Markov Decision Process in Reentrant Line

By Yi Zhang
yzhang33@gatech.edu
12/31/2010

Abstract

Here we analyze the optimal policy for two machine, three buffer reentrant line, where a particular job went to machine one first, then to machine two, then back to machine one to finish up the process. Each machine can only work on one job at a time. It's apparent that machine one can work on jobs from buffer one or from buffer three. The focus of this analysis is to determine which buffer should machine one be working on for each possible scenario, where a scenario is how many items are in each of the three buffers. One feasible state index system is introduced along with matlab/C++ programs to solve for optimal policy using Markov decision process.

1 Overview

Reentrant line is very common among in wafer fabrication lines. It essentially means one station is able to do multiple types of processes for a specific job sequence. As Figure 1

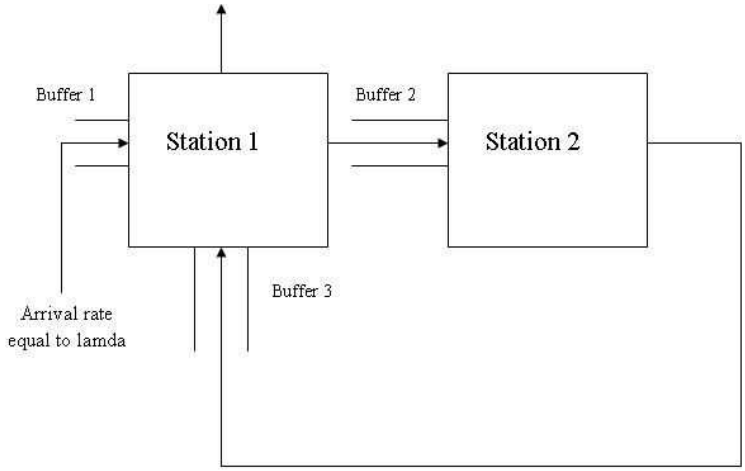


Figure 1: Reentrant line

illustrates, there are two stations processing jobs. Every job enters the system is first being processed by station 1, then station 2, then come back to station 1 for the final process. After that, the job is considered finished and will exit the system. There are buffers in front of every station and each buffer is distinct and specifically assigned to different stages of operation. For example, a job has to enter station 1 twice for initial process and final process, the buffer for initial process is distinct from the buffer for the final process. The arrival for this process is denoted λ per unit time. Each stage of operation has its own exponentially distributed process time with mean m_i where i indicates what stage the job is in. The rate for each process is μ_i, where $\mu_i = \frac{1}{m_i}$. Also assign a cost vector h in which each term h_i is the cost associated with every job waiting in buffer i (including the job that is currently being worked on) per unit time. For example, if h is [10 15 20] per minute for this case, the 10 just means for each job in buffer 1, there is a cost of 10 dollars per minute associated with it. Also, set $Z_i(t)$ as the number of jobs in buffer i (including the job that is currently being worked on) at time t. Thus, the state space for this system Z(t) is equal to $(Z_1(t), Z_2(t), Z_3(t))$

2 Performance Measures and Traffic Intensity

There are two other meaningful statistics that are crucial to better understand this system. First one is average system size. In this case, if T is the time span of the system, average system size can be expressed as

$$\lim_{T\to\infty} \frac{1}{T} \int_0^T (Z_1(t) + Z_2(t) + Z_3(t))\ dt$$

The second statistic is called long run average cost(denoted by γ) meaning in the long run the expected cost per period of running this system. Thus, it is crucial to add the cost vector to the average system size equation to obtain long run average cost, which is

$$\lim_{T\to\infty} \frac{1}{T} \int_0^T (Z_1(t)h_1 + Z_2(t)h_2 + Z_3(t)h_3)\ dt$$

h is the cost vector for waiting in the i^{th} buffer.

The utilization of each machine is also important, denoted ρ_j, for each station j in the setting of reentrant line. If λ = 1job/min, and m_i = [.8 .9 .1] for process 1, 2 and 3. $\rho_1 = 1 \times (.8 + .1) = .9$, because for every arrival into the system, the work load for station one is .9 minute. Similarly, $\rho_2 = 1 \times .9 = .9$, because for every arrival into the system, the work load for station two is .9 minute. In general

$$\rho_i = \lambda \times \sum_i\ m_i$$

where only add up those $m_i\ 's$ that are using the same station.

3 Truncation Number

Truncation number(denoted by N), is the maximum number of jobs allowed in system. Meaning $Z_1(t) + Z_2(t) + Z_3(t) \leq N$ for any t. The goal is to trying to find the optimal policy when the system is not capacitated. We can essentially increase N and see its effects on both Long run average cost and utilization. So as N gets larger and larger, our hope is that the long run average cost will converge and utilization will also converge to theoretical value.

4 What is a policy?

As I mentioned before, the state space for this system Z(t) is equal to $(Z_1(t), Z_2(t), Z_3(t))$. A policy for the reentrant system is basically the specification of how the system will perform once encounter a certain state. Define action 1 to be working on buffer 1 before buffer 3, and action 2 to be working on buffer 3 before buffer 1. So an example of a policy would be the following. At state (i, j, k), where i,j,k are positive integers, do action 1. It means if the system happens to have i number of jobs in queue 1, j number of jobs in queue 2, k number of jobs in queue 3, one should put priorities on the jobs in buffer 1 before buffer 3. Also define, last buffer first serve(LBFS) policy, meaning buffer 3 always get priority before buffer 1. Under the same logic, define first buffer first serve(FBFS) policy, meaning buffer 1 always get priority before buffer 3.There are also other important assumptions needed to be made.

First one is whether or not to allow idleness. In this context, allowing idleness meaning allowing either machine to be not working on any job for a certain state if that is indeed cheaper. This paper will study both cases of allowing idleness and not allowing idleness.

Next assumption is allowing preemption or not. In the setting of LBFS (last buffer first serve) policy meaning jobs at buffer 3 will get priority over jobs at buffer 1. Preemptive policy allows for station 1, even if the station is currently processing jobs from buffer 1, whenever there is a job shows up at buffer 3, the station will stop working on buffer 1's job and immediately switch to buffer 3. Once buffer 3 is empty, station 1 switches back to perform buffer 1's job, since the process time is exponentially distributed, resume the unfinished job makes no difference from restarting the unfinished job in terms of process time. Non-preemptive policy on the other hand, means in the previous situation, station 1 will complete the job from buffer 1 before working on buffer 3's job.

For example, in the setting of LBFS, and currently the system is in state (2,3,4), the possible candidate for next state space using preemptive policy is (3,3,4) meaning a job arrives in buffer 1 with rate λ, (2,2,5) meaning a job finished from station 2 with rate μ_2 , (2,3,3) meaning a job finishes from process 3 with rate μ_3 . Alternatively, if non-preemptive policy is used, specifying what job station 1 is working on is necessary to distinguish between the 2 states. For example, if currently in state (2,3,4;1) meaning station 1 is working on a job from buffer 1, the possible candidate for the next state space

is (3,3,4;1) with rate λ , (2,2,5;1) with rate μ_2 , and (1,4,4;3) with rate μ_1 meaning a job from buffer 1 finishes and went to buffer 2 and now station 1 is working on buffer 3's job because of our LBFS policy. For (2,3,4;3) on the other hand meaning station 1 is working on a job from buffer 3, the possible candidate for the next state space is (3,3,4;3) with rate λ , (2,2,5;3) with rate μ_2 , and (2,3,3;3) with rate μ_3 meaning a job from buffer 3 finishes and went out of the system.

If the arrival rate of the system λ=1job/min, the process rate of each of the three steps are μ_1=4jobs/min, μ_2=1.2jobs/min, μ_3=2jobs/min. First of all, making sure the utilization of both stations are under 1 is very important to make sure the system works. $\rho_1 = 1 \times (\frac{1}{4} + \frac{1}{2}) = \frac{3}{4}$. $\rho_2 = 1 \times \frac{1}{1.2} = \frac{5}{6}$. Table 1 is the generator for LBFS preemptive policy while Table 2 is the generator for LBFS non-preemptive policy when N =2.

In this paper, preemption is allowed.

Table 1: LBFS preemptive policy

current state/next state	(0, 0, 0)	(1, 0 ,0)	(0, 1 ,0)	(0, 0 ,1)	(2, 0 ,0)	(0, 2 ,0)	(0, 0 ,2)	(1, 1 ,0)	(0, 1 ,1)	(1, 0 ,1)
(0, 0, 0)	−1	1								
(1, 0, 0)		−5	4		1					
(0, 1, 0)			−2.2	1.2				1		
(0, 0, 1)	2			−3						1
(2, 0, 0)					−4			4		
(0, 2, 0)						−1.2			1.2	
(0, 0, 2)				2			−2			
(1, 1, 0)						4		−5.2		1.2
(0, 1, 1)			2				1.2		−3.2	
(1, 0, 1)		2								−2

Table 2: LBFS non-preemptive policy

current state/next state	(0, 0, 0)	(1, 0 ,0)	(0, 1 ,0)	(0, 0 ,1)	(2, 0 ,0)	(0, 2 ,0)	(0, 0 ,2)	(1, 1 ,0)	(0, 1 ,1)	(1, 0 ,1;1)	(1, 0, 1;3)
(0, 0, 0)	−1	1									
(1, 0, 0)		−5	4		1						
(0, 1, 0)			−2.2	1.2				1			
(0, 0, 1)	2			−3							1
(2, 0, 0)					−4			4			
(0, 2, 0)						−1.2			1.2		
(0, 0, 2)				2			−2				
(1, 1, 0)						4		−5.2		1.2	
(0, 1, 1)			2				1.2		−3.2		
(1, 0, 1;1)									4	−4	
(1, 0, 1;3)		2									−2

5 Stationary Policy Reentrant Line

Our goal is to find the best policy for this system. In order to do that, a two step procedure called policy evaluation and policy iteration is introduced. Policy evaluation is testing how good the current policy is while policy iteration is the iterative procedure to converge upon the optimal policy. Since this system is integrated over time, which is a continuous variable, Continuous Time Markov Chain (CTMC) is the suitable modeling method. Policy evaluation is done by calculating long run average cost along with relative value function.

In order to calculate long run average cost, first of all, it is essential to get the Generator for this CTMC. Set N as the truncation number, and n_i as the number of jobs in process i (including jobs waiting in buffer and jobs-in-progress). This rule can be expressed as $\sum_i \ n_i \leq \ N$ mathematically. In general, the number of states with N jobs in system and k buffers is

$$\sum_{n=0}^{N} \binom{n+k-1}{k-1}$$

5.1 Policy Evaluation

Now, the generator is ready to find the long run average cost using poisson equation, and ultimately find the optimal policy. The poisson equation for CTMC is $Gv+C = \gamma\ e$, where G is the generator, v is the relative value function meaning the average cost of starting from each state relative to one state which is set to 0, C is the cost vector h for inventory in each process, γ is long run average cost, and e is a column of 1's. In order to capture relative value function v, as stated before, one relative value needs to be set to 0. For example, set the first v_1 to be 0 for this case. To solve this, move Gv to the other side result in $C = \gamma\ e\ -\ Gv$, then do $-1 \times G$ and set the first column of G to 1, and replace v_1 (which already being set to 0) with γ. Now the equation is ready to be solved using inverse of matrix. To illustrate the idea clearer, set a 3 by 3 generator as an example.

