ECS 256: Term Project

Qizhou Fang, Wenru Feng, Jane Park, Yun Qin

Contents

Chapter 1

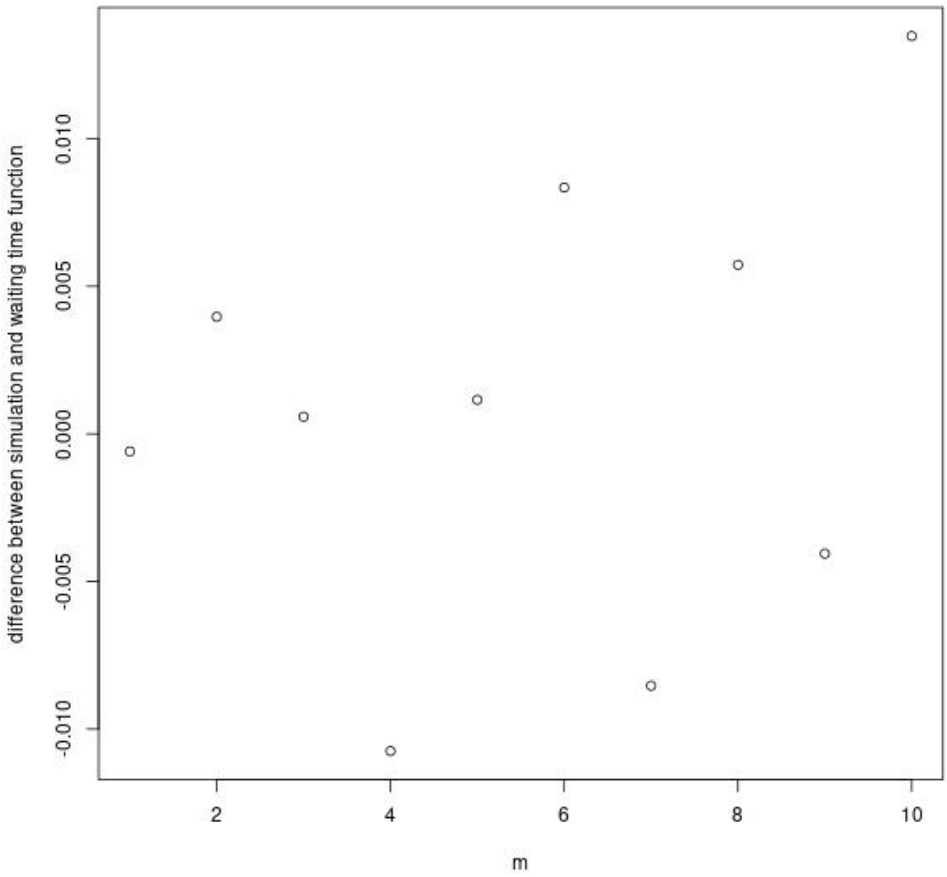
Problem A

1.1 Health Advising System Model

To allocate the resources appropriately in the health advising system, we want to keep the patient happy as well as to reduce the cost for hiring nurses.
There are five parameters that affects the advising system in the model. The call duration is modeled by a exponential random variable with mean μ. It is a fixed parameter as the call duration should not be controlled artificially and it is hardly affected by the change of time and seasons.
The parameter inter arrival time is a exponentially distributed variable with mean v in the model. It is also a uncontrollable parameter. However, it can vary depending on the different seasons(busy or not). Our strategy of adding and shrinking the number of nurses deals with this problem. The probability of receiving a patient when the waiting room is full by adding a nurse(we call it p) is a parameter that can be changed. But in our investigation, we will fix it to be 1. Because a higher p will improve the average waiting time and rejection rate, which are the important criterion we want to satisfy.
The number of nurses(we call it n) available to handle calls and the size of the waiting room(we call it m) are the two parameters we are manipulating. Through a good choice of the two, we make the system to be in favor of client's interest.
State (i, j) in our model signifies that currently there are i people combined of people who are calling the nurses and people who are waiting; and there are j nurses handling the call.
We primarily use three metrics to measure the success of the model: the mean waiting time for each patient, the probability of rejecting a call and the number of nurses ready to serve patients(n). We used this formula to compute the mean waiting time. The result is consistent with simulation.

$$\text{mean queue length} \cdot \frac{\text{mean service time}}{\text{mean number of meetings}}.$$

Figure 1.1: The difference between our measure of waiting time and the simulation result is sufficiently small



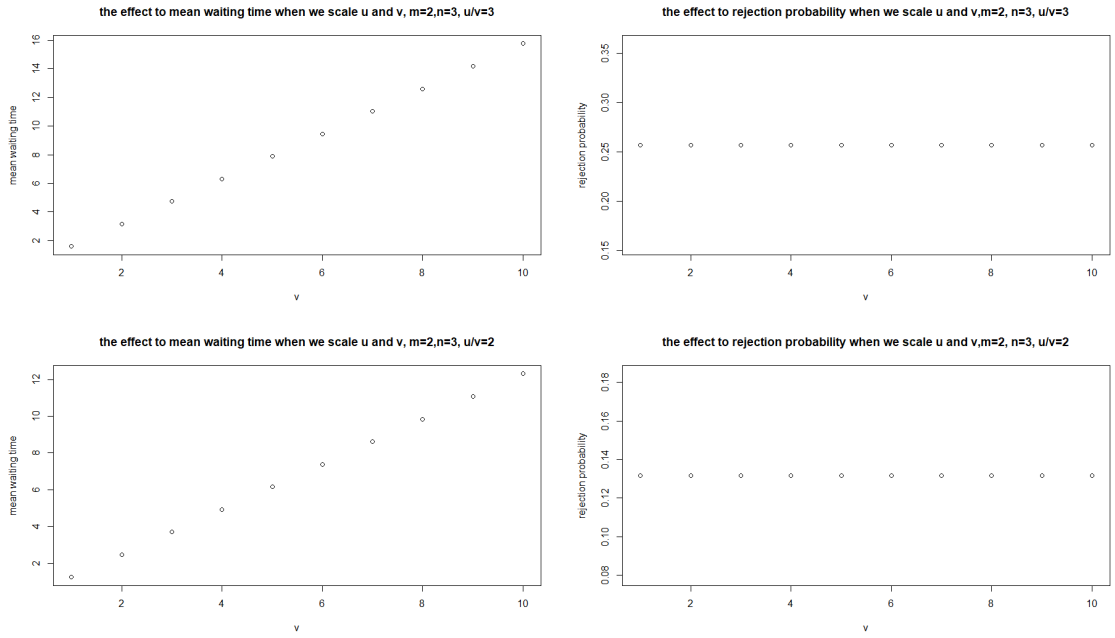
The mean number of nurses talking to patients is one thing we may be interested in. However, even if the mean number of nurses at work is low we still need to pay the rest of the nurses who are waiting but not actually working. Average queue length is important but it is reflected in the mean waiting time.

The mean waiting time is crucial because waiting too long will make patients unhappy. Being rejected results the patients to be even more unhappy then if they waited for a long time. So we make rejection rate a stricter criteria. Finally, the number of nurses ready should be minimized to reduce the cost of the hospital.

Therefore, we use the following logic to select an appropriate combination of m and n. Without touching the bottom line of displeasing the patients, we want to minimize the cost of hiring nurses. In the language of the model, we want the probability of rejection and the mean waiting time for

a patient to be below some thresholds. Under such condition, we want to minimize the number of n. We made the threshold for waiting time to be half of the average call duration and the rejection rate to be 0.01.
In our model, we did not make choice of parameters μ and v according to data collected in the field. This is for the following reason. Through experiments, we found that the rejection probability barely changes when μ or v changes, as long as we maintain their ratio. The average waiting time increases linearly as the two parameters increase when their ratio is maintained. Such result is intuitive. As a result, if we make the success criteria of success for rejecting a patient to be below a constant and the criteria for waiting time proportional to μ, we only need to manipulate the ratio between μ and v.

Figure 1.2: We see as we scale μ and v up, the waiting time increases linearly and the rejection rate almost remains the same



Then, we fix v to be 1 and iterate through u from 0.2 to 5 to investigate the cases for u, v ratio from 0.2 to 5. Each number of u is one individual case we study. We limit the number of nurses to be less than 10 as that is a decent amount. For each combination of m and n, we examine if they satisfy the two criterion. For the ones that pass the test, we choose the combination with the least amount of nurses. This will be our optimal choice for the size of the waiting room and the number of nurses. The result is on the table below.

1.2 Parameter Selection Guideline

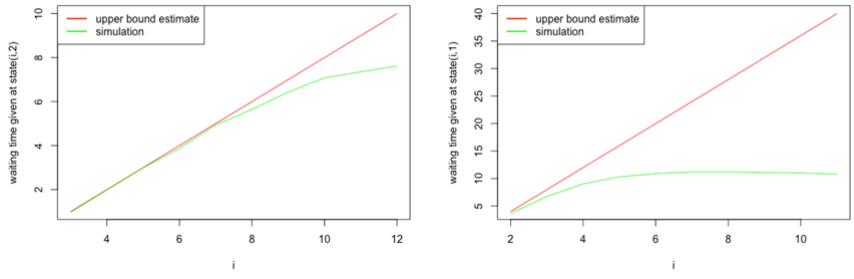
We use the table below as our guideline to choose the number of nurses and the waiting room size. When the health advising system is operating, we can keep track of the average calling duration and the average intercall arrival time within the recent 3 days. We use them as our estimated μ_{est} and v_{est}. We then compute the $r = \dfrac{\mu_{\text{est}}}{v_{\text{est}}}$ ratio and we look at the column with u/v closest to r. For example, if we have $r = \dfrac{\mu_{\text{est}}}{v_{\text{est}}} = 4,3$ then we look at the column that correspond to $u/v = 4.2$. So we should have a waiting room capacity $m = 2$ and total $n = 9$ nurses available.

u/v	0.2	0.6	1	1.4	1.8	2.2	2.6	3	3.4	3.8	4.2	4.6	5
best n	3	4	5	5	6	6	7	8	8	9	9	10	10
best m	0	0	0	1	0	2	1	0	2	1	2	1	3
mean waiting	0	0	0	0.5043483	0	1.0117765	0.5052159	0	1.0110175	0.5047482	1.0134598	0.5057184	1.5195112
rejection probability	0.00109170	0.00296475	0.00306748	0.00708509	0.00782870	0.00875179	0.00814187	0.00813244	0.00793653	0.00726935	0.00948017	0.00862170	0.00853441

1.3 Approximating Waiting Time Given Our Position in the Queue and the Number of Nurses

We want to find the conditional waiting time E(waiting time|we are at state(i,j)). If we have it, we can notify the patients their expected waiting time when they arrive at the queue. From the time we enter the line to when we are served, the number of nurses can't decrease because the queue length is non-zero during this time. Therefore, our mean waiting time is less or equal than the mean waiting time if we always had j nurses. When the j is less than the ratio u/v, it means the queue will steadily grow and we are likely to add nurses at some point before we are served. Otherwise, the number of nurses are likely to stay the same unless the waiting room is almost filled. Our idea to compute the conditional waiting time is based on this pattern. First compute the probability p of adding nurse before we are served. Then we take the waiting time to be p times the waiting time given no nurses will ever be added plus $1 - p$ times a mixture of the waiting time when nurses are not added and when nurses are added.

Figure 1.3: in the first plot, j is no less than the uv ratio so until i is close to waiting room limit, the waiting time is close to the upper bound. But when j is less than the ratio as in plot 2, the conditional waiting time is always much less than the upper bound



We define a function f which we are going to call recursively. It takes arguments:RQ, i, j, m, n. RQ stand for the remaining queue length. In the function, AQ stands for the average number of queue decrease happened given we added nurses before served. g and h are some functions.

$f(RQ, i, j, m, n)$:
  $AQ = g(i, j, m, n)$
  $p = h(i, j, m, n)$
  if $j == n$ or $AQ >= RQ$ **then**
    return $RQ\frac{u}{j}$
  else
    return$(p(AQ\frac{u}{j} + (RQ - AQ)f(RQ - AQ, i, j + 1, m, n)) + (1 - p)RQ\frac{u}{j})$
  end if

First, let RQ be the queue length and call function f(RQ,i,j,m,n), we can get the result. However, since we were not able to compute p and AQ as it is similar but more difficult than the gambler's ruin problem, we could not write the corresponding code.

