will have more probability of facing conflicts and more packets will be remained in buffer, therefore the efficiency will be decreasing.

Code: just the same as (a) change: r1 = 0.75, r2 = 0.25

Question 3

Explain:

- Use the Wright-Fisher model to simulate stochastic genotypic drift during successive generations. This model is a Marcov chain, so we can initial the start state, compute the transition matrix and produce generations for 100 times, see if it will converge to a certain state.
- In test, find that after 100 times, the system is still not converged. So I set 2000 times, and it will converge to certain state.
-

Results:

- When the start state of initial allele distribution is
  - **A1A1 = 0, A1A2 = 100, A2A2 = 0**:

```
Convergence after 1643  iterations
Composition at this iteration:
[4.94803316e-01 1.62308560e-06 1.82085670e-06 1.87797616e-06
 1.90540084e-06 1.92273922e-06 1.93442294e-06 1.94279347e-06
 1.94912040e-06 1.95408557e-06 1.95809122e-06 1.96139453e-06
```

```
1.94912041e-06 1.94279348e-06 1.93442295e-06 1.92273923e-06
1.90540085e-06 1.87797617e-06 1.82085672e-06 1.62308561e-06
5.04803316e-01]
```

  - the state is convergence after **1643 itertions**
  - the final state is x0 = **0.4948**, x200 = **0.5048**
  - genetic compositon: initial **A1:A2 = 1:1** , end **A1:A2 = 1:1**

- When the start state of initial allele distributionis is
  - **A1A1 = 50, A1A2 = 0, A2A2 = 50**

```
Convergence after 1643  iterations
Composition at this iteration:
[4.94803316e-01 1.62308560e-06 1.82085670e-06 1.87797616e-06
 1.90540084e-06 1.92273922e-06 1.93442294e-06 1.94279347e-06
 1.94912040e-06 1.95408557e-06 1.95809122e-06 1.96139453e-06
 1.96416821e-06 1.96653244e-06 1.96857338e-06 1.97035442e-06
```

```
1.97669920e-06 1.97568401e-06 1.97456245e-06 1.97331640e-06
1.97192324e-06 1.97035443e-06 1.96857339e-06 1.96653245e-06
1.96416822e-06 1.96139454e-06 1.95809123e-06 1.95408558e-06
1.94912041e-06 1.94279348e-06 1.93442295e-06 1.92273923e-06
1.90540085e-06 1.87797617e-06 1.82085672e-06 1.62308561e-06
5.04803316e-01]
```

  - the state is convergence after **1643 itertions**
  - the final state is x0 = **0.4948**, x200 = **0.5048**

- genetic compositon: initial **A1:A2 = 1:1** , end **A1:A2 = 1:1**

- When the start state of initial allele distributionis is
  - **A1A1 = 20, A1A2 = 0, A2A2 = 80**

```
Convergence after 1558  iterations
Composition at this iteration:
[7.94803649e-01 1.62033262e-06 1.81776827e-06 1.87479083e-06
 1.90216899e-06 1.91947795e-06 1.93114185e-06 1.93949817e-06
 1.94581436e-06 1.95077110e-06 1.95476995e-06 1.95806765e-06
 1.96083661e-06 1.96319682e-06 1.96523429e-06 1.96701230e-06
 1.96857689e-06 1.96701073e-06 1.96523271e-06 1.96319522e-06
 1.96083500e-06 1.95806602e-06 1.95476830e-06 1.95076944e-06
 1.94581268e-06 1.93949649e-06 1.93114015e-06 1.91947625e-06
 1.90216729e-06 1.87478914e-06 1.81776661e-06 1.62033113e-06
 2.04803649e-01]
```

  - the state is convergence after **1558 itertions**
  - the final state is x0 = **0.7948**, x200 = **0.2048**
  - genetic compositon: initial **A1:A2 = 2:8** , end **A1:A2 = 2:8**

- When the start state of initial allele distributionis is
  - **A1A1 = 20, A1A2 = 30, A2A2 = 50**

```
Convergence after 1626  iterations
Composition at this iteration:
[6.44803814e-01 1.61897668e-06 1.81624712e-06 1.87322197e-06
 1.90057722e-06 1.91787171e-06 1.92952585e-06 1.93787518e-06
 1.94418609e-06 1.94913869e-06 1.95313420e-06 1.95642915e-06
 1.95919580e-06 1.96155404e-06 1.96358982e-06 1.96536634e-06
 1.96693079e-06 1.96536595e-06 1.96358942e-06 1.96155365e-06
 1.95919540e-06 1.95642874e-06 1.95313379e-06 1.94913828e-06
 1.94418568e-06 1.93787477e-06 1.92952543e-06 1.91787129e-06
 1.90057680e-06 1.87322155e-06 1.81624671e-06 1.61897631e-06
 3.54803814e-01]
```

  - the state is convergence after **1626 itertions**
  - the final state is x0 = **0.6448**, x200 = **0.3548**
  - genetic compositon: initial **A1:A2 = 70:130 = 35:65** , end **A1:A2 = 35:65**

- **Composition of the initial population will decide the steady-state outcome! The steady-state outcome is the same as the composition of the initial population!**
- **Why does this scenario seem to defy the assertions of the Perron−Frobenius theorem and the Markov chain ergodic theorem?**
  - the entries of transition matrix P are not all positive, some are 0.

- different initial genetic composition will decide the final composition. And the transition matrix is not irreducble. For example, if we start at A1 in range (0, 200) and not equal to 0 or 200, then in the following generations, the composition will gradually drift to 0 and 200, thus will never get back to the original state.

Code:

```
import numpy as np
from scipy.special import comb

#number of individuals
N = 100
num_A1A1 = 0
num_A1A2 = 100
num_A2A2 = 0
#initial distribution
input = np.zeros((2*N+1,)) #initial distribution
num_A1 = 2*num_A1A1 + num_A1A2
input[num_A1+1] = 1
# transition matrix
P = np.zeros((2*N+1, 2*N+1))
for i in range(0,2*N+1):
    for j in range(0,2*N+1):

        P[i,j] = comb(2*N,j, exact=True, repetition=False)*((i)/(2*N))**(j))*((1-(i)/(2*N))**(2*N-j))

#number of steps to take
t = 2000
output = np.zeros((t+1,2*N+1)) #save all the output state
#generate first output value
output[0,:] = input
#do the M.C. t times
for i in range(1,t+1):
    output[i,:] = np.dot(output[i-1,:],P)
    if np.allclose(output[i,:],output[i-1,:]):
        print('Convergence after '+str(i),' iterations')
        print('Composition at this iteration:')
        print(output[i,:])
        break
print(output)
```