1. Set $G = \begin{pmatrix} -3 & 2 & 1 \\ 1 & -2 & 1 \\ 2 & 3 & -5 \end{pmatrix}$ and $C = \begin{pmatrix} 10 \\ 15 \\ 20 \end{pmatrix}$, formulation looks like,

$$\begin{pmatrix} -3 & 2 & 1 \\ 1 & -2 & 1 \\ 2 & 3 & -5 \end{pmatrix} \times \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} + \begin{pmatrix} 10 \\ 15 \\ 20 \end{pmatrix} = \gamma \times \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

2. Set v_1 to 0 and move Gv over,

$$\begin{pmatrix} 10 \\ 15 \\ 20 \end{pmatrix} = \begin{pmatrix} \gamma \\ \gamma \\ \gamma \end{pmatrix} - \begin{pmatrix} -3 & 2 & 1 \\ 1 & -2 & 1 \\ 2 & 3 & -5 \end{pmatrix} \times \begin{pmatrix} 0 \\ v_2 \\ v_3 \end{pmatrix}$$

3. Do $-1 \times G$ and set the first column of G to 1, then replace the first v (which already being set to 0) with γ , result is

$$\begin{pmatrix} 10 \\ 15 \\ 20 \end{pmatrix} = \begin{pmatrix} 1 & -2 & -1 \\ 1 & 2 & -1 \\ 1 & -3 & 5 \end{pmatrix} \times \begin{pmatrix} \gamma \\ v_2 \\ v_3 \end{pmatrix}$$

4. Solve
$$\begin{pmatrix} \gamma \\ v_2 \\ v_3 \end{pmatrix} = \begin{pmatrix} 1 & -2 & -1 \\ 1 & 2 & -1 \\ 1 & -3 & 5 \end{pmatrix}^{-1} \times \begin{pmatrix} 10 \\ 15 \\ 20 \end{pmatrix}$$

5. $\begin{pmatrix} \gamma \\ v_2 \\ v_3 \end{pmatrix} = \begin{pmatrix} 14.375 \\ 1.25 \\ 1.875 \end{pmatrix}$, where γ is long run average cost,v_2 and v_3 are the relative cost starting from state 2 and 3 with respect to v_1, where $v_1 = 0$

5.2 Policy iteration

With those statistics in hand, optimal policy for this system can be found by policy iterations, which means compare between difference actions for each state and find the best action for each state then start from policy evaluation again. Once the best action does not change any more, it means the current policy is optimal. For example, we can compare the difference between FBFS policy and LBFS policy for state $(1, 0, 1)$. Let v be the relative value function from policy evaluation. Also let G_1, G_2 be the generator for FBFS and LBFS policy respectively. We are indeed comparing the current cost plus future cost for those two policies. So, for FBFS, current cost plus future cost equals $h_1 + h_3 + G_1(101) * v$, where $G_1(1, 0, 1)$ is the row in the generator for FBFS corresponding to state $(1, 0, 1)$. Similarly, for LBFS, current cost plus future cost equals $h_1 + h_3 + G_2(101) * v$, where $G_2(1, 0, 1)$ is the row in the generator for LBFS corresponding to state $(1, 0, 1)$. Now, whichever of those two numbers are cheaper is the more cost efficient action to take. Note that since both terms has $h_1 + h_3$ in them, we can simply drop them and just compare $G_1(101)$*v with $G_2(101) * v$. Finding the best action space for this situation then go back to do policy evaluation again until the action space remains the same. The result is the optimal policy.

6 Index scheme

One very important step to take in order to model this system is to develop a proper state space index scheme. The scheme essentially shows at any given point of time, what does the system currently look like. An old index scheme can be found in appendix A. In this section, a new index scheme of this specific system will be introduced.

6.1 New Scheme Overview

Considering the problem with the old index Scheme, the goal of the new scheme will be not only indexing each state in a very organized manner but solve the problem in the old scheme as well. We want to make our transition matrix more concentrated in the diagonal

so that it can be computed quicker than before. In fact, it is obviously that for each state transition, the number of items in each buffer will at most go up or down by one. Thus, the new indexing scheme will organize each state based on the total number of items in all three buffers, so that an addition or reduction of one job in each buffer will not drastically change the index of the next possible state. An example of the new index scheme can be found in Table 3.

Table 3: New Index Example

State	Jobs in buffer i	Jobs in buffer j	Jobs in buffer k
1	0	0	0
2	1	0	0
3	0	1	0
4	0	0	1
5	2	0	0
6	1	1	0
7	0	2	0
8	1	0	1
...	...	...	...

6.2 New Scheme formulation

The new index scheme is based on the total number of jobs in system, hereby denote as n. Also let i,j,k be the number of jobs in each buffer 1,2,3 respectively. Notice, n is equal to the sum of i,j,k. Let p be the corresponding position in the state space. As an example, from Table 3, the third row shows when $i = 0$,$j = 1$,$k = 0$, the corresponding $p = 3$. It is also very important to keep in mind that the total number of states up until the number of jobs in the system reaches n is equal to $\binom{n+3}{3}$.

In order to fully utilize this scheme, the formulation of translating i,j,k to p and translating p back to i,j,k is crucial. Let's do the first one first. It essentially means giving a specific set of jobs in each buffer, the function will generate the exact state number in which the system is currently in. The formulation will look like the following.

$$p = \binom{n+2}{3} + \frac{k(2n+3-k)}{2} + j + 1$$

Now, for the equation going from p to i,j,k, it is imperative to first find the right n. Once n is known, then the next step is to identify i,j and k. Notice, for any p, p has to be at least equal to $\binom{n+2}{3}$ and at most equal to $\binom{n+3}{3}$. Thus, the following inequality must be true.

$$\binom{n+2}{3} \leq p \leq \binom{n+3}{3} \Rightarrow \frac{(n+2)\ (n+1)\ (n)}{6} \leq p \leq \frac{(n+3)\ (n+2)\ (n+1)}{6} \quad (6.1)$$

Also because the following two expressions

$$(n+2)\ (n+1)\ (n) > \ n^3$$

$$(n+3)\ (n+2)\ (n+1) < \ (n+2)^3$$

it is reasonable to assume the following

$$n^3 < 6p < \ (n+2)^3 \Rightarrow \ n < \sqrt[3]{6p} < \ n+2$$

Thus, $\lfloor \sqrt[3]{6p} \rfloor$ is either equal to n or $n+1$. Only one number will fit equation 6.1. So far, n is found. The next step is going from n to i,j,k. In order to do that, set $m = \ p - \frac{(n+2)(n+1)(n)}{6}$,we obtain

$$\frac{k(2n+3-k)}{2} < \ m \leq \frac{(k+1)(2n+2-k)}{2}$$

Through quadratic formula, the formulation becomes

$$\frac{(2n+1) - \sqrt{(2n+3)^2 - 8m}}{2} \leq \ k \leq \frac{(2n+3) - \sqrt{(2n+3)^2 - 8m}}{2}$$

Since the lower bound and upper bound of k only differs by 1, it is safe to assume that k is equal to the roof of the lower bound.

$$k = \lceil \frac{(2n+1) - \sqrt{(2n+3)^2 - 8m}}{2} \rceil \tag{6.2}$$

$$j = \ m - \frac{k(2n+3-k)}{2} - 1 \tag{6.3}$$

$$i = n - k - j \tag{6.4}$$

Since it is shown that given p, through this formulation i,j,k is identified; also given i,j,k, p is identified. Thus, this index scheme is useful to model the operations in three machine two buffer reentrant line.

7 Searching for Solver

A big part of this whole program is solving the poisson equation and obtain an accurate long run average cost, that makes the solver extremely important. The purpose of this section is to highlight the process of choosing a suitable solver to fit our program. This is mostly a two step process, where step one is finding a "good" system size, the "good" here meaning the calculated utilization is close to theoretical. Step two is using that system size and solve for the stationary distribution π. Then using $longrunaveragecost = \pi' \times costvector$, I am able to find the target value of long run average cost. What's remaining to do is to

Table 4: Data

N	Machine one Util	Machine two Util	Solving time
140	.9695	.9695	1989.911207 seconds
170	.9725	.9725	4342.593642 seconds
180	.9732	.9732	6761.181127 seconds
190	.9739	.9739	8998.422883 seconds
195	.9742	.9742	10064.562879 seconds
205	.9748	.9748	14058.091007 seconds

find a good solver such that the long run average cost obtained using that particular solver to solve poisson equation is close to this target value. Note that FBFS policy will be used for the sole purpose of testing the effectiveness of various solvers. Upon finding the best solver, it will be applied to the larger goal of MDP.

7.1 Part A

First of all, the concept of null space is used to help solving for our stationary distribution. The matlab code of null space solver is in appendix C as well. Note that those two codes are both needed to be in the save folder in order to call the function. The arrival rate is 1.96 so that our theoretical utilization is 98% and we want to find the system size that can achieve an utilization within 1% of that value. Under the policy of first buffer first serve (FBFS), Table 4 shows the solving time and the utilization of each possible N. Note that N=190 or above might fail sometime using the null space solver.

Given the table above, system size equals to 150 is chosen to be the result of the first step

7.2 Part B

Note that all tests are being done using $\lambda = 1.96$. All solvers are set to default accuracy which is $1e-6$ and maximum iteration is set to be 10000. Also the TARGET long run average cost is 91.9214.

The goal for this section is to find a good solver to solve poisson equation, so that the resulting long run average cost is close to the result obtained from null space solver. In parallel with trying to do that, I also tested various solvers for their capabilities of solving $\pi G = 0$ directly. Let me present the result for this first then I will present the result for solving poisson equation.

Out of numerous iterative solvers I only picked four to test with. The main reason for ruling some of those out before testing is matrix restriction like the requirement of the

Table 5: Solve $\pi G = 0$ Total=150

	γ	Machine one Util	Machine two Util	Solving time
BICG	0	0	0	1691.18 seconds
BICGSTAB	7.771	.8019	.7709	3134.69 seconds
LSQR	117.6262	.9812	.9875	2829.63 seconds
QMR	.0415	.0208	0	1700.43 seconds

Table 6: Solve $\pi G = 0$ Total=95

	γ	Machine one Util	Machine two Util	Solving time
BICG	0	0	0	1691.18 seconds
BICGSTAB	63.0815	0.9612	0.9611	103.25 seconds
LSQR	76.6124	.9737	.9817	291.12 seconds
QMR	3.5383	.5877	.4830	1084.47 seconds

matrix to be symmetric for example the Preconditioned conjugate gradients method. In the end, I picked the following four methods: BiConjugate Gradients Method(BICG), Bi-Conjugate Gradients Stabilized Method(BICGSTAB), LSQR Method and Quasi-Minimal Residual Method (QMR). Two different total(95 and 150) are being used for this test. The results shows that when total=150 every single solver fails, the reason for this I think would be our right hand side is a vector is all 0's except for one number being 1, so it's a fairly ill-conditioned vector. At total=95, bicgstab gives a fairly reasonable result compare to others. Table 5 is the result for total=150 and Table 6 is the result for total=95.