Chapter 2

Problem B

List of Variables and Parameters

`trials`	The number of independent trials, set at default value `10000`
`a`,b	The lower bound, and upper bound of the connected interval $[a, b] \subset \mathbb{R}$ for unif($[a, b]$)
`q`	The number of equidistantly spaced quantiles
`rv`	The number of exponential stages
`lambda`	The parameter of the i.i.d. exponential states Greek symbol λ will be used only some contextual circumstances
`target_mean`	The mean of a given distribution (uniform and folded normal), in our coded examples
`target_var`	The variance of a given distribution (uniform and folded normal), in our coded examples
`mean`	The mean of a given trial's *method of stages* simulations
`var`	The variance of a given trial's *method of stages* simulations
`pi_vec`	The pi vector associated to a given matrix transition matrix P

2.1 Summary of goal

We investigate the *method of stages* (MoS), which uses Markov chain methods on non-Markovian processes. In these chains, the holding times must be exponentially distributed in order to attain and apply the Markov property. Thus, we inspect continuous distributions which are non-memoryless (i.e., non-exponenential). We investigate the uniform distribution as a canonical example of a simple *bounded* non-exponential distribution, and folded normal distribution as a representative of an *unbounded* non exponential distribution. Here, when we say [*un*]*bounded distribution*, we mean that the support of the outcomes of the random variables are from an un/bounded connected subset of $\mathbb{R}$.

2.2 Bounded - Uniform Distribution

We would like to explore how the number of quantiles `q`, and the number of states `rv` with exponential holding time distributions affect the accuracy of MoS. Within these `rv` i.i.d. exponentially distributed stages, we must also observe how the choices of exponential parameter `lambda` affect that accuracy as well. Also the way we select the parameter `lambda` for the exponential random variables (identically or randomly chosen parameters) affect the approximation result. Here we designed a simulation experiment to explore these effects, we selected unif$(1, 2)$, with mean $\mu =$ `target_mean` $= 1.5$, variance as $\sigma^2 =$ `target_var` $= 1/12 \approx 0.833$. Taking `10000` trials for each simulation.

2.2.1 The number of quantiles `q`

Denote the number of quantiles as `q`, for some `q` ≥ 1. According to the blog (Nov. 17, 0750), the quantiles have to be uniform, so the probability of each quantile is `1/q`. Here we suppose the larger the number of exponential random variables we use, the better the approximation result we can get, we take the number of exponential random variables as `rv = 1000`. We selected to use equal `rv` and `lambda` for all the exponential random variables to simplify the calculation.

Table 2.1 and Figure 2.1 show the simulation results with varying choices of `q`. The absolute difference of mean is:

```
abs_diff_mean = abs(mean - target_mean) = abs(mean - 1.5).
```

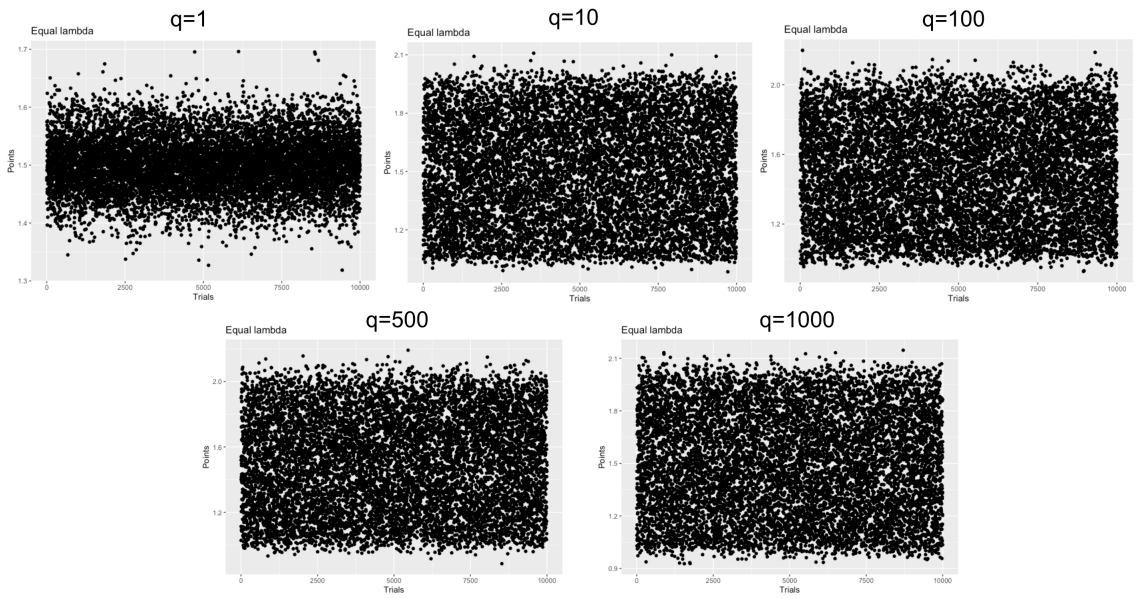
The absolute difference of variance is:

```
abs_diff_var = abs(var - target_variance) = abs(var - 1/12).
```

Table 2.1: Results with different number of quantiles

q	mean	abs_diff_mean	var	abs_diff_var
1	1.500644	0.0006436835	0.002278724	0.08105461
10	1.498552	0.001448175	0.07015399	0.01317934
100	1.496853	0.00314678	0.08296554	0.0003677937
500	1.50181	0.001809608	0.08497675	0.001643419
1000	1.499178	0.0008219751	0.08527087	0.001937534

Figure 2.1: Results with different number of quantiles



From both the simulation data and plots, we can conclude that the more quantiles we have, the better approximation results we get, as determined by the errors between means and variances of our example target uniform distribution and our approximations varying `q`.

2.2.2 The number of exponentially distributed stages `rv`

Denote the number of exponential random variables as `rv`, for some `rv` ≥ 1. Here we already proved that the larger the number of quantiles is, the better the approximation result we can get, we take the number of quantiles `q = 1000`. We selected to use equal `lambda` for all the exponential random variables to simplify the calculation. What is the value of `lambda`?

Table 2.2 and Figure 2.2 show the simulation results with varying choices of `rv`. The absolute difference of mean is:

```
abs_diff_mean = abs(mean - target_mean) = abs(mean - 1.5).
```

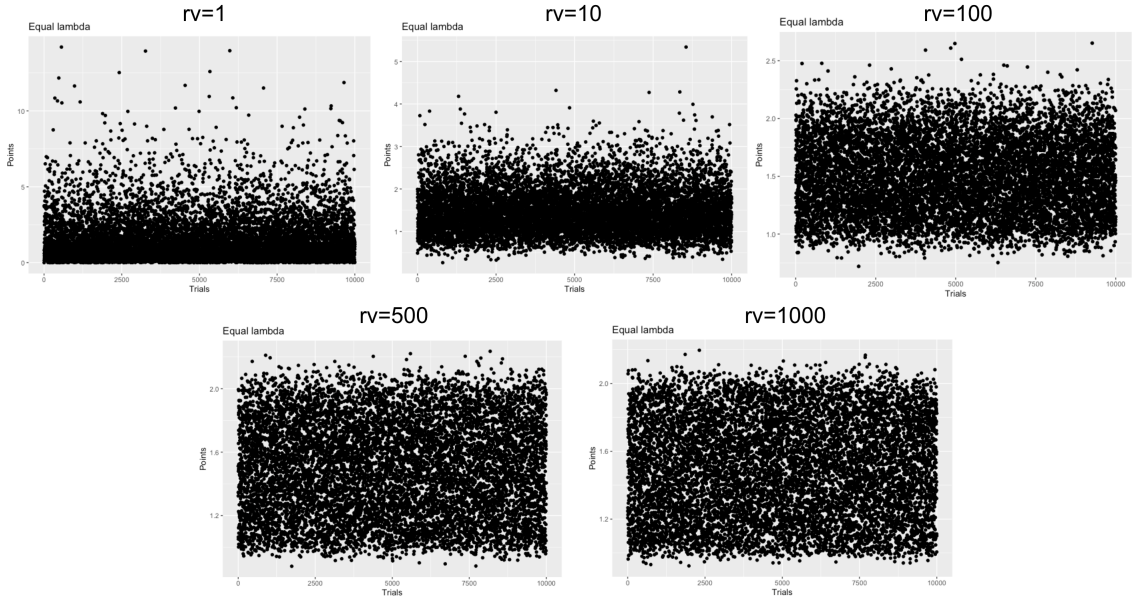
The absolute difference of variance is

```
abs_diff_var = abs(var - target_var) = abs(var - 1/12).
```

Table 2.2: Results with different number of exponential random variables

rv	mean	abs_diff_mean	var	abs_diff_var
1	1.494744	0.005256351	2.375037	2.291704
10	1.493348	0.006652106	0.2949088	0.2115755
100	1.500201	0.0002013491	0.1072753	0.02394201
500	1.498113	0.001887456	0.08805417	0.004720835
1000	1.500543	0.0005432369	0.08478476	0.001451425

Figure 2.2: Results with different number of exponential random variables



From both the simulation data and plots, we can conclude that the more exponential random variables we have, the better approximation results we get.

2.2.3 Methods to select the stages' exponential distribution parameter `lambda`

For each quantile, we can approximate it as the sum of `rv` exponential random variables. Here we need sum of the mean values of these random variables to match the quantile value, then there would be two methods: one is to have all the `lambda` equal (so that we have an *Erlang distribution*), the other is to assign `lambda` randomly while having the sum equal to the quantile's value, as needed. The parameter `lambda` = `1/mean`.

When approximating a constant c using the sum of several exponential random variables, we want to use the optimized parameters. It is of course required that the sum of means of these random

variables is equal to the constant c. A better choice of the parameters therefore is the choice that minimizes the variance of their sum.
the formulation of the problem is $X_1, X_2, \cdots, X_n$ are the exponential variables to approximate the constant c, X_i belongs to $\exp(\lambda_i)$.

Let $Y = X_1 + X2 + \cdots + X_n$.

Then $E(Y) = c$ so $\sum_{i=1}^{n} \frac{1}{\lambda_i} = c$.

The variance of Y is

$$\mathrm{Var}(Y) = \mathrm{Var}(X_1 + X2 + \cdots + X_n) = \mathrm{Var}(X_1) + \mathrm{Var}(X2) + \cdots + \mathrm{Var}(X_n) = \sum_{i=1}^{n} \frac{1}{\lambda_i^2} = c.$$

the problem became one of a constrained optimization case:

$$f(x_1, x_2, \cdots, x_n) = \sum_{i=1}^{n} \frac{1}{x_i^2} \quad \text{and} \quad h(x_1, x_2, \cdots, x_n) \sum_{i=1}^{n} \frac{1}{x_i}.$$

We want to minimize $f(\vec{x})$ given $h(\vec{x}) = 0$, which according to the *KKT condition*, the optimized $\vec{x}$ satisfies: $0 = \partial f(\vec{x}) + \lambda \partial h(\vec{x})$ and $h(x) = 0$.

$$\vec{0} = \begin{pmatrix} -2/x_1^3 \\ -2/x_2^3 \\ \vdots \\ -2/x_n^3 \end{pmatrix} + \begin{pmatrix} -\lambda/x_1^2 \\ -\lambda/x_2^2 \\ \vdots \\ -\lambda/x_n^2 \end{pmatrix} \quad \text{and} \quad h(\vec{x}) = 0.$$

Solving the equations, we get $-\lambda = \frac{2}{x_1} = \frac{2}{x_2} = \cdots = \frac{2}{x_n}$
therefore $x_1 = x_2 = \cdots = x_n$, all the exponential random variables should have the same parameters.

From the above theory, we verify that choosing one `lambda` for all exponential holding time distributions has smaller variance which should be a better choice. However, we would like to use a simulation to see the difference between the two methods.