For the test of which solver gives a good long run average cost closest to the target value, One more solvers are introduced in addition to those exist four. It's called Biconjugate gradients stabilized (l) method(BICGSTABL). BICGSTABL is exactly equal to BICGSTAB when the parameter l is equal to one. Note that the target long run average cost is 91.9214. Table 7 is the results of using those five solvers to solve poisson equation in order to find a long run average cost close to the target value. In conclusion, BICGSTABL has a slightly more accurate result at a cost of almost 5 times the calculation time. BICGSTAB gives a long run average cost differs from the target value at 0.1 level and BICGSTABL gives a long run average cost differs from the target value at .0001 level. Judging from the chart, the implementation of BICGSTAB in the MDP code is a fairly good choice and should produce a good result. BICGSTAB was chosen as the solver due to its accuracy and reasonable running times.

Table 7: Solve poisson Total=150

	γ	Solving time
BICG	192.8503	724.34 seconds
BICGSTAB	91.3499	43.73 seconds
LSQR	119.0538	795.96 seconds
QMR	147.6766	100.22 seconds
BICGSTABL	91.9213	197.16 seconds

8 Matlab Codes for new index scheme

Using the state index scheme developed in the last section, it is necessary to also develop corresponding Matlab codes to generate the optimal policy. We should distinguish between policy with idleness(allowing machine to be idle at certain state) and without idleness. Since C++ is more efficient at doing loops, we decide to use C++ to make the generator and interface it with Matlab so that Matlab can use this generator for further analysis. In this case, it is also more convenient to generate cost vector inside C++ as well, due to the fact that looping is also required to generate the cost vector. The compiler that I used is Microsoft Visual C++ 2008. The difference in C++ codes between idle and non-idle policy is basically Idle policy has more actions which allow a machine to be idle at a certain state, while non-idle policy will have less actions because it always needs to busy working on something. Details of each action will be presented later along with some results of the program. The policy improvement algorithm here is basically generate a matrix, where each row is a state and each column represent the cost of take certain action at the specific state. We then analyze the matrix row by row to see which action is the cheapest. By iterating this algorithm, it will reach a point where no more significant improvements can be made anymore. Then, the optimal policy has been reached. The program will generate which action to take at each state and the optimal long run average cost of taking those actions. In appendix C, there are Matlab codes and C++ codes used to generate optimal policy, calculating utilizations(note that the code included is for nonidle policy, for idle just change the index of matrix from 2 to 5), code for Nullspace solver(used to solve $\pi G = 0$ which is useful to find utilization) and code for graphing the actions so that we can visualize it.

9 Results

This section will be dedicated to provide some results and observations from the above codes. First of all, it is worth noting that the parameters used to generate most of the

following results are shown in Table 8. This section is divided into three subsections where the first one will display the results for long run average cost, the second one is devoted to utilization and the third one is for observations. Note that some sample optimal policy and its corresponding plots are available in appendix B.

Table 8: Parameters used

Parameter	
Process rate buffer 1	3
Process rate buffer 2	2
Process rate buffer 3	6
Arrival rate(λ)	Varied
Maximum jobs allowed in system(Totals)	Varied
Holding cost buffer 1	2
Holding cost buffer 2	1
Holding cost buffer 3	1.5

9.1 Long run average cost

The results are based on two different arrival rate and several different N's. Table 9 shows the long run average cost and its corresponding solving time for the idle policy while Table 10 shows the same thing for non-idle policy.

Table 9: Long run average cost-Idle

Total	$\gamma(\lambda = 1.7)$	Solving time in seconds	$\gamma(\lambda = 1.96)$	Solving time in seconds
20	10.4781	2.1377	14.3924	1.8187
50	14.4718	17.5313	30.3314	27.0142
80	14.9912	1243.0119	43.6178	1239.0745
110	15.0377	2279.2695	55.1311	4332.6435
140	15.0411	5906.7566	65.2053	7086.1507
170	15.0413	7269.5327	74.0287	6979.3833
200	15.0413	9345.1847	81.7422	9960.8784

9.2 Utilization

It is also important to take a look at the utilization of each machine in different settings. Theoretically, as I have mentioned in the beginning of this paper, if arrival rate is λ and mean service time for buffer 1 2 and 3 are μ_1, μ_2 and μ_3 respectively, we can obtain the utilization for machine one by doing $\lambda(\mu_1 + \mu_3)$ and utilization for machine two by doing

Table 10: Long run average cost-nonIdle

Total	$\gamma(\lambda = 1.7)$	Solving time in seconds	$\gamma(\lambda = 1.96)$	Solving time in seconds
20	10.5658	0.7368	14.5338	3.2147
50	14.5283	113.5187	31.0602	147.1050
80	14.9992	1105.9223	44.9853	1496.9165
110	15.0384	2463.8067	57.0554	1314.7304
140	15.0411	4334.8160	67.5646	10051.2236
170	15.0413	5439.3160	76.7171	10257.3785
200	15.0413	12524.0859	84.9133	19918.6757

$\lambda\mu_2$. For example, when arrival rate is 0.5jobs/unit time, and mean service times are $\frac{1}{3}$, $\frac{1}{2}$,and $\frac{1}{6}$, we expect to have utilization for machine one to be $0.5\times(\frac{1}{3}+\frac{1}{6}) = \frac{1}{4}$ and utilization for machine two to be $0.5 \times \frac{1}{2} = \frac{1}{4}$. Now, for this program, the way to calculate utilization is by first finding out the stationary distribution for all states using nullspace solver. Then, for example, to find out the utilization for machine one under the idling policy, we should only add up those stationary distributions for states where machine one has something in buffer 1 or 3 and it is not forced to be idle. It is basically adding up all the long run fraction of time that the machine is not idle and not starved. Also using "—" to denote the program didn't solve for a very long time and is terminated manually. Table 11 shows the utilization for machine one and machine two along with its corresponding solving time for the idle policy while Table 12 shows the same thing for non-idle policy.

Table 11: Utilization-Idle

Total	Utilization($\lambda = 1.7$)	Solving time in seconds	Utilization($\lambda = 1.7$)	Solving time in seconds
20	0.8223\0.8154	0.0675	0.8930\0.8864	0.0519
50	0.8481\0.8473	4.0203	0.9458\0.9362	3.3837
80	0.8499\0.8498	136.8827	0.9607\0.9526	89.5322
110	0.8500\0.8500	1101.4284	0.9676\0.9611	847.4564
140	– – – – –	– – ––	– – – – –	– – ––
170	– – – – –	– – ––	– – – – –	– – ––
200	– – – – –	– – ––	– – – – –	– – ––

9.3 Observations

When arrival rate equals 1.7, it seems that the long run average cost converges when maximum systems size equals around 110 or 140 depending on what kind of accuracy is needed. And theoretical utilization can be achieved. However, when arrival rate equals 1.96, long run average cost keeps increasing even when maximum system size equals 200,

Table 12: Utilization-nonIdle

Total	Utilization($\lambda = 1.7$)	Solving time in seconds	Utilization($\lambda = 1.96$)	Solving time in seconds
20	0.8160\0.8160	0.0650	0.8793\0.8793	0.1294
50	0.8476\0.8476	6.7353	0.9394\0.9394	6.3433
80	0.8498\0.8498	174.3054	0.9569\0.9569	106.0623
110	– – – – –	– – – –	0.9651\0.9651	902.3632
140	– – – – –	– – – –	– – – – –	– – – –
170	– – – – –	– – – –	– – – – –	– – – –
200	– – – – –	– – – –	– – – – –	– – – –

and the best utilization archived is around 96.7% while the theoretical utilization is 98%.

Table 13: Nullspace solver limit

Total	Utilization machine one	Utilization machine two	Solving time in seconds
110	0.9647	0.9647	237.6307
140	0.9695	0.9695	1989.9112
170	0.9725	0.9725	2717.0514
180	0.9732	0.9732	6761.1811
190	0.9739	0.9739	8998.4228
195	0.9742	0.9742	10064.5628
200	run 1 failed and run 2 yields 0.9745	0.9745	8848.3197
205	0.9748	0.9748	14058.0910
210	failed	failed	– – – – –
220	failed	failed	– – – – –
230	failed	failed	– – – – –

10 Acknowledgement

This report results from two research courses from Professor J. G. "Jim" Dai. I am very grateful for his guidance on this paper. I also want to thank Nathen Petty and Dan Roshbach for their excellent previous work on this topic.

11 Problems unsolved

Please refer to appendix B before reading this section. There are still problems with this program. For example, looking at Figure 6, when buffer one has 0 items in it, machine one should be working on buffer three, which means there should be a dot there. But it didn't

have one. The reason for that lies within the way we defined actions. Action one is FBFS which means if buffer one has items in it, work on buffer one, but if buffer one is empty, then work on buffer three. Similarly, action two is defined to be if buffer three has items in it, work on buffer three, but if buffer three is empty, then work on buffer one. So with this definition of actions, if buffer one has no items in it, we can use action one, but there is no items in buffer one, so the machine will still be working on buffer three, and now it does make sense. But actions needed to be redefined for MDP processes.

12 Appendix A

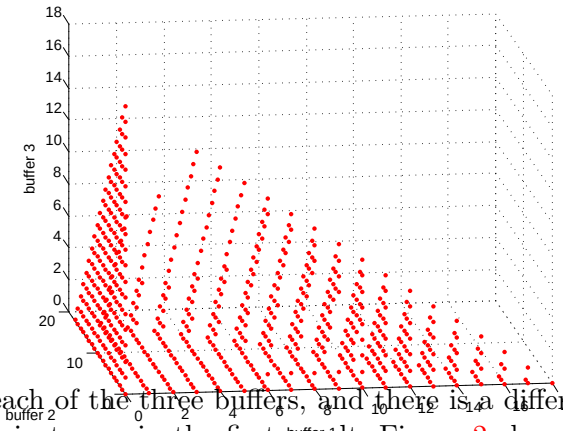
The previous index scheme for each possible state in a two machine three buffer reentrant line setting is based on the maximum buffer size of each buffer. If the maximum size of each buffer is two, an example of the possible state space are displayed in Table 14. This index scheme has a problem of potentially inefficient in term of state transition, because there is a chance of the system going from a very small state(e.g state 5) to a very large state(e.g state 200) in one transition if the buffer size is large enough. The reason for that is if all three buffer sizes are very large, then moving one item from buffer two to buffer three will cause the index to jump over all the intermediate states in between which will be very large since the buffer size is large.