From the simulation, we have shown that the higher both `q` and then `rv` are, the smaller the error between the original system and our MoS approximated system via the ℓ^1-norm. We would like to compare the two methods with the case when `q = 1000`, `rv = 1000`.

Table 2.3 and Figure 2.3 show the simulation results of the two methods. The error determined by the ℓ^1-norm (absolute difference) in the mean value is

```
abs_diff_mean = abs( mean - target_mean ) = abs( mean - 1.5 ).
```

Similarly, the error in variance value is

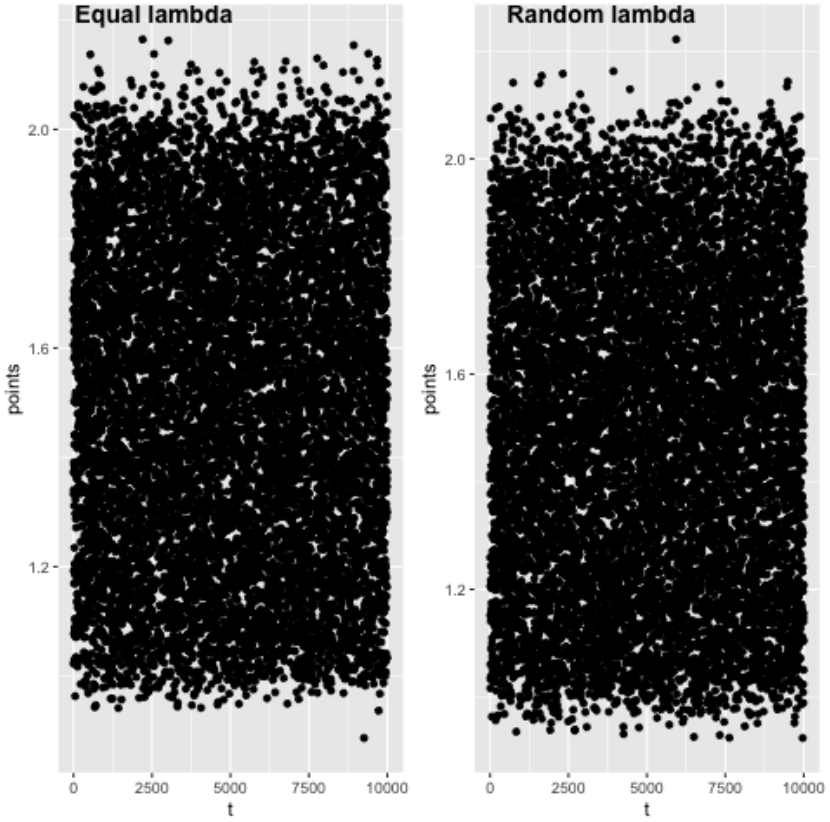
```
abs_diff_var = abs( var - target_var ) = abs( var - 1/12 ).
```

The execution time is the time the system takes to generalize the points.

Table 2.3: Comparison on equal and random method

	equal	random
mean	1.500695	1.498663
abs_diff_mean	0.0006949667	0.001337182
var	0.08462845	0.08594688
abs_diff_var	0.00129512	0.002613544
execution time	1.748195 s	45.27267 s

Figure 2.3: Comparison on equal and random method



From both the simulation data and plots, we can see that the equal method is slightly better with closer mean and variance to the expected values, however, the performance of the two methods on approximation is still very similar. The huge difference between the two methods is the time they take to generalize the points, the equal method is a lot faster, which makes it a better choice for our following simulations.

2.3 Unbounded - Folded/Truncated Normal Distribution

Please note: When we write about the "truncated normal" distribution or refer to functions in the `truncnorm R` *package at any point, including in our figures, we do mean "folded normal" distribution. In the* `truncnorm` *package, we may set* `b=Inf` *in the parameters of the functions, which is equivalent to the folded normal with infinite measure support.*

Because we need the support of our random variables to be non-negative in our Markov Chain, we have to "fold" the normal distribution at 0. From the conclusions in section 1, we know that the higher `q` and `rv` are the lower our ℓ^1-norm errors (i.e. the better our approximation results we get). However, due to the calculation capacities of our choice of consumer-grade hardware, we would like to know what choice of `q` and `rv` will give us sufficiently good accuracy between a given chain of folded normally distributed system, and an MoS approximation of that system.

As found in Section 2.2.3, the cost and benefits of choosing identically distributed, versus randomly chosen parameters for our exponentially distributed holding time parameters points to choosing identically distributed exponential distributions (between any two consecutive quantile) not only being more accurate in comparing means and variances to the target mean and variance, it is also computationally faster. Therefore, we choose to have our exponential distributions between any two consecutive quantiles be identically distributed, and therefore Erlang.

2.3.1 Simulation

We would like to perform a basic simulation experiment with `q = 1000`, `rv = 1000` using identical choice of `lambda` and `trials = 10000` to see how our MoS works on a chain of folded normal distributions. Here we inspect how MoS approximates one folded normal distribution with `mu = 0`, `sigma = 1`.

Table 2.4 and Figure 2.4 show the simulation results with `q = 1000`, `rv = 1000` and equal mean method for parameter `lambda`.

The ℓ^1-norm difference for the mean is:

$$\texttt{abs\_diff\_mean} = \texttt{abs(mean - target\_mean)} = \texttt{abs(mean - 0.7978846)}.$$

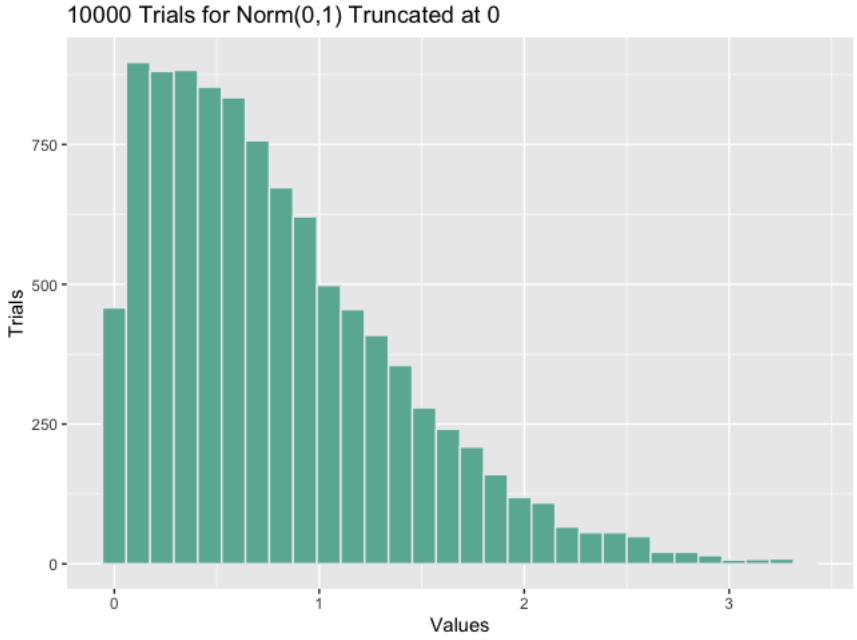
Likewise:

$$\texttt{abs\_diff\_var} = \texttt{abs(var - target\_var)} = \texttt{abs(var - 0.3633802)}.$$

Table 2.4: Simulation results for truncated normal distribution

mean	abs_diff_mean	var	abs_diff_var
0.7948102	0.003074394	0.3574479	0.005932353

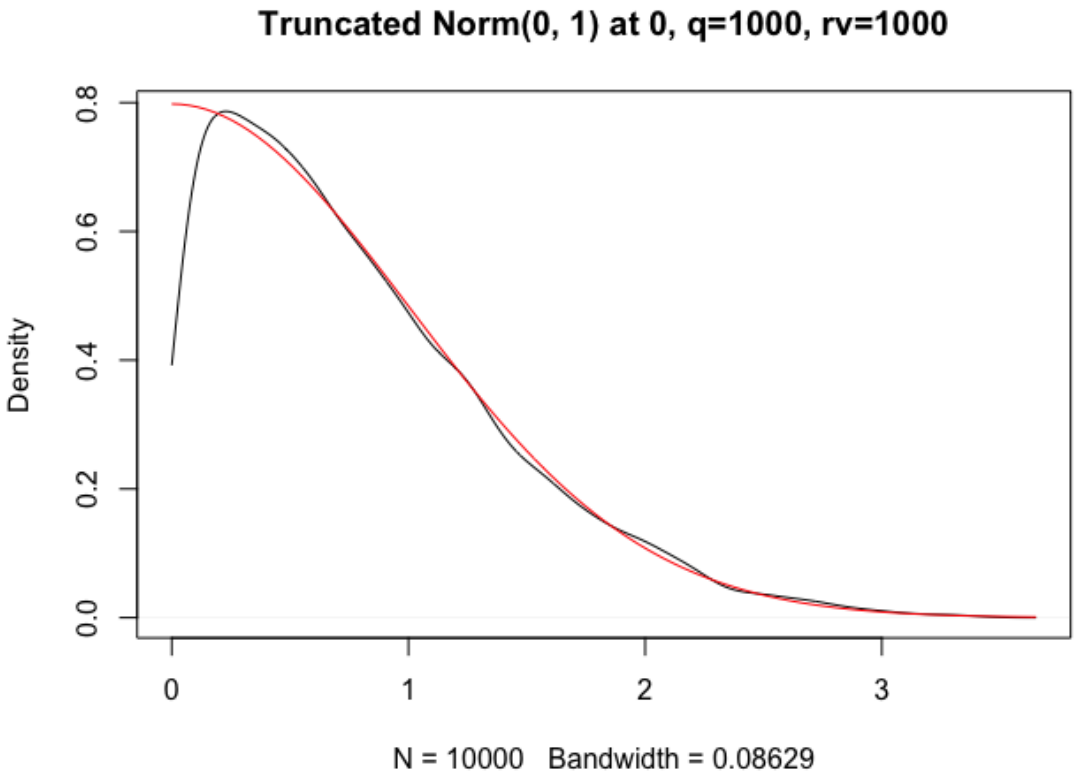
Figure 2.4: Histogram of the simulation results



2.3.2 Interacting choices of `q` and `rv`: Speculations on Optimization

We have noticed in our data that the choice of `q` $\geq$ `rv` or `q` $\leq$ `rv` depends highly on the relationship between the values of `target_mean` and `target_var`. Of course, by principles of Riemann sums in maximizing `q`, mixed with approximations via something similar to the *Stone-Weierstraß theorem* with sums of exponential functions (as opposed to polynomials, however the Taylor series of smooth functions gives rise to the polynomial aspect of these exponential functions) in maximizing `rv` that we want to just have very high values of both numbers. However, it is very difficult to suggest telling a computer to take arbitrarily large matrix computations of size $(\texttt{q*rv})^2$ (taking the number of states in the system to be `1` for preliminary simplicity.

Figure 2.5: Choosing high `q` and `rv` values will yield a more accurate MoS approximation for probability distributions.



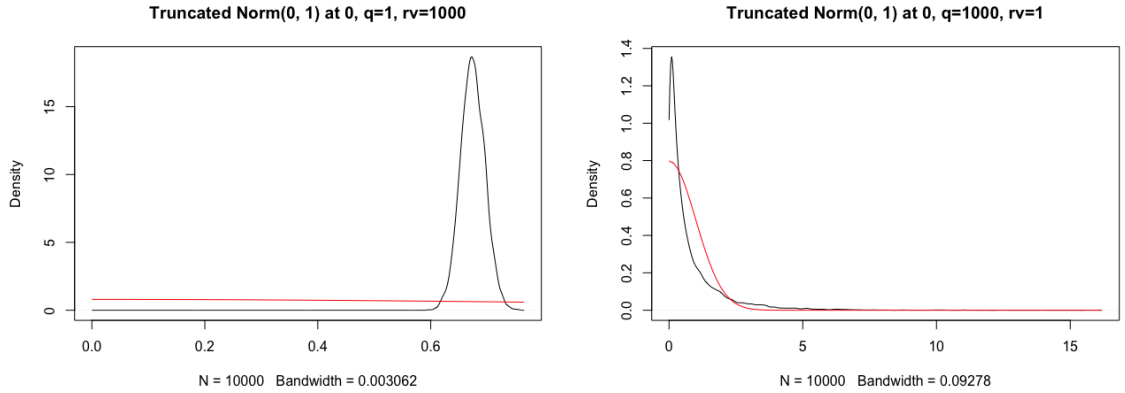
But it is possible to tend towards the accuracy of an MoS approximation of a system by knowing whether the variance is high or low, relatively to the mean.