Table 14: Old Index Example

State	Jobs in buffer i	Jobs in buffer j	Jobs in buffer k
1	0	0	0
2	1	0	0
3	2	0	0
4	0	1	0
5	1	1	0
6	2	1	0
7	0	2	0
8	1	2	0
...	...	...	...

13 Appendix B

This section will display some sample optimal policy and their corresponding plot under non-idling and non-preemption setting. For optimal policy, each column corresponds to each state and if there is a 1 there meaning we use FBFS policy and if there is a 2 there meaning we use LBFS policy. For the plot, each of the three axis represent the number



of jobs in each of the three buffers, and there is a different color associated with different policy. For instance, in the first result, Figure 2 shows exactly where will we use FBFS policy while Figure 3 shows where to use LBFS policy by putting a dot at corresponding location. Figure 4 through Figure 21 are individual plots for LBFS action when the number of items in second buffer equals to $0, 1...17$. Of course, when second buffer have 18 items in it, the other two buffers will have 0 items in them, hence there is no jobs being worked on in machine one. Two results will be shown. First one will be included in this paper and second one will be on separate text file. First result is when $\lambda = 0.9$ and $N = 18$. And the second result is when $\lambda = 1.7$ and $N = 110$. Note that the truncation number N is tested to make sure that the result is already converged. So N is always sufficiently large to ensure a valid optimal policy.

13.1 Optimal policy sample 1(figures)

Figure 2: FBFS

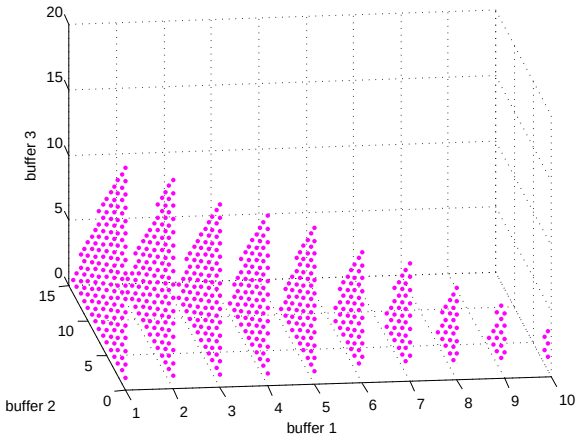


Figure 3: LBFS

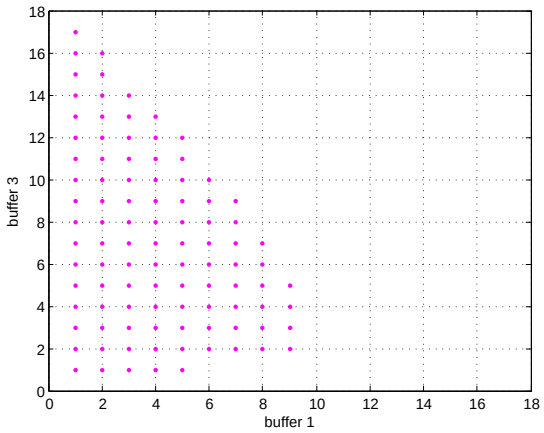


Figure 4: Second buffer has 0

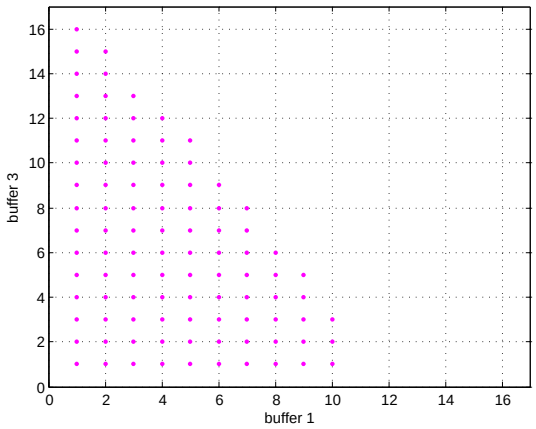


Figure 5: Second buffer has 1

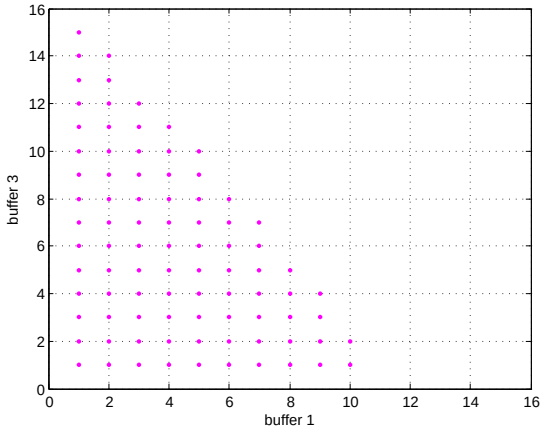


Figure 6: Second buffer has 2

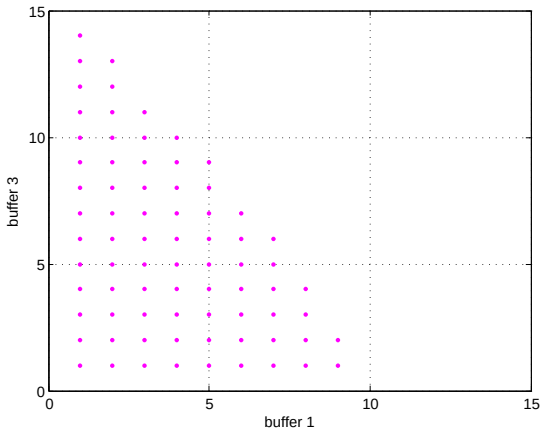


Figure 7: Second buffer has 3

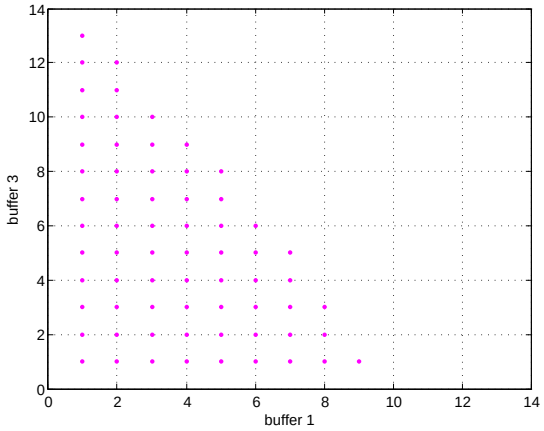


Figure 8: Second buffer has 4

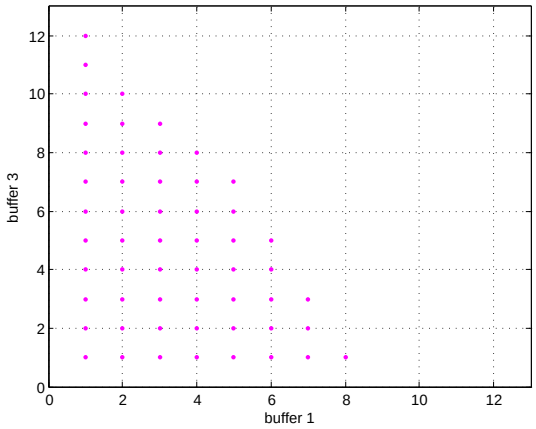


Figure 9: Second buffer has 5

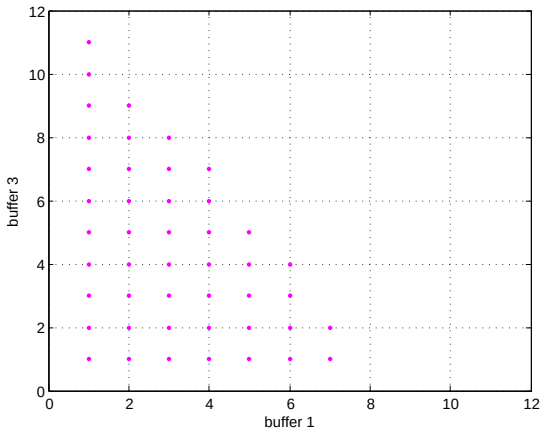


Figure 10: Second buffer has 6

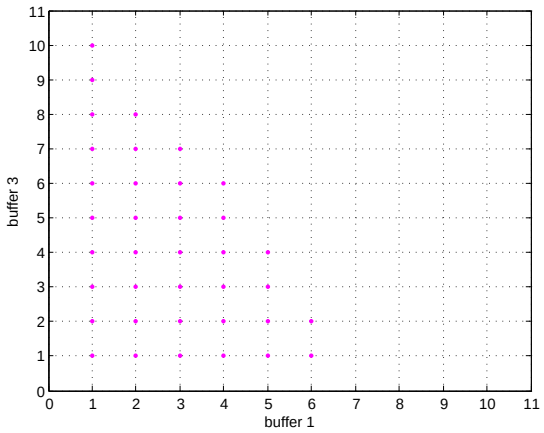


Figure 11: Second buffer has 7

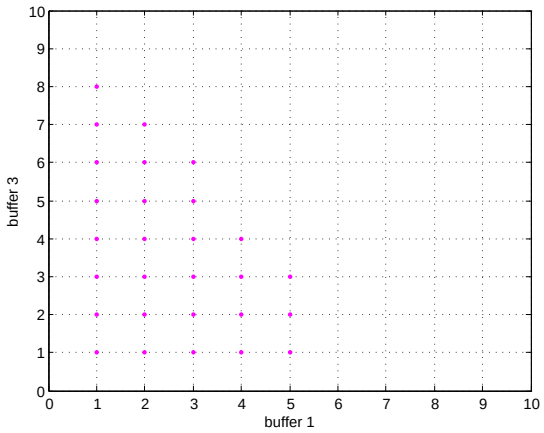


Figure 12: Second buffer has 8

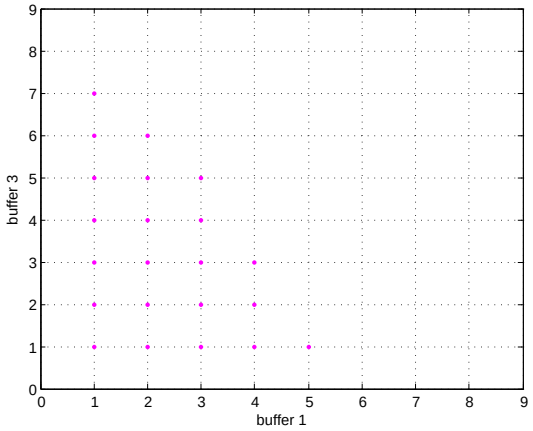


Figure 13: Second buffer has 9

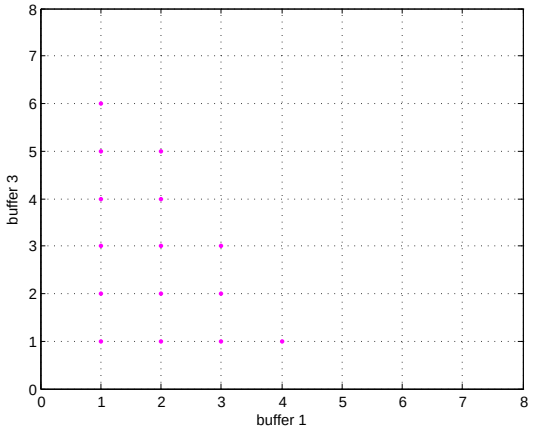


Figure 14: Second buffer has 10

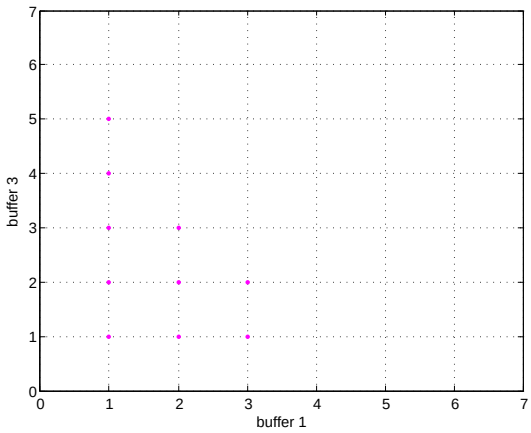


Figure 15: Second buffer has 11

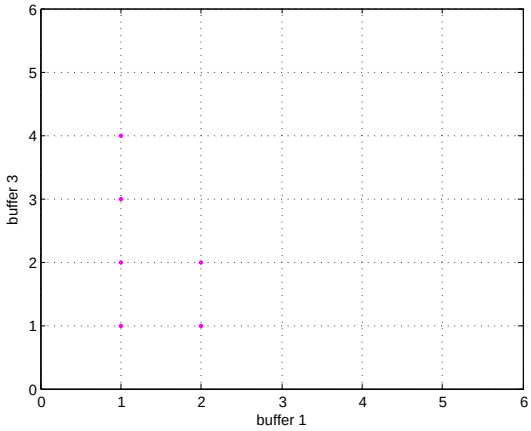


Figure 16: Second buffer has 12

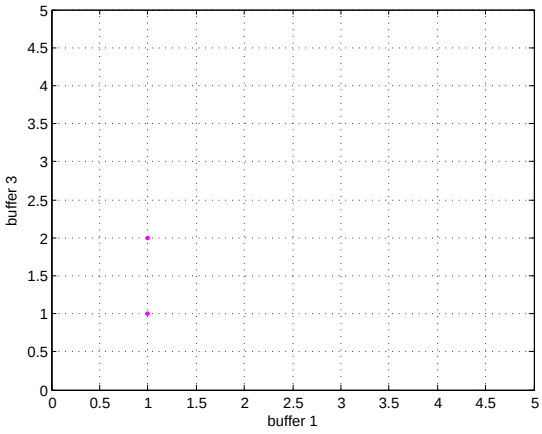
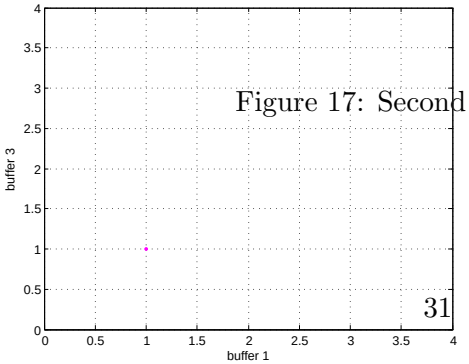


Figure 17: Second buffer has 13



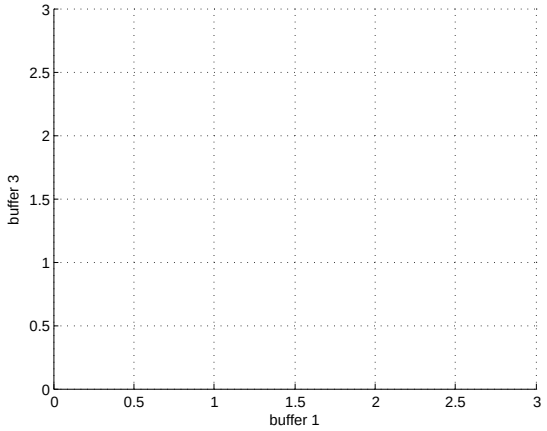


Figure 19: Second buffer has 15

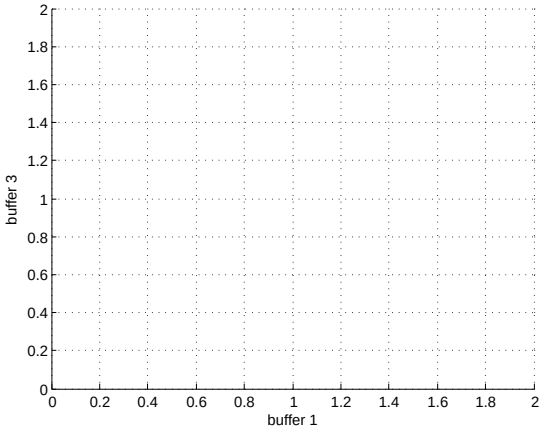


Figure 20: Second buffer has 16

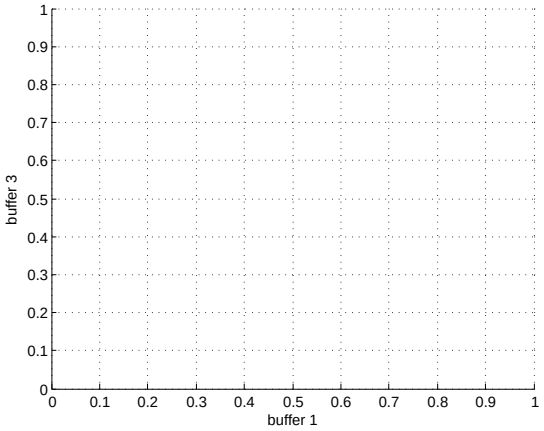


Figure 21: Second buffer has 17

13.2 Optimal policy sample 1(policy)

Here we will display the actual policy. Table 15 shows the state indexes for the this sample solution.