If we have a relatively low variance `target_var` as compared to `target_mean`, we noticed that we want a greater number of `rv` than `q` to get better accuracy in the method of stages approximation.

Analysing the compactly supported uniform distribution's case, we see the lower variance of that system favored having higher number `rv` of stages represented by Erlang distribution holding times, and a relatively lower number `q` of quantiles.

However, if we have a higher variance `target_var` compared to `target_mean`, it's more favorable to have `q` $>$ `rv`. We see this is true for the folded normal distribution with `mu = 0` and `var = 1`:

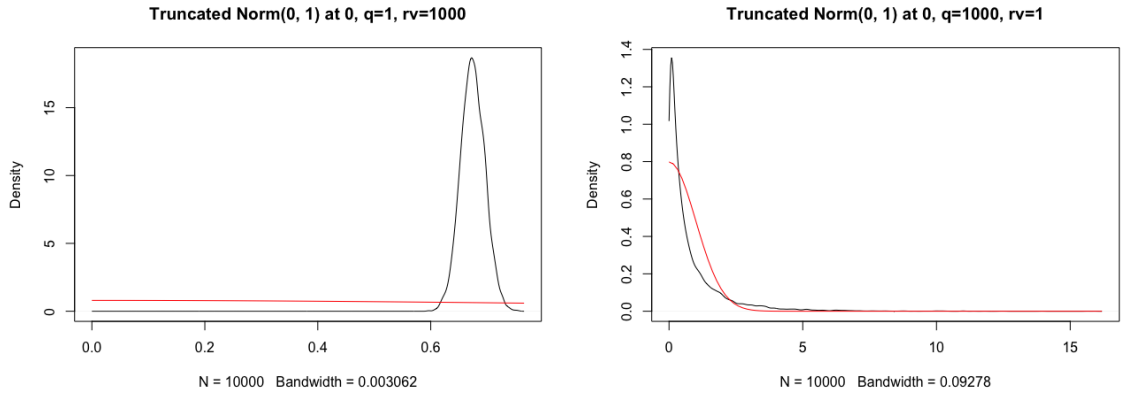
Figure 2.6: Comparisons of the original distribution in red; MoS approximation in blue.

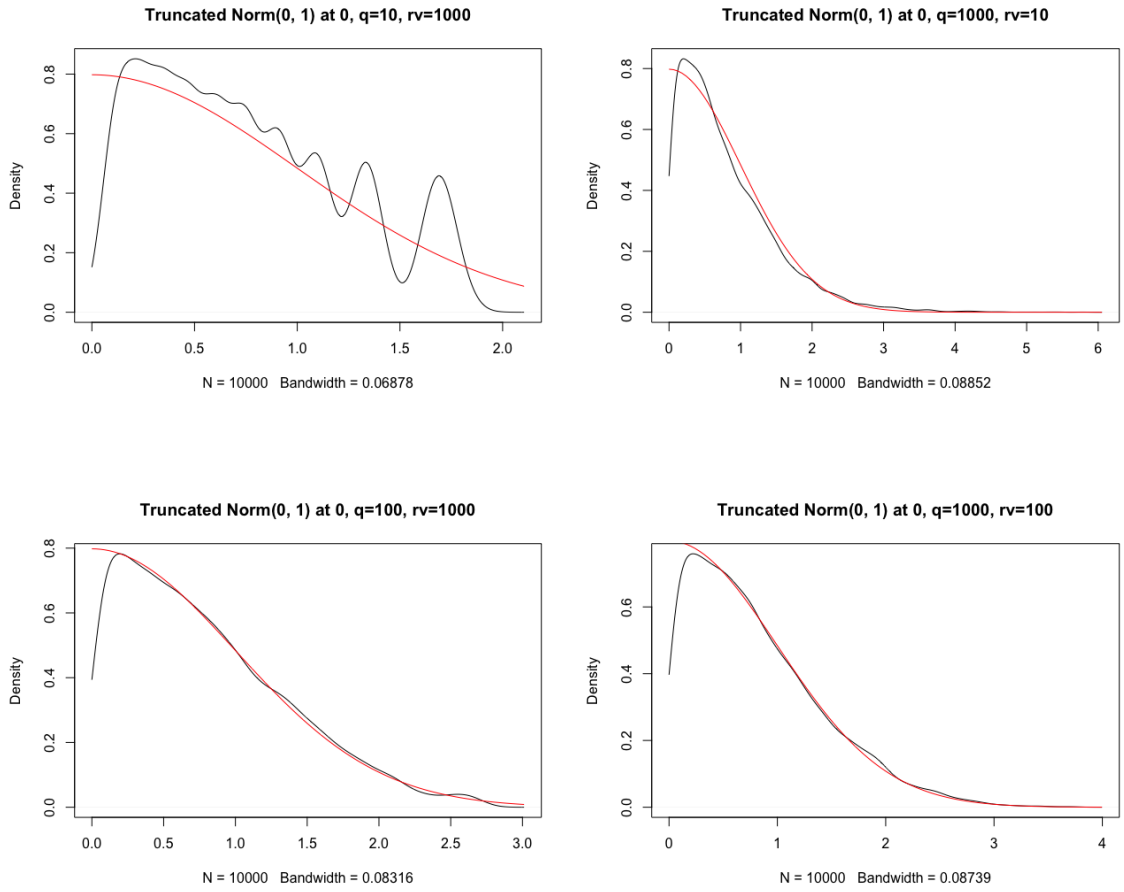


The curve of the density is wildly inaccurate for when $(\texttt{q}, \texttt{rv}) = (1, 1000)$ from the start, as opposed to when $(\texttt{q}, \texttt{rv}) = (1000, 1)$ which looks fairly accurate by comparison. Thus, we may hypothesize that in order to optimize the accuracy of the MoS approximation system, that we should have a much higher value of `q` than `rv` here.

Here is a sequence of figures comparing different values of `q` and `rv`:

Figure 2.7: Comparing MoS approximations with various `q` and `rv` to the original folded normal distribution.





Of course, we notice that once we hit a certain threshold for a fixed `rv = 1000` and `q = 1000`, namely somewhere around `q = 100` and `rv = 100` (resp.), we have a higher accuracy. However, there is still a non-negligible difference in the density function approximation for the $(\texttt{q}, \texttt{rv}) = (100, 1000)$, as compared to its adjacent $(1000, 100)$ graph.

This is, in flavor, similar to the *bias-variance tradeoff*, but it is not quite the same. In fact. it seems that a more simple phenomenon is at play: If the variance is larger, the choice to partition the graph at larger number `q` equidistant points along the support interval gives us a greater accuracy on the Riemann sums notion. However, the variance being smaller means that we may get away with having a smaller `q`, because the closer to constant the graph of the density function is, the less refined a Riemann sum needs to be to approach the value of the Riemannian integral, amounting to the cdf, roughly speaking. After the Riemann sums level of inspecting these systems, we use principles of an analogous result to that of the *Stone-Weierstraß theorem*'s related to i.i.d. exponentials as a sum comprising an Erlang distribution will give that the "finer tuning" choices of approximating small partitions of the original given distribution. The more terms of i.i.d. exponential distributions comprising the Erlang distribution, the more mild and smooth the distribution will likely appear.

Thus, having higher values of `rv` favors distributions with lower variance.

2.4 Application in general system

The general system has n states, its embedded transition matrix is P ($P(i,j)$ is the probability that once the holding time at state i is finished, we then jump to state j.), the holding time is a non-exponential random variable, here we only consider uniform distribution and truncated normal distribution.

2.4.1 Our MOS functions

We generated a two MoS functions for our general system, function `mos_uniform()` for the system with uniform distributed holding time, and `mos_tnormal()` for the system with truncated normal distributed holding time.

Input for function `mos_uniform()` are: the embedded transition matrix P, the array of the left bounds for uniform distributed holding time at all states `a_vec`, the array of the right bounds for uniform distributed holding time at all states `b_vec`, the number of quantiles `q_num`, and the number of exponential random variables `rv_num`.

Input for function `mos_tnormal()` are: the embedded transition matrix P, the array of the mean for normal distributed holding time at all states `mean_vec`, the array of the standard deviation for normal distributed holding time at all states `sd_vec`, the number of quantiles `q_num`, and the number of exponential random variables `rv_num`.

The output of both functions is the pi vector, our experiments are all based on these two functions.

2.4.2 Decreased pi vector `pi_vec` Accuracy when Holding Time Distributions Have Low Variance

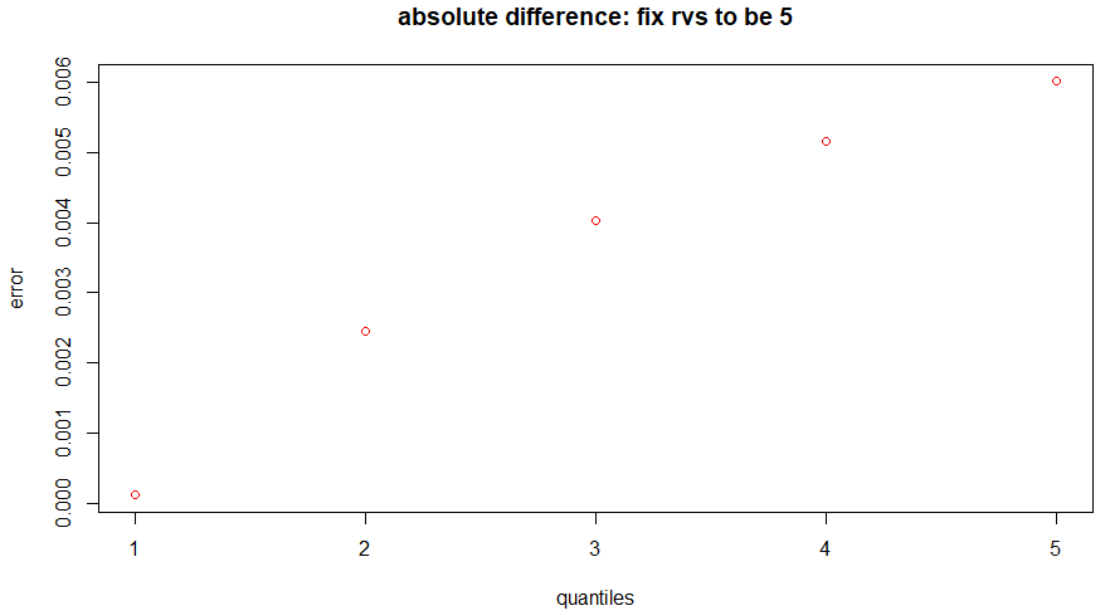
In approximating a single random variable, the more quantiles and exponential random variables we have, the more our result is close to the actual distribution. We observed in our experiments that as we increase the number of exponential random variables, the `pi_vec` obtained by the MoS always gets closer to the `pi_vec` obtained by simulation (we define closeness `d` to be the sum of absolute difference between each entry $\sum_{i=1}^{n} abs(Pi_{1i} - Pi_{2i})$). We ran simulations until the total holding time reaches $6 * 10^7$. Thus we can be confident that the simulation `pi_vec` is extremely close to the actual `pi_vec`. In most cases, increased the number of quantiles lowers `d`. But in the following case we found a counter-intuitive result. We took a transitional probability matrix

$$P = \begin{pmatrix} 0.0 & 0.2 & 0.3 & 0.1 & 0.4 \\ 0.5 & 0.0 & 0.0 & 0.3 & 0.2 \\ 0.3 & 0.4 & 0.0 & 0.3 & 0.0 \\ 0.0 & 0.1 & 0.7 & 0.0 & 0.2 \\ 0.3 & 0.0 & 0.2 & 0.5 & 0.0 \end{pmatrix}$$