Table 15: State index for sample one

State	Jobs in buffer i	Jobs in buffer j	Jobs in buffer k
1	0	0	0
2	1	0	0
3	0	1	0
4	0	0	1
5	2	0	0
6	1	1	0
7	0	2	0
8	1	0	1
9	0	1	1
10	0	0	2
11	3	0	0
12	2	1	0
13	1	2	0
...	...	...	...
40	1	4	0
...	...	...	...

```
Columns 1 through 13

    1     1     1     1     1     1     1     2     1     1     1     1     1

  Columns 14 through 26

    1     2     2     1     2     1     1     1     1     1     1     1     2

  Columns 27 through 39

    2     2     1     2     2     1     2     1     1     1     1     1     1

  Columns 40 through 52

    1     1     2     2     2     2     1     2     2     2     1     2     2

  Columns 53 through 65
```

```
   1     2     1     1     1     1     1     1     1     1     1     2     2

Columns 66 through 78

   2     2     2     1     2     2     2     2     1     2     2     2     1

Columns 79 through 91

   2     2     1     2     1     1     1     1     1     1     1     1     1

Columns 92 through 104

   1     1     2     2     2     2     2     1     2     2     2     2     2

Columns 105 through 117

   1     2     2     2     2     1     2     2     2     1     2     2     1

Columns 118 through 130

   2     1     1     1     1     1     1     1     1     1     1     1     1

Columns 131 through 143

   2     2     2     2     2     2     1     2     2     2     2     2     2

Columns 144 through 156

   1     2     2     2     2     2     1     2     2     2     2     1     2

Columns 157 through 169

   2     2     1     2     2     1     2     1     1     1     1     1     1

Columns 170 through 182

   1     1     1     1     1     1     1     2     2     2     2     2     2

Columns 183 through 195

   2     1     2     2     2     2     2     2     2     1     2     2     2

Columns 196 through 208

   2     2     2     1     2     2     2     2     2     1     2     2     2
```

```
Columns 209 through 221

   2     1     2     2     2     1     2     2     1     2     1     1     1

Columns 222 through 234

   1     1     1     1     1     1     1     1     1     1     1     2     2

Columns 235 through 247

   2     2     2     2     2     2     1     2     2     2     2     2     2

Columns 248 through 260

   2     2     1     2     2     2     2     2     2     2     1     2     2

Columns 261 through 273

   2     2     2     2     1     2     2     2     2     2     1     2     2

Columns 274 through 286

   2     2     1     2     2     2     1     2     2     1     2     1     1

Columns 287 through 299

   1     1     1     1     1     1     1     1     1     1     1     1     1

Columns 300 through 312

   2     2     2     2     2     2     2     2     2     1     2     2     2

Columns 313 through 325

   2     2     2     2     2     2     1     2     2     2     2     2     2

Columns 326 through 338

   2     2     1     2     2     2     2     2     2     2     1     2     2

Columns 339 through 351

   2     2     2     2     1     2     2     2     2     2     1     2     2

Columns 352 through 364
```

```
   2     2     1     2     2     2     1     2     2     1     2     1     1

Columns 365 through 377

   1     1     1     1     1     1     1     1     1     1     1     1     1

Columns 378 through 390

   1     2     2     2     2     2     2     2     2     2     2     1     1

Columns 391 through 403

   2     2     2     2     2     2     2     2     2     1     2     2     2

Columns 404 through 416

   2     2     2     2     2     2     1     2     2     2     2     2     2

Columns 417 through 429

   2     2     1     2     2     2     2     2     2     2     1     2     2

Columns 430 through 442

   2     2     2     2     1     2     2     2     2     2     1     2     2

Columns 443 through 455

   2     2     1     2     2     2     1     2     2     1     2     1     1

Columns 456 through 468

   1     1     1     1     1     1     1     1     1     1     1     1     1

Columns 469 through 481

   1     1     1     2     2     2     2     2     2     2     2     2     2

Columns 482 through 494

   1     1     2     2     2     2     2     2     2     2     2     2     1

Columns 495 through 507

   1     2     2     2     2     2     2     2     2     2     1     2     2
```

```
Columns 508 through 520

   2     2     2     2     2     2     2     1     2     2     2     2     2

Columns 521 through 533

   2     2     2     1     2     2     2     2     2     2     2     1     2

Columns 534 through 546

   2     2     2     2     2     1     2     2     2     2     2     1     2

Columns 547 through 559

   2     2     2     1     2     2     2     1     2     2     1     2     1

Columns 560 through 572

   1     1     1     1     1     1     1     1     1     1     1     1     1

Columns 573 through 585

   1     1     1     1     1     1     1     2     2     2     2     2     2

Columns 586 through 598

   2     2     2     1     1     1     2     2     2     2     2     2     2

Columns 599 through 611

   2     2     2     1     1     2     2     2     2     2     2     2     2

Columns 612 through 624

   2     2     1     1     2     2     2     2     2     2     2     2     2

Columns 625 through 637

   1     2     2     2     2     2     2     2     2     2     1     2     2

Columns 638 through 650

   2     2     2     2     2     2     1     2     2     2     2     2     2

Columns 651 through 663
```

```
   2     1     2     2     2     2     2     2     1     2     2     2     2

Columns 664 through 676

   2     1     2     2     2     2     1     2     2     2     1     2     2

Columns 677 through 689

   1     2     1     1     1     1     1     1     1     1     1     1     1

Columns 690 through 702

   1     1     1     1     1     1     1     1     1     1     1     1     1

Columns 703 through 715

   1     1     1     2     2     2     2     2     1     1     1     1     1

Columns 716 through 728

   1     1     2     2     2     2     2     2     2     1     1     1     1

Columns 729 through 741

   1     2     2     2     2     2     2     2     2     1     1     1     2

Columns 742 through 754

   2     2     2     2     2     2     2     2     1     1     2     2     2

Columns 755 through 767

   2     2     2     2     2     2     1     1     2     2     2     2     2

Columns 768 through 780

   2     2     2     1     2     2     2     2     2     2     2     2     1

Columns 781 through 793

   2     2     2     2     2     2     2     1     2     2     2     2     2

Columns 794 through 806

   2     1     2     2     2     2     2     1     2     2     2     2     1
```

```
Columns 807 through 819

   2     2     2     1     2     2     1     2     1     1     1     1     1

Columns 820 through 832

   1     1     1     1     1     1     1     1     1     1     1     1     1

Columns 833 through 845

   1     1     1     1     1     1     1     1     1     1     1     1     1

Columns 846 through 858

   1     1     2     1     1     1     1     1     1     1     1     1     1

Columns 859 through 871

   1     1     2     2     2     1     1     1     1     1     1     1     1

Columns 872 through 884

   1     2     2     2     2     2     1     1     1     1     1     1     1

Columns 885 through 897

   2     2     2     2     2     2     1     1     1     1     1     2     2

Columns 898 through 910

   2     2     2     2     2     1     1     1     1     2     2     2     2

Columns 911 through 923

   2     2     2     1     1     1     2     2     2     2     2     2     2

Columns 924 through 936

   1     1     2     2     2     2     2     2     2     1     2     2     2

Columns 937 through 949

   2     2     2     2     1     2     2     2     2     2     2     1     2

Columns 950 through 962
```

```
   2     2     2     2     1     2     2     2     2     1     2     2     2

Columns 963 through 975

   1     2     2     1     2     1     1     1     1     1     1     1     1

Columns 976 through 988

   1     1     1     1     1     1     1     1     1     1     1     1     1

Columns 989 through 1001

   1     1     1     1     1     1     1     1     1     1     1     1     1

Columns 1002 through 1014

   1     1     1     1     1     1     1     1     1     1     1     1     1

Columns 1015 through 1027

   1     1     1     1     1     1     1     1     1     1     1     1     1

Columns 1028 through 1040

   1     1     1     1     1     1     1     1     1     1     1     1     1

Columns 1041 through 1053

   1     1     1     1     1     1     1     2     1     1     1     1     1

Columns 1054 through 1066

   1     1     1     1     1     2     2     2     1     1     1     1     1

Columns 1067 through 1079

   1     1     1     2     2     2     2     1     1     1     1     1     1

Columns 1080 through 1092

   2     2     2     2     2     1     1     1     1     1     2     2     2

Columns 1093 through 1105

   2     2     1     1     1     1     2     2     2     2     2     1     1
```

```
Columns 1106 through 1118

   1     2     2     2     2     2     1     1     2     2     2     2     2

Columns 1119 through 1131

   1     2     2     2     2     2     1     2     2     2     2     1     2

Columns 1132 through 1144

   2     2     1     2     2     1     2     1     1     1     1     1     1

Columns 1145 through 1157

   1     1     1     1     1     1     1     1     1     1     1     1     1

Columns 1158 through 1170

   1     1     1     1     1     1     1     1     1     1     1     1     1

Columns 1171 through 1183

   1     1     1     1     1     1     1     1     1     1     1     1     1

Columns 1184 through 1196

   1     1     1     1     1     1     1     1     1     1     1     1     1

Columns 1197 through 1209

   1     1     1     1     1     1     1     1     1     1     1     1     1

Columns 1210 through 1222

   1     1     1     1     1     1     1     1     1     1     1     1     1

Columns 1223 through 1235

   1     1     1     1     1     1     1     1     1     1     1     1     1

Columns 1236 through 1248

   1     1     1     1     1     1     1     1     1     1     1     1     1

Columns 1249 through 1261
```

```
    1     1     1     1     1     1     1     1     1     1     1     1     1

  Columns 1262 through 1274

    1     1     1     1     1     1     1     1     1     1     1     1     1

  Columns 1275 through 1287

    1     1     1     1     1     1     1     1     1     1     1     1     1

  Columns 1288 through 1300

    1     1     1     1     1     2     1     1     1     1     1     1     1

  Columns 1301 through 1313

    2     1     1     1     1     1     2     2     1     1     1     1     2

  Columns 1314 through 1326

    2     1     1     1     2     2     1     1     2     2     1     2     2

  Columns 1327 through 1330

    1     2     1     1
```

14 Appendix C

14.1 Matlab:2-Machine,3-buffer without idleness

```
function[newPolicy] = MDP1(mu,lambda,total,h)
%MDP takes in process rates, arrival rate, MAX buffer sizes, and
%hodling costs and carriest out policy improvement
tic
mu_1=mu(1);mu_2=mu(2);mu_3=mu(3);
h_1=h(1);h_2=h(2);h_3=h(3);
numStates=nchoosek(total+3,3);
```

```
newPolicy=zeros(1,numStates);

n=0;

mex NewMake1.cpp

[r,c,v,costVector]=NewMake1(mu,lambda,total,h,4);

r(r == 0) = [];

c(c == 0) = [];

v(v == 0) = [];

P{1}=sparse(r,c,v);

% we never use this costVector1
[r,c,v,costVector1]=NewMake1(mu,lambda,total,h,5);

r(r == 0) = [];

c(c == 0) = [];

v(v == 0) = [];

P{2}=sparse(r,c,v);
costVector(costVector==0) = [];

costVector = [0,costVector];

%%%%%%%%%%%%%%%%%%UNIFORMIZE%%%%%%%%%%%%%%%%%%%%%%%%%%%%
for i =1:2

    %fill diagonal

    P{i}=P{i}+diag(-(sum(P{i}(:,:),2)));

%    P{i}=P{i}/(lambda+max(mu_1,mu_3)+mu_2);

%    P{i}=speye(numStates)+P{i};

end

%lets always begin with the LBFS policy (3,2)
```

```
oldPolicy=2*(ones(1,numStates));

while(sum(oldPolicy==newPolicy)~=length(oldPolicy))

    %count number of iterations.

    n=n+1;

    if sum(newPolicy)~=0

        oldPolicy=newPolicy;

    end

    %store policies

    policies{n,1}=oldPolicy;

    %based on a, construct Pa (The policy determines the policy transition

    %matrix)

    Pa=spdiags((oldPolicy==1)',0,numStates, numStates)*P{1}+spdiags((oldPolicy==2)',0,numStates,numS

    %subtract from identity

%    Pa=speye(numStates)-Pa;

    %replace first column by all one's

    Pa= -Pa;
    Pa(:,1)=1;

%    solve the system to produce the relative value functio;

    if n>1

    RVF = bicgstab(Pa,costVector',1e-6,10000,[],[],policies{n-1,2});

    else

    RVF = bicgstab(Pa,costVector',1e-6,10000);

    end

    %store the long run cost next to the appropriate policy
```

```
    policies{n,2}=RVF;

    %becuase we have stored this RVF, we can replace the long run average cost (the top value)
    RVF(1)

    RVF(1)=0;

    %POLICY IMPROVEMENT ALGORITHM

    %start with creating a matrix...each row of the matrix is a state and each

    %number in that row is the 'cost' of this state under each different

    %action.  So, for example, if theres 4 states and 2 different actions, this matrix will

    %be a 4 by 2, 4 rows for 4 states, and 2 different costs per state


%    b(1,:)=costVector(:)+(P{1}*RVF);
    b(1,:)=(P{1}*RVF);

%    b(2,:)=costVector(:)+(P{2}*RVF);
    b(2,:)=(P{2}*RVF);


    %now we go through each row and see which actions is the cheapest cost/highest profit

    %by analyzing the matrix row by row.  In order to analyze each row, we must go

    %to each value in the rows and see which one is smallest.  We then take the

    %column index of the cheapest in each row to be the 'action' in each state.

    %In the end you will see us compare this new policy c with the old

    %policy


    [newPolicy,newPolicy]=min(b);

end
toc
```

```
end
```

14.2 Matlab:2-Machine,3-buffer with idleness

```
function[newPolicy] = MDP1_Idle(mu,lambda,total,h)

tic
mu_1=mu(1);mu_2=mu(2);mu_3=mu(3);

h_1=h(1);h_2=h(2);h_3=h(3);

numStates=nchoosek(total+3,3);

newPolicy=zeros(1,numStates);

n=0;


%MDP takes in process rates, arrival rate, MAX buffer sizes, and

%hodling costs and carriest out policy improvement


mex NewMake1_Idle.cpp

for i=1:5

    [r,c,v,costVector] = NewMake1_Idle(mu,lambda,total,h,i);

    r(r == 0) = [];

    c(c == 0) = [];

    v(v == 0) = [];

    % Insert one element into the sparse matrix to make dimension match

    if i<4
        r = [r,numStates];
        c = [c,numStates];
        v = [v,0];
    end
```

```
    P{i}=sparse(r,c,v);
end
costVector(costVector==0) = [];
costVector = [0,costVector];
%%%%%%%%%%%%%%%%%%UNIFORMIZE%%%%%%%%%%%%%%%%%%%%%%%%%%%%
for i =1:5
    %fill diagonal
    P{i}=P{i}+diag(-(sum(P{i}(:,:),2)));

    P{i}=P{i}/(lambda+max(mu_1,mu_3)+mu_2);

    P{i}=speye(numStates)+P{i};

end
%lets always begin with the LBFS policy (3,2)
oldPolicy=5*(ones(1,numStates));

while(sum(oldPolicy==newPolicy)~=length(oldPolicy))
    %count number of iterations.
    n=n+1;

    if sum(newPolicy)~=0
        oldPolicy=newPolicy;
    end

    %store policies
```

```
policies{n,1}=oldPolicy;


%based on a, construct Pa (The policy determines the policy transition

%matrix)

Pa=sparse(numStates,numStates);

for i=1:5

    Pa = Pa + spdiags((oldPolicy==i)',0,numStates, numStates)*P{i};

end
%subtract from identity

Pa=speye(numStates)-Pa;
%Pa=-Pa;

%replace first column by all one's

Pa(:,1)=1;


%solve the system to produce the relative value functio;

if n>1

RVF = bicgstab(Pa,costVector',1e-6,10000,[],[],policies{n-1,2});

else

RVF = bicgstab(Pa,costVector',1e-6,10000);

end


%store the long run cost next to the appropriate policy

policies{n,2}=RVF;



%becuase we have stored this RVF, we can replace the long run average cost (the top
%value)
RVF(1)
```

```
    RVF(1)=0;

    %may not need, delete if ya dont
    %h(1)=0;
    %h(2:length(RVF))=RVF(2:length(RVF));

    %POLICY IMPROVEMENT ALGORITHM
    %start with creating a matrix...each row of the matrix is a state and each
    %number in that row is the 'cost' of this state under each different
    %action.  So, for example, if theres 4 states and 2 different actions, this matrix will
    %be a 4 by 2, 4 rows for 4 states, and 2 different costs per state

    for i=1:5
        b(i,:)=costVector(:)+(P{i}*RVF);
    end

    %now we go through each row and see which actions is the cheapest cost/highest profit
    %by analyzing the matrix row by row.  In order to analyze each row, we must go
    %to each value in the rows and see which one is smallest.  We then take the
    %column index of the cheapest in each row to be the 'action' in each state.
    %In the end you will see us compare this new policy c with the old
    %policy

    [newPolicy,newPolicy]=min(b,[],1);
end
```

```
%policies{n,2}

toc



end
```