We use P and some given holding time distributions to generate Q and then `pi_vec` using the method of stages. For uniform holding times. Let each state has a holding time with parameter (a_i,

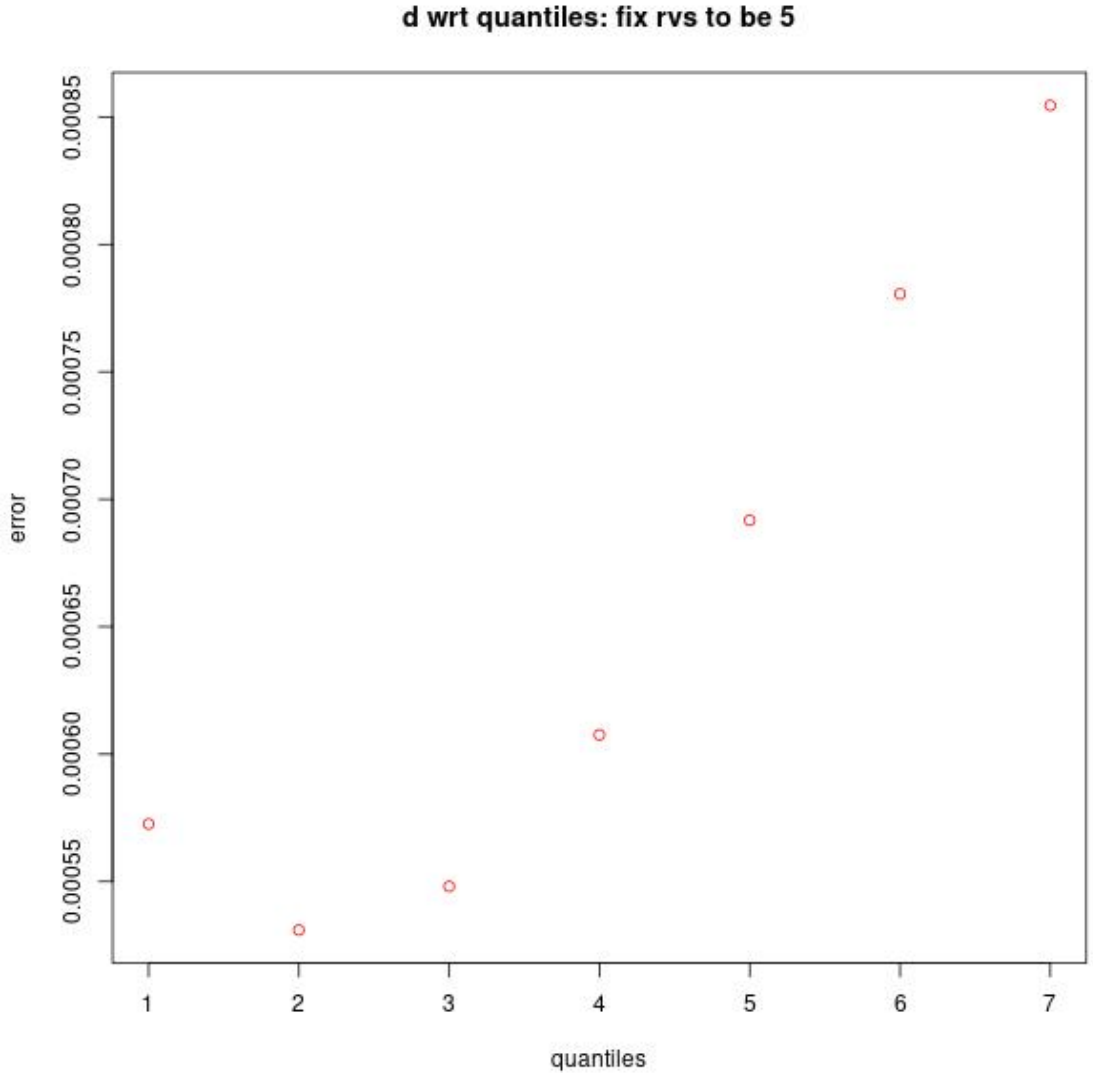
b_i). We take A to be a vector that contains the first parameter a_i for each state and B contains the second parameter b_i. We first choose distributions with $A = (1, 0, 2, 0, 1)$, $B = (1.4, 0.4, 2.4, 0.4, 1.4)$. When we fixed `rv = 5`, we see `d` increases as quantiles increase.

Figure 2.8: In this cases with uniform distributed holding time, increasing number of quantiles results in increasing error



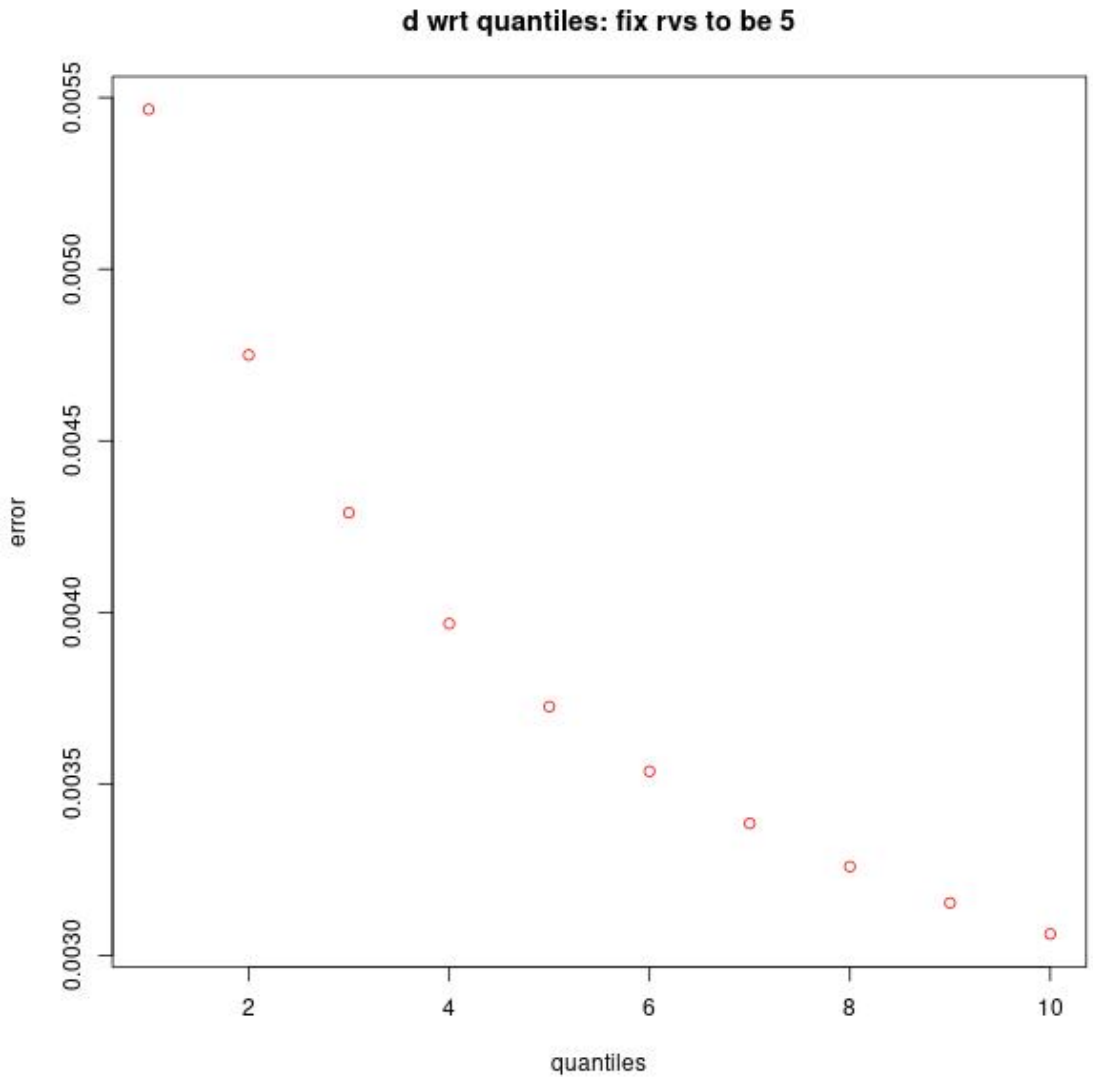
The increase in the number of quantiles actually worsen our `pi_vec` estimation, this is surprising. When we look at the truncated normal holding time. Let each normal distribution have parameter (μ_i, σ_i)M be a vector that contains the μ_i for each state and SD contains σ_i. Each distribution is truncated at $x = 0$.
The same situation occurs when we have SD $= (0.5, 0.5, 0.5, 0.5, 0.5)$.

Figure 2.9: The strange situation not only happens to the normal distributed holding time but for truncated normal holding time as well



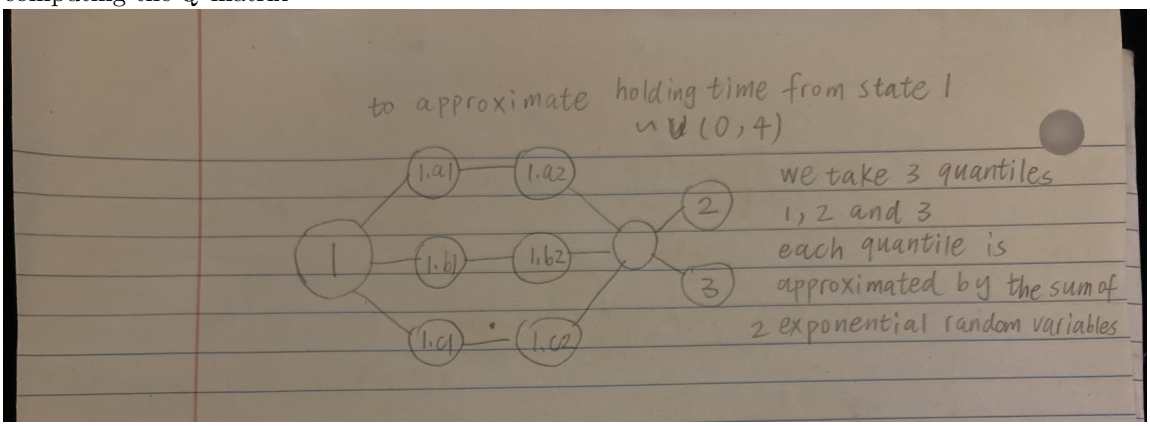
In some other experiments, like when SD $= (0.5, 1.5, 2.5, 1.5, 0.5)$, we saw that the error decreases as quantiles increases.

Figure 2.10: The error decreases when the number of quantiles increases, this matches the intuition



This is what one would expect. As increasing the number of quantiles help us to approximate the holding time better. Then we tried changing the parameters and found that when the mean of all standard deviations grow pass a certain point, the increase in quantiles will always result in the decrease of d. So why is more quantiles undesirable when standard deviations for holding time distributions are small? We reviewed our model, specifically the part where Q matrix is calculated. We noticed each time we shift from original states to the heads of each branch of fictitious sub-states, we lose some information when we add more quantiles. We demonstrate this by an example.

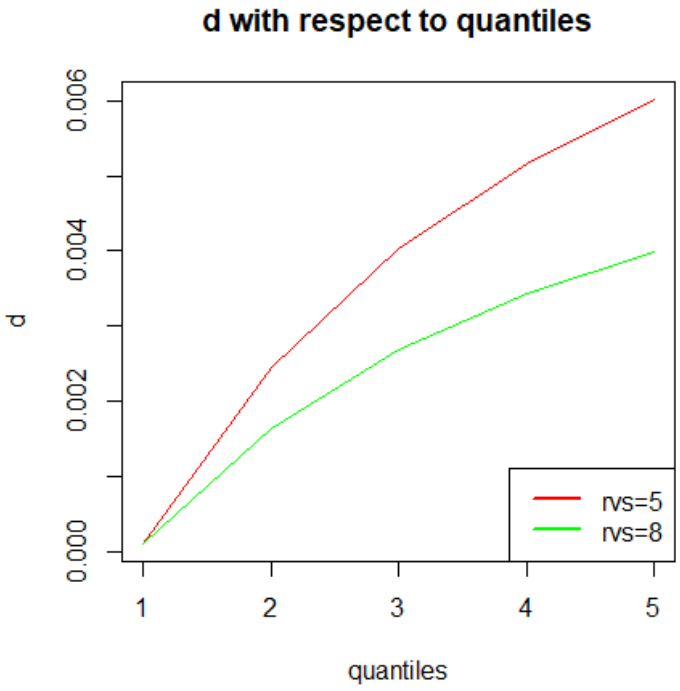
Figure 2.11: demonstration of how taking more quantiles is with the cost of information loss when computing the Q matrix

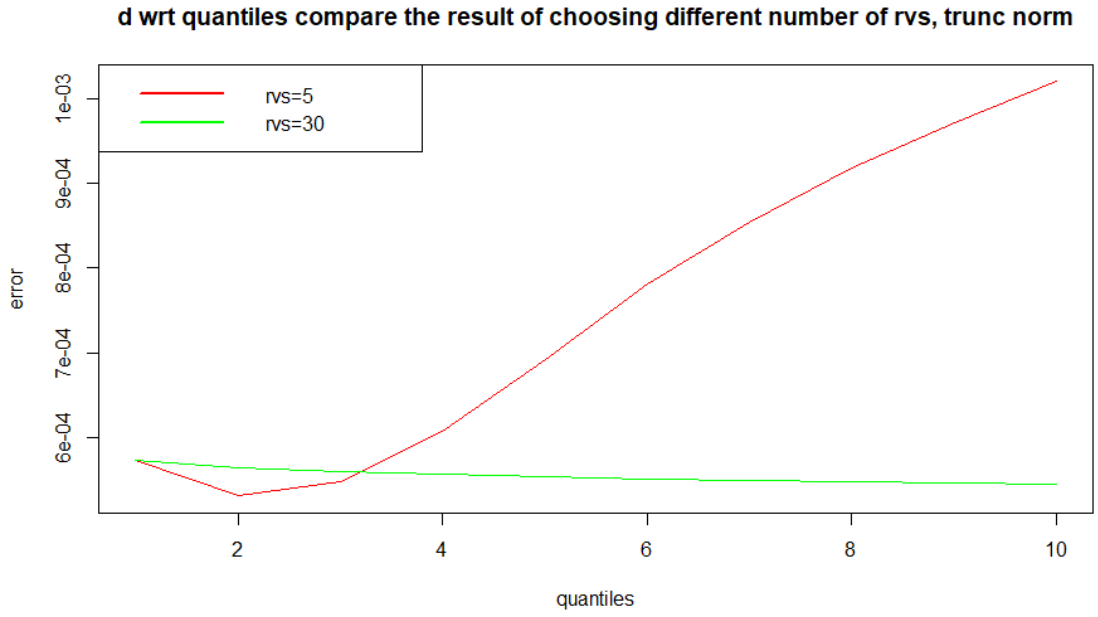


The mean time elapsed from state 1 to state 1.a1 is 0.5, from 1 to 1.b1 is 1 and from 1 to 1.c1 is 1.5. The mean holding time is different for choosing branch a or b or c. However, as we compute the entries correspond to the transitions: $Q_{1.a1,1}$, $Q_{1.b1,1}$ and $Q_{1.c1,1}$ each equals to $\lambda_1 P$(choose branch i). Since we choose each path with the same probability, these entries all equals to each other. We thus fail to catch the difference in holding time when choosing different branches.

If such assumption is true, we would see an increasing number of exponential random variables mitigating or even overturning the negative effect of increasing the number of quantiles. Because as we have more random variables, the difference in the holding time from the original state 1 to each branch of path will be smaller.

Figure 2.12: Increasing the number of random variables for the uniform holding time example mitigated the pattern, in the truncated normal case, it reverted the pattern.





Examining the plots above. The information supports our hypothesis. For the uniform holding time, the increase of random variables mitigates the negative effect of increasing the quantiles while in the truncated normal case it overturns the effect. This is likely because the benefit of describing the distribution more accurately through taking more quantiles outweights the loss of information when taking the Q matrix.

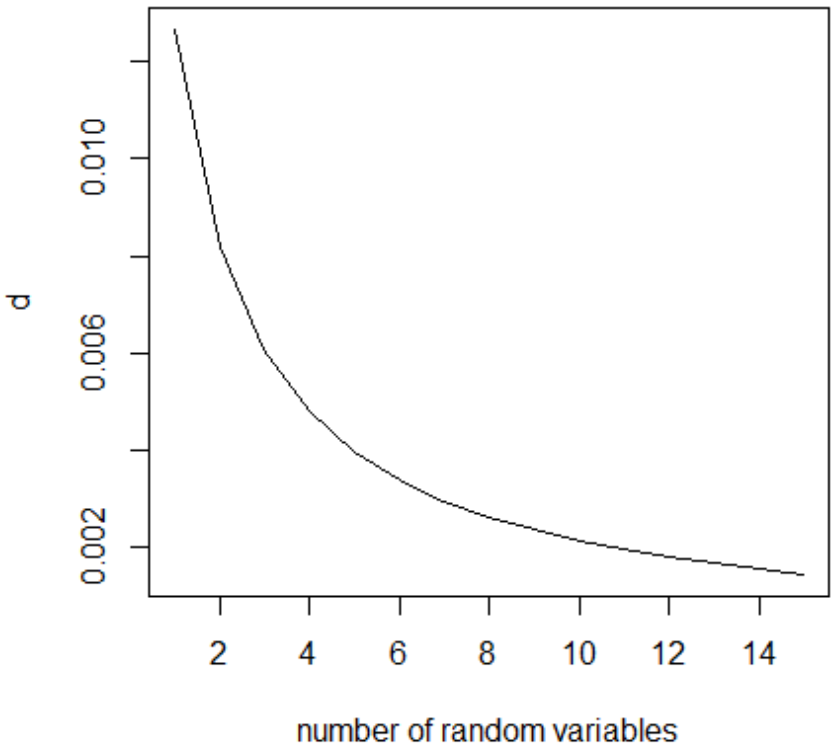
2.5 Accuracy

Base on experiments, we say the MoS model is a good system to investigate the long term behavior of systems that does not have exponential holding times. In all of the plots we have shown in previous sections, even with minimal number of quantiles and number of random variables, the errors d are significantly small (i.e. $<< 0.01$). This is the case for almost all of our experiments.
One might have a concern that when the system grows larger (i.e. the number of states in the original system gets bigger). The error may become a problem. We tested the error for system with 3 states with roughly the same system with 5 states. Here "roughly the same system with different number of states" is that the mean of the means for the three distributions is the same as the mean of the means for the five distributions. Same is the case for the standard deviations. It turns out that if we increase the number of states, d will increase in most cases, but only a small amount. The result is on table2 below. So we conclude that if we increase the size of system, the error will still be acceptable.

	3 states	5 states
Error for uniform distributed holding time system1	0.001283455	0.001584502
Error for truncated normal distributed holding time system1	0.0108545	0.009328718
Error for uniform distributed holding time system2	0.001124415	0.001995297
Error for truncated normal distributed holding time system 2	0.003015755	0.004793288

Nevertheless, there are some cases when we need to be cautious. We observed that when the mean holding times for some states increase, d can increase a fair amount. This may be due to two reasons: 1. longer holding times decrease the accuracy of simulation. 2. larger holding time results more information loss when computing the Q matrix, as discussed in the earlier section. If we increase the number of random variables, we can mitigate this issue. For example, if we have $A = (1.2, 1.5, 1.8)$ and $B = (8, 9, 8)$, when we use one random variable, d will exceed 0.01, but when we increase the number of random variables, d becomes much smaller.

Figure 2.13: Increasing the number of random variables helps to reduce the error when we have longer holding time



2.6 Feasibility

In our MoS model, when the original system has n states, take q number of quantiles for each holding time and using rv number of random variables to approximate each quantile, we will generate a total number of $n + n \times q \times rvs$ states. Solving for the `pi_vec` is an $O(m^3)$ operation, therefore the computational complexity for our system is $O((n \times q \times rvs)^3)$. This number can become scary when one of the three factors increases. However, most improvements happen in the first a few increases on number of quantiles or random variables(partly because the distributions we took are nice). So we don't need to make them big. The complexity will be a problem when we have too many states in the system.

Chapter 3

Contributions

Contributions listed alphabetically wherever they apply

Chapter 1:

Section 1.1: Written by Qizhou Fang as well as the findQ and simulation function coding, Wenru hand calculated an example pi vector to verify the answer

Section 1.2: Written by Qizhou Fang

Section 1.3: Written by Qizhou Fang, Wenru generated the plots

Chapter 2:

Section 2.1: Written by Jane Park

Section 2.2: Written by Yun Qin; editted/formatted by Jane Park

Subsection 2.2.1: Written by Yun Qin; edited/formatted by Jane Park

Subsection 2.2.2: Written by Yun Qin; edited/formatted by Jane Park

Subsection 2.2.3: Written by Qizhou Fang, Yun Qin; edited/formatted by Jane Park

Section 2.3: Written by Yun Qin; editted/formatted by Jane Park

Subsection 2.3.1: Written by Yun Qin; editted/formatted by Jane Park

Subsection 2.3.2: Written by Jane Park; Figures generated by Yun Qin

Section 2.4: Written by Yun Qin; editted/formatted by Jane Park

Subsection 2.4.1: Written by Yun Qin; editted/formatted by Jane Park

Subsection 2.4.2: Written by Qizhou Fang, findQ coding and simulation by Qizhou Fang