14.3 C++:2-Machine,3-buffer without idleness

```
#include "mex.h"
#include <vector>
using namespace std;
int Index(int,int,int);

int Index(int i, int j, int k)
{
    int p;
    int m;
    m = i+j+k;
    p = ((m+2)*(m+1)*(m)/6)+k*(2*m+3-k)/2+j+1;
    return p;
}

void MakeGenerators(vector<double> mu, double lambda, int total, int action, vector<double> h, doubl

{

int i,j,k;
int ell=0;
    int m=0;
    int q=0;

for(i=0; i<total+1; i++){

for(j=0; j<(total+1-i); j++){

for(k=0; k<(total+1-i-j); k++){

                m=i+j+k;

/* ARRIVAL */

if(m<total){
```

```
ell++;

r[ell] = Index(i,j,k);

c[ell] = Index(i+1,j,k);

v[ell] = lambda;

}


/* ACTIOn (1/2) */

if(action ==4){

if(i>0){

ell++;

r[ell] = Index(i,j,k);

c[ell] = Index(i-1,j+1,k);

v[ell] = mu[0];

}

else{

if(k>0){

ell++;

r[ell] = Index(i,j,k);

c[ell] = Index(i,j,k-1);

v[ell] = mu[2];

}

}


if(j > 0){
```

```
ell++;

r[ell] = Index(i,j,k);

c[ell] = Index(i,j-1,k+1);

v[ell] = mu[1];

}

}

/* END ACTION (1/2) */

/* ACTION (3/2) */

if(action == 5){

if(k>0){

ell++;

r[ell] = Index(i,j,k);

c[ell] = Index(i,j,k-1);

v[ell] = mu[2];

}

else{

if(i > 0){

ell++;

r[ell] = Index(i,j,k);

c[ell] = Index(i-1,j+1,k);

v[ell] = mu[0];

}

}
```

```
if(j > 0){

ell++;

r[ell] = Index(i,j,k);

c[ell] = Index(i,j-1,k+1);

v[ell] = mu[1];

}

}

/* END ACTION (3/2) */

            q=Index(i,j,k);
            costVector[q-1] = i*h[0] + j*h[1] + k*h[2];
}

}

}

}


/* the gateway function */

void mexFunction( int nlhs, mxArray *plhs[],

 int nrhs, const mxArray *prhs[])

{

double *r,*c,*v ;
    double *costVector ;
double lambda;
vector<double> mu(3);
int total;
vector<double> h(3);
    int action;

/*  check for proper number of arguments */
```

```
/* NOTE: You do not need an else statement when using mexErrMsgTxt

within an if statement, because it will never get to the else

statement if mexErrMsgTxt is executed. (mexErrMsgTxt breaks you out of

the MEX-file) */

if(nrhs!=5)

mexErrMsgTxt("Five inputs required.");

if(nlhs!=4)

mexErrMsgTxt("Four output required.");


/* check to make sure the first input argument is a scalar */

if( !mxIsDouble(prhs[0]) || mxIsComplex(prhs[0]) ||

mxGetN(prhs[0])*mxGetM(prhs[0])!=3 ) {

mexErrMsgTxt("Input x must be a scalar.");

}

mu.assign(mxGetPr(prhs[0]), mxGetPr(prhs[0])+3);
lambda = mxGetScalar(prhs[1]);
total=mxGetScalar(prhs[2]);
h.assign(mxGetPr(prhs[3]),mxGetPr(prhs[3])+3);
action = (int)  mxGetScalar(prhs[4]);

/*  set the output pointer to the output vector r */

plhs[0] = mxCreateDoubleMatrix(1,(3 * (total+3)*(total+2)*(total+1)/6),mxREAL);

/*  set the output pointer to the output vector c */

plhs[1] = mxCreateDoubleMatrix(1,(3 * (total+3)*(total+2)*(total+1)/6),mxREAL);

/*  set the output pointer to the output vector v */

plhs[2] = mxCreateDoubleMatrix(1,(3 * (total+3)*(total+2)*(total+1)/6),mxREAL);
```

```
    /*  set the output pointer to the output vector costVector    */

    plhs[3] = mxCreateDoubleMatrix(1, ((total+3)*(total+2)*(total+1)/6),mxREAL);


/*  create a C pointer to a copy of the output matrix */

r = mxGetPr(plhs[0]);

/*  create a C pointer to a copy of the output matrix */

c = mxGetPr(plhs[1]);

/*  create a C pointer to a copy of the output matrix */

v = mxGetPr(plhs[2]);

    costVector = mxGetPr(plhs[3]);

/*  call the C subroutine */

MakeGenerators(mu, lambda, total, action, h, r,c,v, costVector);

}
```

14.4 C++:2-Machine,3-buffer with idleness

```
#include "mex.h"
#include <vector>
using namespace std;
int Index(int,int,int);

int Index(int i, int j, int k)
{
    int p;
    int m;
    m = i+j+k;
    p = ((m+2)*(m+1)*(m)/6)+k*(2*m+3-k)/2+j+1;
    return p;
}

void MakeGenerators(vector<double> mu, double lambda, int total, int action, vector<double> h, doubl
```

```
{

int i,j,k;
int ell=0;
    int m=0;
    int q=0;

for(i=0; i<total+1; i++){

for(j=0; j<(total+1-i); j++){

for(k=0; k<(total+1-i-j); k++){

                m=i+j+k;

/* ARRIVAL */

if(m<total){

ell++;

r[ell] = Index(i,j,k);

c[ell] = Index(i+1,j,k);

v[ell] = lambda;

}


/* ACTIOn (1/0) */

if(action ==1){

if((i>0)){

ell++;

r[ell] = Index(i,j,k);

                        c[ell] = Index(i-1,j+1,k);

v[ell] = mu[0];

}
```

```
else{
if(k>0){
ell++;
r[ell] = Index(i,j,k);
                              c[ell] = Index(i,j,k-1);
v[ell] = mu[2];
}
}
}
/* END ACTION (1/0) */

/* ACTION (3/0) */
if(action == 2){

if(k>0){
ell++;
r[ell] = Index(i,j,k);
                          c[ell] = Index(i,j,k-1);
v[ell] = mu[2];
}
else{
if(i > 0){
ell++;
r[ell] = Index(i,j,k);
```

```
c[ell] = Index(i-1,j+1,k);

v[ell] = mu[0];

}

}

}

/* END ACTION (3/0) */


/* ACTION (0/2) */

if(action ==3){


if(j > 0){

ell++;

r[ell] = Index(i,j,k);

c[ell] =Index(i,j-1,k+1);

v[ell] = mu[1];

}

}

  /* END ACTION (0/2) */


/* ACTIOn (1/2) */

if(action ==4){


if(i>0){

ell++;

r[ell] = Index(i,j,k);
```

```
c[ell] = Index(i-1,j+1,k);

v[ell] = mu[0];

}

else{

if(k>0){

ell++;

r[ell] = Index(i,j,k);

c[ell] = Index(i,j,k-1);

v[ell] = mu[2];

}

}


if(j > 0){

ell++;

r[ell] = Index(i,j,k);

c[ell] = Index(i,j-1,k+1);

v[ell] = mu[1];

}

}

/* END ACTION (1/2) */


/* ACTION (3/2) */

if(action == 5){
```

```
if(k>0){

ell++;

r[ell] = Index(i,j,k);
c[ell] = Index(i,j,k-1);

v[ell] = mu[2];

}

else{

if(i > 0){

ell++;

r[ell] = Index(i,j,k);

c[ell] = Index(i-1,j+1,k);

v[ell] = mu[0];

}

}



if(j > 0){

ell++;

r[ell] = Index(i,j,k);

c[ell] = Index(i,j-1,k+1);

v[ell] = mu[1];

}

}

/* END ACTION (3/2) */
```

```
            q = (m+2)*(m+1)*(m)/6+k*(2*m+3-k)/2+j+1;
            costVector[q-1] = i*h[0] + j*h[1] + k*h[2];

}

}

}

}
```

```
/* the gateway function */
void mexFunction( int nlhs, mxArray *plhs[],

 int nrhs, const mxArray *prhs[])

{

double *r,*c,*v ;
    double *costVector ;
double lambda;
vector<double> mu(3);
int total;
vector<double> h(3);
    int action;


/*  check for proper number of arguments */

/* NOTE: You do not need an else statement when using mexErrMsgTxt

within an if statement, because it will never get to the else

statement if mexErrMsgTxt is executed. (mexErrMsgTxt breaks you out of

the MEX-file) */
if(nrhs!=5)

mexErrMsgTxt("Five inputs required.");

if(nlhs!=4)
```

```
mexErrMsgTxt("Four output required.");


/* check to make sure the first input argument is a scalar */

if( !mxIsDouble(prhs[0]) || mxIsComplex(prhs[0]) ||

mxGetN(prhs[0])*mxGetM(prhs[0])!=3 ) {

mexErrMsgTxt("Input x must be a scalar.");
    }



mu.assign(mxGetPr(prhs[0]), mxGetPr(prhs[0])+3);
lambda = mxGetScalar(prhs[1]);
total=mxGetScalar(prhs[2]);
h.assign(mxGetPr(prhs[3]),mxGetPr(prhs[3])+3);
action = (int)  mxGetScalar(prhs[4]);


/*  set the output pointer to the output vector r */

plhs[0] = mxCreateDoubleMatrix(1,(4 * (total+3)*(total+2)*(total+1)/6),mxREAL);

/*  set the output pointer to the output vector c */

plhs[1] = mxCreateDoubleMatrix(1,(4 * (total+3)*(total+2)*(total+1)/6),mxREAL);

/*  set the output pointer to the output vector v */

plhs[2] = mxCreateDoubleMatrix(1,(4 * (total+3)*(total+2)*(total+1)/6),mxREAL);

    /*  set the output pointer to the output vector costVector    */

    plhs[3] = mxCreateDoubleMatrix(1,(4 * (total+3)*(total+2)*(total+1)/6),mxREAL);



/*  create a C pointer to a copy of the output matrix */

r = mxGetPr(plhs[0]);

/*  create a C pointer to a copy of the output matrix */
```

```
c = mxGetPr(plhs[1]);

/*  create a C pointer to a copy of the output matrix */

v = mxGetPr(plhs[2]);

    costVector =mxGetPr(plhs[3]);

/*  call the C subroutine */

MakeGenerators(mu, lambda, total, action, h, r,c,v, costVector);



}
```