Section 2.5: Written by Qizhou Fang; `insertgraphics` formatted by Jane Park

Section 2.6: Written by Qizhou Fang; `insertgraphics` formatted by Jane Park

Appendix A

Problem A Code

```
#this is the code that generates the optimal choice of m,n table

#mean_waititng(m,n,u,v,pi_vec)
#rejection_prob(m,n,p,pi_vec)
# find out the best combinations of m and n for each raito of u,v
n_vec=numeric(0)
m_vec=numeric(0)
mw_vec=numeric(0)
rj_vec=numeric(0)
for (u in seq(0.2,5,0.4))
{
  best_m=-5
  least_nurse=20
  good_mw=-1
  good_rj=-1
  for (n in ceiling(u):10)
  {
    for (m in 0:5)
    {
      pi_vec=call_system(m,n,u,1,1)
      mw=mean_waititng(m,n,u,1,pi_vec)
      rj=rejection_prob(m,n,1,pi_vec)
      if (mw<(u/2)&&rj<0.01)
      {
        if (n<least_nurse)
        {
          best_m=m
          least_nurse=n
          good_mw=mw
          good_rj=rj
        }
```

```
      }
    }
  }
  n_vec=c(n_vec,least_nurse)
  m_vec=c(m_vec,best_m)
  mw_vec=c(mw_vec,good_mw)
  rj_vec=c(rj_vec,good_rj)
}
x=seq(0.2,5,0.4)
plot(x,n_vec)
plot(x,m_vec)
# find how scaling the parameters affects the metrics
m=2
n=3
p=1
u=2
v=1
lst1=rep(0,10)
lst2=rep(0,10)

x1=c(1:10)
for (i in 1:10)
{
  pi_vec=call_system(m,n,i*u,i*v,p)
  mw=mean_waititng(m,n,i*u,i*v,pi_vec)
  rj=rejection_prob(m,n,p,pi_vec)
  lst1[i]=mw
  lst2[i]=rj
  mw_vec[i]=mw
}
plot(x1,lst1,main="the effect to mean waiting time when we scale u and v, m=2,n=3, u/v=2",
xlab="v",ylab="mean waiting time")
plot(x1,lst2,main="the effect to rejection probability when we scale u and v,m=2, n=3,
u/v=2",xlab="v",ylab="rejection probability")
```

```
# this is the function that computes the pi vector

# f(i,j)=1+j+i*(n+1)
findQ=function(m,n,u,v,p)
{
  convert=function(i,j)
  {
    return(1+j+i*(n+1))
  }
  # matrix storing lamb_ij_kl
  M=matrix(0,(m+n+1)*(n+1),(m+n+1)*(n+1))
  # vector storing lamb_i
  lamb_vec=rep(0,(m+n+1)*(n+1))


  for (i in 0:(m+n))
  {
    for (j in 0:n)
    {
      if (i-j<=m)
      {
        up=0
        down=0
        #each case we can go up or down
        #we treat each case separately
        queue=max(i-j,0)
        meet=min(i,j)

        # more people or nurses
        # regular mu_ij=1/v
        if (queue==m)
        {
          if (j==n)
          {
            # can't go up
          }
          else
          {
            # can go up to i+1,j+1
            M[1+j+i*(n+1),1+j+1+(i+1)*(n+1)]=p/v
            up=p/v
          }

        }
        # the queue is not full, can add one more person
        else
        {
          M[1+j+i*(n+1),1+j+(i+1)*(n+1)]=1/v
```

```
      up=1/v
    }
    # less people or nurses
    # regular d_ij=min(i,j)/u

    # if there are people it's possible to go down
    if (i>0)
    {
      # if no one in queue


      # if there are nurses
      if(j>0)
      {
        if (queue==0)
        {
          # reduce one nurse and patient
          M[1+j+i*(n+1),j+(i-1)*(n+1)]=meet/u
          down=meet/u
        }
        # if there are people in queue, reduce patient by 1
        else
        {
          M[1+j+i*(n+1),1+j+(i-1)*(n+1)]=meet/u
          down=meet/u
        }
      }
      # if no nurses, can't go down
      else
      {

      }
    }
    # no person, can't go down
    else
    {

    }
  }
  # illegal states, ignore them
  else
  {

  }

  # update lamb_ij
  lamb_vec[1+j+i*(n+1)]=up+down
}
```

```
}

# now we have the matrix containing lamb_ij_kl
# and the vector containing lamb_ij
# now try to find Q using M and lamb_vec
Q=matrix(0,(m+n+1)*(n+1),(m+n+1)*(n+1))
for (k in 1:((m+n+1)*(n+1)))
{
  for (l in 1:((m+n+1)*(n+1)))
  {
    if (k==l)
    {

      Q[k,l]=-lamb_vec[k]
    }
    else
    {

      Q[k,l]=M[l,k]
    }
  }
}

#print(lamb_vec)
# deletion of impossible states
# deletion of transient states
count=0
for (i in 0:(m+n))
{
  for (j in 0:n)
  {
    if (i-j>m )
    {
      Q=Q[-(convert(i,j)-count),-(convert(i,j)-count)]

      count=count+1
    }
    else if(j>i)
    {
      Q=Q[-(convert(i,j)-count),-(convert(i,j)-count)]

      count=count+1
    }
  }
}
#print(t(M))

return(Q)
```

```
}
find_pi=function(Q)
{
  num=nrow(Q)
  Q[num,]=rep(1,num)
  b=c(rep(0,num-1),1)
  #b=rbind(b)
  #b=t(b)
  #pi_vec=solve(Q,b)
  #pi_vec=inv(Q)%*%b
  pi_vec=solve(Q,b)
  return(pi_vec)
}

call_system=function(m,n,u,v,p)
{
  Q=findQ(m,n,u,v,p)
  pi_vec=find_pi(Q)
  for (i in 0:(m+n))
  {
    for (j in 0:n)
    {
      if (i-j>m)
      {
        pi_vec=append(pi_vec,0,j+i*(n+1))
      }
      else if(j>i)
      {
        pi_vec=append(pi_vec,0,j+i*(n+1))
      }
    }
  }
  return(pi_vec)
}
queue_vec=function(m,n,pi_vec)
{
  q_vec=rep(0,m+1)
  for (i in 0:(m+n))
  {
    for (j in 0:n)
    {
      queue=max(i-j,0)
      if (i-j<=m)
      {
        q_vec[queue+1]=q_vec[queue+1]+pi_vec[j+1+i*(n+1)]
      }
    }
  }
```

```
  return(q_vec)
}
meeting_vec=function(m,n,pi_vec)
{
  m_vec=rep(0,n+1)
  for (i in 0:(m+n))
  {
    for (j in 0:n)
    {
      meet=min(i,j)
      if (i-j<=m)
      {
        m_vec[meet+1]=m_vec[meet+1]+pi_vec[j+1+i*(n+1)]
      }
    }
  }
  return(m_vec)
}
people_vec=function(m,n,pi_vec)
{
  p_vec=rep(0,m+n+1)
  for (i in 0:(m+n))
  {
    for (j in 0:n)
    {
      if (i-j<=m)
      {
        p_vec[i+1]=p_vec[i+1]+pi_vec[j+1+i*(n+1)]
      }
    }
  }
  return(p_vec)
}
nurse_vec=function(m,n,pi_vec)
{
  n_vec=rep(0,n+1)
  for (i in 0:(m+n))
  {
    for (j in 0:n)
    {
      if (i-j<=m)
      {
        n_vec[j+1]=n_vec[j+1]+pi_vec[j+1+i*(n+1)]
      }
    }
  }
  return(n_vec)
}
```

```
queue_length=function(m,n,pi_vec)
{
  l=0
  queue_vec=queue_vec(m,n,pi_vec)
  for (i in 0:m)
  {
    l=l+i*queue_vec[i+1]
  }
  return(l)
}
num_people=function(m,n,pi_vec)
{
  l=0
  people_vec=people_vec(m,n,pi_vec)
  for (i in 0:(m+n))
  {
    l=l+i*people_vec[i+1]
  }
  return(l)
}
num_nurse=function(m,n,pi_vec)
{
  l=0
  nurse_vec=nurse_vec(m,n,pi_vec)
  for (i in 0:n)
  {
    l=l+i*nurse_vec[i+1]
  }
  return(l)
}
num_meeting=function(m,n,pi_vec)
{
  l=0
  meeting_vec=meeting_vec(m,n,pi_vec)
  for (i in 0:n)
  {
    l=l+i*meeting_vec[i+1]
  }
  return(l)
}
utility_rate=function(m,n,pi_vec)
{
  rate=num_meeting(m,n,pi_vec)/num_nurse(m,n,pi_vec)
  return(rate)
}
mean_waititng=function(m,n,u,v,pi_vec)
{
  e_queue=queue_length(m,n,pi_vec)
```

```
  e_meeting=num_meeting(m,n,pi_vec)
  handling_t=u/e_meeting
  e_wait=e_queue*handling_t
  return(e_wait)
}
mean_residence=function(m,n,u,v,pi_vec)
{
  wt=mean_waititng(m,n,u,v,pi_vec)
  e_res=wt+u
  return(e_res)
}
rejection_prob=function(m,n,p,pi_vec)
{
  prob=0
  for (i in (0:m+n))
  {
    for (j in 0:n)
    {
      if (i-j==m)
      {
        if (j==n)
        {
          prob=prob+pi_vec[1+j+i*(n+1)]
        }
        else
        {
          prob=prob+pi_vec[1+j+i*(n+1)]*(1-p)
        }
      }
    }
  }
  return(prob)
}
# conditional waiting time
wt_notice=function()
{

}
m=2
n=3
u=3
v=2
p=1
pi_vec=call_system(m,n,u,v,p)
#print(queue_length(m,n,pi_vec))
#print(mean_residence(m,n,u,v,pi_vec))
print(mean_waititng(m,n,u,v,pi_vec))
print(rejection_prob(m,n,p,pi_vec))
```

```
# this is the simulation code for problem 1




q_simu=function(m,n,u,v,p,duration,k,l)
{
  state_vec=rep(0,(1+m+n)*(1+n))
  # random starting nurses and people
  #j=sample.int(n+1,1)-1
  #i=sample.int(j+m+1,1)-1
  wt=0
  wij=0
  wt_vec=numeric(0)
  if_waiting=FALSE
  i=0
  j=0
  ct=0
  q=max(k-l,0)
  progress=0
  while (ct<duration)
  {
    if (i==k&&j==l)
    {
      if_waiting=TRUE
    }
    # find the queue length
    # find the number of meetings
    if (i<=j)
    {
      queue=0
      meet=i
    }
    else
    {
      queue=i-j
      meet=j
    }
    arrival=rexp(1,1/v)
    if (meet!=0)
    {
      cf=rexp(1,meet/u)
    }
    else
    {
      cf=Inf
    }
    ht=min(cf,arrival)
```

```
    wt=wt+queue*ht
    state_vec[1+j+i*(n+1)]=state_vec[1+j+i*(n+1)]+ht
    # if it is arrival
    if (arrival<=cf)
    {
      if (queue==m)
      {
        if (j<n)
        {
          num=runif(1)
          # add patient and nurse
          if (num<p)
          {

            i=i+1
            j=j+1

          }
          # reject the patient
          else
          {

          }
        }
        # reject the patient since there is no room
        else if(j==n)
        {

        }
      }
      else
      {
        i=i+1

      }

    }

    # if it is cf
    else
    {
      # reduce 1 nurse if queue is empty
      if (queue==0)
      {

        j=j-1
      }
      else
```

```
      {

      }
      i=i-1
      if (if_waiting==TRUE)
      {
        progress=progress+1
      }

    }
    ct=ct+ht
    if (if_waiting==TRUE)
    {
      wij=wij+ht
    }
    if (progress==q)
    {
      wt_vec=c(wt_vec,wij)
      wij=0
      progress=0
      if_waiting=FALSE
    }
  }
  return(sum(wt_vec)/length(wt_vec))
  #return(state_vec)

}
m=3
n=3
u=2
v=1
p=1
duration=80000
print(q_simu(m,n,u,v,p,duration,3,1))
#jpeg("plt1.jpg",width=600,height=600)
#plot(x,err_vec,main="",xlab="m",ylab="difference between simulation and waiting time function")
#lines(x,r2,type="l",col="green")
#legend("topleft",legend=c("rvs=5","rvs=30"),col=c("red","green"),lwd=2)
#dev.off()
```

Appendix B

Problem B Code

B.1 Uniform

```
library(ggplot2)
library(ggpubr)
# Define the number of trials
trials <- 10000
# approximate U(1,2) using MOS
a <- 1
b <- 2
# Denote rv as the number of exponential rvs,rv>=1
rv <- 100
# Denote q as the number of quantiles, q>=1
q <- 100
# Mean and Variance of this uniform dist
target_mean <- 1.5
target_var <- 1/12
# -------Simulation on states with equal means-------
approx_equal <- function(){
  list1 <- numeric(0)
  for (i in 1:trials)
  {
    prob <- runif(1)
    prob_arr <- seq(0, 1, by=(1/q))
    idx <- findInterval(prob, prob_arr)
    lambda <- a + idx*((b-a)/(q+1))
    approx <- sum(rexp(rv, rv/lambda))
    list1 <- c(list1, approx)
  }
  return(list1)
}
```

```r
# Plot
start_time1 <- Sys.time()
list1 <- approx_equal()
end_time1 <- Sys.time()
list1_data <- data.frame(t = 1:trials, points = list1)
plot1 <- ggplot(data = list1_data, aes(x = t, y = points)) + geom_point()
plot1 + ggtitle("Equal lambda") + xlab("Trials") + ylab("Points")
# Mean
mean1 <- mean(list1)
# Variance
var1 <- var(list1)
# -------Simulation on states with random means (sum is the quantile)-------
approx_random <- function(){
  list2=numeric(0)
  for (i in 1:trials)
  {
    prob <- runif(1)
    quantile_arr <- seq(0, 1, by=(1/q))
    idx <- findInterval(prob, quantile_arr)
    lambda <- a + idx*((b-a)/(q+1))
    # generate m random means whose sum is lambda
    mean_arr <- runif(rv, min=1, max=rv)
    mean_arr <- lambda*mean_arr/sum(mean_arr)
    approx_arr <- numeric(0)
    for(j in 1:rv){
      approx_arr <- c(approx_arr, rexp(1, 1/mean_arr[j]))
    }
    approx <- sum(approx_arr)
    list2 <- c(list2, approx)
  }
  return(list2)
}
# Plot
start_time2 <- Sys.time()
list2 <- approx_random()
end_time2 <- Sys.time()
list2_data <- data.frame(t = 1:trials, points = list2)
plot2 <- ggplot(data = list2_data, aes(x = t, y = points)) + geom_point()
plot2 + ggtitle("Random lambda") + xlab("Trials") + ylab("Points")
# Mean
mean2 <- mean(list2)
# Variance
var2 <- var(list2)
# Save two plots
#ggarrange(plot1, plot2, labels = c("Equal lambda", "Random lambda"), ncol = 2, nrow = 1)%>%
#  ggexport(filename = "/Users/pmh/Yun/ECS256_probability/project/uniform_dist_simu.png")
#-----------Compare------------
print(end_time1 - start_time1)
```

```
print(end_time2 - start_time2)
print(mean1)
print(mean2)
print(abs(mean1-target_mean))
print(abs(mean2-target_mean))
print(var1)
print(var2)
print(abs(var1-target_var))
print(abs(var2-target_var))
```