14.5 Utilization calculation

```
tic
Pa=sparse(numStates,numStates);
    for i=1:2

        Pa = Pa + spdiags((newPolicy==i)',0,numStates, numStates)*P{i};

    end

   Pa=Pa+diag(-(sum(Pa')));


Pa=transpose(Pa);
[a,b]=spspaces(Pa,2,1e-8);
pie=b{1}(:,b{3});
pie=pie/sum(pie);



%utilization for machine 1
total = ceil(exp(1/3*log(length(newPolicy)*6)))-2;
getIndex1=[];
for i=0:total
    for j=0:total-i
        for k=0:total-i-j
            m=i+j+k;
```

```
if(i~=0 || k~=0)
    getIndex1=[getIndex1 (m+2)*(m+1)*(m)/6+k*(2*m+3-k)/2+j+1];
else
   getIndex1=getIndex1;
end
        end
    end

end
util1=sum(pie(getIndex1))




%utilization for machine 2
getIndex2=[];
for i=0:total
    for j=0:total-i
        for k=0:total-i-j
            m=i+j+k;
if j~=0
    getIndex2=[getIndex2 (m+2)*(m+1)*(m)/6+k*(2*m+3-k)/2+j+1];
else
   getIndex2=getIndex2;
end

        end
    end

end
util2=sum(pie(getIndex2))



toc




end
```

14.6 Nullspace solver part a

```
function [SpLeft, SpRight] = spspaces(A,opt,tol)
%  PURPOSE: finds left and right null and range space of a sparse matrix A
%
```

```
% ------------------------------------------------
%  USAGE: [SpLeft, SpRight] = spspaces(A,opt,tol)
%
%  INPUT:
%       A                           a sparse matrix
%       opt                         spaces to calculate
%                                   = 1: left null and range space
%                                   = 2: right null and range space
%                                   = 3: both left and right spaces
%       tol                         uses the tolerance tol when calculating
%                                   null subspaces (optional)
%
%   OUTPUT:
%       SpLeft                      1x4 cell. SpLeft = {} if opt =2.
%           SpLeft{1}               an invertible matrix Q
%           SpLeft{2}               indices, I, of rows of the matrix Q that
%                                   span the left range of the matrix A
%           SpLeft{3}               indices, J, of rows of the matrix Q that
%                                   span the left null space of the matrix A
%                                   Q(J,:)A = 0
%           SpLeft{4}               inverse of the matrix Q
%       SpRight                     1x4 cell. SpRight = {} if opt =1.
%           SpLeft{1}               an invertible matrix Q
%           SpLeft{2}               indices, I, of rows of the matrix Q that
%                                   span the right range of the matrix A
%           SpLeft{3}               indices, J, of rows of the matrix Q that
%                                   span the right null space of the matrix A
%                                   AQ(:,J) = 0
%           SpLeft{4}               inverse of the matrix Q
%
%   COMMENTS:
%       uses luq routine, that finds matrices L, U, Q such that
%
%           A = L | U 0 | Q
%                 | 0 0 |
%
%       where L, Q, U are invertible matrices, U is upper triangular. This
%       decomposition is calculated using lu decomposition.
%
%       This routine is fast, but can deliver inaccurate null and range
%       spaces if zero and nonzero singular values of the matrix A are not
%       well separated.
%
%   WARNING:
%       right null and rang space may be very inaccurate
%
```

```
% Copyright  (c) Pawel Kowal (2006)
% All rights reserved
% LREM_SOLVE toolbox is available free for noncommercial academic use only.
% pkowal3@sgh.waw.pl

if nargin<3
    tol                 = max(max(size(A)) * norm(A,1) * eps,100*eps);
end

switch opt
    case 1
        calc_left       = 1;
        calc_right      = 0;
    case 2
        calc_left       = 0;
        calc_right      = 1;
    case 3
        calc_left       = 1;
        calc_right      = 1;
end

[L,U,Q]                 = luq(A,0,tol);

if calc_left
    if ~isempty(L)
        LL              = L^-1;
    else
        LL              = L;
    end
    S                   = max(abs(U),[],2);
    I                   = find(S>tol);
    if ~isempty(S)
        J               = find(S<=tol);
    else
        J               = (1:size(S,1))';
    end
    SpLeft              = {LL,I,J,L};
else
    SpLeft              = {};
end
if calc_right
    if ~isempty(Q)
        QQ              = Q^-1;
    else
        QQ              = Q;
    end
```

```
    S                   = max(abs(U),[],1);
    I                   = find(S>tol);
    if ~isempty(S)
        J               = find(S<=tol);
    else
        J               = (1:size(S,2))';
    end
    SpRight             = {QQ,I,J,Q};
else
    SpRight             = {};
end
```

14.7 Nullspace solver part b

```
function [L,U,Q] = luq(A,do_pivot,tol)
%  PURPOSE: calculates the following decomposition
%
%       A = L |Ubar  0 | Q
%             |0     0 |
%
%       where Ubar is a square invertible matrix
%       and matrices L, Q are invertible.
%
% ------------------------------------------------
%  USAGE: [L,U,Q] = luq(A,do_pivot,tol)
%  INPUT:
%         A             a sparse matrix
%         do_pivot      = 1 with column pivoting
%                       = 0 without column pivoting
%         tol           uses the tolerance tol in separating zero and
%                       nonzero values
%
%   OUTPUT:
%         L,U,Q          matrices
%
%   COMMENTS:
%         based on lu decomposition
%
% Copyright  (c) Pawel Kowal (2006)
% All rights reserved
% LREM_SOLVE toolbox is available free for noncommercial academic use only.
% pkowal3@sgh.waw.pl

[n,m]                   = size(A);
```

```
if ~issparse(A)
    A                   = sparse(A);
end

%-------------------------------------------------------------------------
%       SPECIAL CASES
%-------------------------------------------------------------------------
if size(A,1)==0
    L                   = speye(n);
    U                   = A;
    Q                   = speye(m);
    return;
end
if size(A,2)==0
    L                   = speye(n);
    U                   = A;
    Q                   = speye(m);
    return;
end

%-------------------------------------------------------------------------
%       LU DECOMPOSITION
%-------------------------------------------------------------------------
if do_pivot
    [L,U,P,Q]           = lu(A);
    Q                   = Q';
else
    [L,U,P]             = lu(A);
    Q                   = speye(m);
end
p                       = size(A,1)-size(L,2);
LL                      = [sparse(n-p,p);speye(p)];
L                       = [P'*L P(n-p+1:n,:)'];
U                       = [U;sparse(p,m)];

%-------------------------------------------------------------------------
%       FINDS ROWS WITH ZERO AND NONZERO ELEMENTS ON THE DIAGONAL
%-------------------------------------------------------------------------
if size(U,1)==1 || size(U,2)==1
    S                   = U(1,1);
else
    S                   = diag(U);
end
I                       = find(abs(S)>tol);
Jl                      = (1:n)';
Jl(I)                   = [];
```

```
Jq                      = (1:m)';
Jq(I)                   = [];

Ubar1                   = U(I,I);
Ubar2                   = U(Jl,Jq);
Qbar1                   = Q(I,:);
Lbar1                   = L(:,I);

%-------------------------------------------------------------------------
%       ELININATES NONZEZO ELEMENTS BELOW AND ON THE RIGHT OF THE
%       INVERTIBLE BLOCK OF THE MATRIX U
%
%       UPDATES MATRICES L, Q
%-------------------------------------------------------------------------
if ~isempty(I)
    Utmp                = U(I,Jq);
    X                   = Ubar1'\U(Jl,I)';
    Ubar2               = Ubar2-X'*Utmp;
    Lbar1               = Lbar1+L(:,Jl)*X';

    X                   = Ubar1\Utmp;
    Qbar1               = Qbar1+X*Q(Jq,:);
    Utmp                = [];
    X                   = [];
end

%-------------------------------------------------------------------------
%       FINDS ROWS AND COLUMNS WITH ONLY ZERO ELEMENTS
%-------------------------------------------------------------------------
I2                      = find(max(abs(Ubar2),[],2)>tol);
I5                      = find(max(abs(Ubar2),[],1)>tol);

I3                      = Jl(I2);
I4                      = Jq(I5);
Jq(I5)                  = [];
Jl(I2)                  = [];
U                       = [];

%-------------------------------------------------------------------------
%       FINDS A PART OF THE MATRIX U WHICH IS NOT IN THE REQIRED FORM
%-------------------------------------------------------------------------
A                       = Ubar2(I2,I5);

%-------------------------------------------------------------------------
%       PERFORMS LUQ DECOMPOSITION OF THE MATRIX A
%-------------------------------------------------------------------------
```

```
[L1,U1,Q1]              = luq(A,do_pivot,tol);

%-------------------------------------------------------------------------
%       UPDATES MATRICES L, U, Q
%-------------------------------------------------------------------------
Lbar2                   = L(:,I3)*L1;
Qbar2                   = Q1*Q(I4,:);
L                       = [Lbar1 Lbar2 L(:,J1)];
Q                       = [Qbar1; Qbar2; Q(Jq,:)];

n1                      = length(I);
n2                      = length(I3);
m2                      = length(I4);
U                       = [Ubar1 sparse(n1,m-n1);sparse(n2,n1) U1 sparse(n2,m-n1-m2);sparse(n-n1-n2,
```

14.8 Make plot

```
% make plot of the states where we need to take action 1, except for the
% marginal points
function makeplot(in,action)

colors = {'r.' 'm.'  'y.'  'g.' 'b.'};

count = 1;

total = ceil(exp(1/3*log(length(in)*6)))-2;

% i stands for state(ina,inb,inc)
% report the state of each minority action
for i = 0:total
    for j = 0:total-i
        for k = 0:total-i-j

            m = i+j+k;

            p = (m+2)*(m+1)*(m)/6+k*(2*m+3-k)/2+j+1;

            if in(p)==action

                ina(count) = i;

                inb(count) = j;

                inc(count) = k;
```

```
                count = count + 1;
            end

        end
    end
end

plot3(ina, inb, inc, colors{action})

xlabel('buffer 1')

ylabel('buffer 2')

zlabel('buffer 3')

hold on

grid on

end
```