B.2 Truncated Normal

```
# PLEASE UNCOMMENT THE LINE BELOW IF FIRST RUN THIS SCRIPT
# install.packages('truncnorm')
library('truncnorm')
library(ggplot2)
library(ggpubr)
# Define the number of trials
trials <- 10000
# approximate truncated Norm(0, 1) using MOS
mu <- 0
sigma <- 1
# Denote rv as the number of exponential rvs, rv>=1
rv <- 100
# Denote q as the number of quantiles, q>=1
q <- 100
# Mean and Variance of this uniform dist
target_mean <- etruncnorm(a=0, b=Inf, mean=mu, sd=sigma)
target_var <- vtruncnorm(a=0, b=Inf, mean=mu, sd=sigma)
# -------Simulation on states with equal means-------
approx_equal <- function(){
  list <- numeric(0)
  for (i in 1:trials)
  {
    prob <- runif(1)
    prob_arr <- seq(0, 1, by=(1/q))
    idx <- findInterval(prob, prob_arr)
    lambda <- qtruncnorm(idx*(1/(q+1)), a=0, b=Inf, mean=mu, sd=sigma)
    approx <- sum(rexp(rv, rv/lambda))
    list <- c(list, approx)
  }
  return(list)
}
# Plot
start_time <- Sys.time()
```

```
list <- approx_equal()
end_time <- Sys.time()
list_data <- data.frame(value=list)
plot1 <- ggplot(data = list_data, aes(x = value)) + geom_histogram(fill="#69b3a2", color="#e9ecef")
plot1 + ggtitle("10000 Trials for Norm(0,1) Truncated at 0") + xlab("Values") + ylab("Trials")
#ggarrange(plot1, ncol = 1, nrow = 1)%>%
#  ggexport(filename = "/Users/pmh/Yun/ECS256_probability/project/norm_dist_simu.png")
# Results
mean <- mean(list)
var <- var(list)
print(end_time - start_time)
print(mean)
print(abs(mean-target_mean))
print(var)
print(abs(var-target_var))

# density plot
plot2 <- plot(density(list,from=0), main="Truncated Norm(0, 1) at 0, q=100, rv=100")
curve(dtruncnorm(x, a=0,b=Inf,mean=0,sd=1), 0, add = TRUE, col = "red")
```

```
# this code is for pi vector computation for problem 2

library(truncnorm)
mos_Q=function(P,quant,e_vars)
{


  #number of states
  n=nrow(P)

  #number of branches
  branches=ncol(quant)
  # calculate the possible num of transitions from each state
  adj_l=list()
  count=0
  # the state number of each ori state
  state_num=numeric(0)

  for (i in 1:n)
  {
    count=count+1
    possible_vec=numeric(0)
    state_num=c(state_num,count)
    for (j in 1:n)
    {
      # construct the list of lists containing the states that can be reached
```

```
    # from state i
    if (P[i,j]>0)
    {
      possible_vec=c(possible_vec,j)
    }
  }
  adj_l=append(adj_l,list(possible_vec))
  count=count+e_vars*branches
}
#print(adj_l)
q_size=count
Q=matrix(0,nrow=q_size,ncol=q_size)
#print(q_size)
#print(state_num)
# state conversion to one dimension function
convert=function(i,j,k)
{
  return(state_num[i]+(j-1)*e_vars+k)
}

# fill in the transitional probabilities
for (i in 1:n)
{
  substates=branches*e_vars
  lamb_list=numeric(0)
  # compute lambda convert(i)
  for (j in 1:branches)
  {
    lambda=(e_vars+1)/quant[i,j]
    lamb_list=c(lamb_list,lambda)
  }
  aver=sum(lamb_list)/branches
  # -lamb i,i
  Q[state_num[i],state_num[i]]=-aver
  for (j in 1:branches)
  {
    Q[state_num[i],convert(i,j,1)]=aver/branches
    for (k in 1:e_vars)
    {
      Q[convert(i,j,k),convert(i,j,k)]=-lamb_list[j]
      if(k!=e_vars)
      {
        Q[convert(i,j,k),convert(i,j,k+1)]=lamb_list[j]

      }
      else
      {
        for (l in adj_l[[i]])
```

```
          {

            Q[convert(i,j,k),state_num[l]]=P[i,l]*lamb_list[j]
          }
        }
      }
    }
  }
  # transpose Q
  Q=t(Q)
  return(Q)

}
# compute pi vector
find_pi=function(Q)
{
  num=nrow(Q)
  Q[num,]=rep(1,num)
  b=c(rep(0,num-1),1)
  #b=rbind(b)
  #b=t(b)
  #pi_vec=inv(Q)%*%b
  pi_vec=solve(Q,b)
  return(pi_vec)
}
# to get back the pi vector with original states

mos_pi=function(P,quant,e_vars)
{
  Q=mos_Q(P,quant,e_vars)
  pi_1=find_pi(Q)
  states=nrow(P)
  pi_vec=rep(0,states)
  branches=ncol(quant)
  for (i in 1:states)
  {
    for (j in (i+(i-1)*e_vars*branches):(i+i*e_vars*branches))
    {
      pi_vec[i]=pi_vec[i]+pi_1[j]
    }
  }
  return(pi_vec)
}
unif_quant=function(a_vec,b_vec,divisions)
{
  delta=1/(divisions+1)
  states=length(a_vec)
  quant=matrix(0,states,divisions)
```

```
  # generate percentage_vec
  percentage_vec=numeric(0)
  for (i in 1:divisions)
  {
    percentage_vec=c(percentage_vec,i*delta)
  }
  # generate quant matrix
  for (i in 1:states)
  {
    # transform interval
    quant[i,]=(percentage_vec*(b_vec[i]-a_vec[i]))+a_vec[i]
  }
  return(quant)
}

truncnorm_quant=function(mean_vec,sd_vec,divisions)
{
  delta=1/(divisions+1)
  states=length(mean_vec)
  quant=matrix(0,states,divisions)
  # generate percentage_vec
  percentage_vec=numeric(0)
  for (i in 1:divisions)
  {
    percentage_vec=c(percentage_vec,i*delta)
  }
  # generate quant matrix
  for (i in 1:states)
  {
    quant[i,]=qtruncnorm(percentage_vec,a=0,b=Inf,mean=mean_vec[i],sd=sd_vec[i])
  }
  return(quant)
}

# input parameters of distribution for ht of each state,
# the transitional probability matrix P
# the number of exp rvs to use, the number of percentages to use
# output pi_vec
mos_uniform=function(P,a_vec,b_vec,quant_num,rv_num)
{
  quant=unif_quant(a_vec,b_vec,quant_num)
  pi_vec=mos_pi(P,quant,rv_num)
  return(pi_vec)
}
mos_tnormal=function(P,mean_vec,sd_vec,quant_num,rv_num)
{
  quant=truncnorm_quant(mean_vec,sd_vec,quant_num)
  pi_vec=mos_pi(P,quant,rv_num)
```

```
  return(pi_vec)
}
r1=c(0.0,0.4,0.6)
r2=c(0.2,0.0,0.8)
r3=c(0.5,0.5,0.0)
P3=rbind(r1,r2,r3)
a3=c(1.2,1.5,1.8)
b3=c(8,9,8)
#for (i in 1:10)
#{
  # change quauntiles number first then rv numbers
  #pi_vec1=mos_tnormal(P,a_vec,b_vec,i,12)
  #pi_vec2=mos_tnormal(P,a_vec,b_vec,i,30)
  #err1_vec=abs(pi_vec1-simu_vec)
  #err2_vec=abs(pi_vec2-simu_vec)
  #err1=sum(err1_vec)
  #err2=sum(err2_vec)
  #r1=c(r1,err1)
  #r2=c(r2,err2)
#}
#x=c(1:10)
#plot(x,r1,main="d wrt quantile",xlab="quantiles",ylab="error",col="red",type="l")
quants=15
#rvs=1

simu_u3=c(0.2521248, 0.3359102, 0.4119650)
e_vec=numeric(0)
x=numeric(0)
for (i in 1:15)
{
  err=sum(abs(mos_uniform(P3,a3,b3,quants,i)-simu_u3))
  e_vec=c(e_vec,err)
  x=c(x,i)
}
plot(x,e_vec,xlab="number of random variables",ylab="d",type="l")
```

```
# this is the simulation code for problem 2

#input P, a_vec, b_vec, max_time
#output simulated pi_vec
#simu
library(truncnorm)
uniform_simu=function(P,a_vec,b_vec,max_time)
{
  states=nrow(P)
  #initialize state i, simu_pi, and time
```

```
t=0
i=sample.int(states,1)
simu_pi=rep(0,states)
#generate cumulative probability matrix
c_matrix=matrix(0,states,states)
for (i in 1:states)
{
  for (j in 1:states)
  {
    if (j==1)
    {
      c_matrix[i,j]=P[i,j]
    }
    else
    {
      c_matrix[i,j]=c_matrix[i,j-1]+P[i,j]
    }
  }
}

while(t<max_time)
{
  # generate random percentage
  num=runif(1)
  # generate random hitting time
  ht=runif(1,a_vec[i],b_vec[i])
  simu_pi[i]=simu_pi[i]+ht
  #see which state it will jump to
  for (j in 1:states)
  {
    if (j==1)
    {
      if((0<num)&&(num<=c_matrix[i,j]))
      {
        tempr=1
      }
    }
    else
    {
      if((c_matrix[i,j-1]<num)&&(num<=c_matrix[i,j]))
      {
        tempr=j
      }
    }
  }
  i=tempr
  t=t+ht
}
```

```
  return(simu_pi/(sum(simu_pi)))
}
tnormal_simu=function(P,mean_vec,sd_vec,max_time)
{
  states=nrow(P)
  #initialize state i, simu_pi, and time
  t=0
  i=sample.int(states,1)
  simu_pi=rep(0,states)
  #generate cumulative probability matrix
  c_matrix=matrix(0,states,states)
  for (i in 1:states)
  {
    for (j in 1:states)
    {
      if (j==1)
      {
        c_matrix[i,j]=P[i,j]
      }
      else
      {
        c_matrix[i,j]=c_matrix[i,j-1]+P[i,j]
      }
    }
  }

  while(t<max_time)
  {
    # generate random percentage
    num=runif(1)
    # generate random hitting time
    ht=rtruncnorm(1,a=0,b=Inf,mean=mean_vec[i],sd=sd_vec[i])
    simu_pi[i]=simu_pi[i]+ht
    #see which state it will jump to
    for (j in 1:states)
    {
      if (j==1)
      {
        if((0<num)&&(num<=c_matrix[i,j]))
        {
          tempr=1
        }
      }
      else
      {
        if((c_matrix[i,j-1]<num)&&(num<=c_matrix[i,j]))
        {
          tempr=j
```

```
        }
      }
    }
    i=tempr
    t=t+ht
  }
  return(simu_pi/(sum(simu_pi)))
}

r1=c(0.0,0.4,0.6)
r2=c(0.2,0.0,0.8)
r3=c(0.5,0.5,0.0)
P3=rbind(r1,r2,r3)
a3=c(1.2,1.5,1.8)
b3=c(8,9,8)
print(uniform_simu(P3,a3,b3,max_time = 120000000))
#0.2197650 0.1171213 0.2686937 0.2006243 0.1937956
#0.2347546, 0.1363880, 0.2609917, 0.1985844, 0.1692813
```