ECE 6101 Project 1 Report

Matt A. Appel[1]

I. INTRODUCTION

The goal of this report is to address the problems given in Project 1. It will entail the creation of the design of 3 distributed random number generators: Uniform, Exponential, Poisson. These random number generators will be used to model an $M/M/1$ Queue for which the arrivals are Poisson with rate λ and the services are Exponential with rate μ. A $M/E_k/1$ Queue will also be modeled in which the service follows a Erlang distribution. All code was implemented and compiled on Ubuntu 18.04. The executable run to create the plots in the results section is called test. run using ./test. Makefile is provided but requires python matlplotlib to be installed for plotting, remove plots otherwise. Check python path to Python.h in Makefile.

The paper will be broken down into the following structure. Section 2 outlines the distribution algorithms created for this project. It provides a short review of each distribution requirement as well as the implementation approach. Section 3 outlines the Queue structure that will be used in this report. It then provides the approach and algorithm used to implement the 2 Queue models: $M/M/1$ and $M/E_k/1$ required for this report. Section 4 then provides solutions to problem 1 through 4 of the project assignment utilizing the algorithms presented in sections 2 and 3. Finally a conclusion is presented.

II. DISTRIBUTED RANDOM NUMBER GENERATORS

This section will total 4 random number generators (rngs) where each distributed rng may utilize another for implementation. To start a base random number generator is needed. This can be provided by most programming languages. For this project a custom rng was created used the multiplicative congruential generator (MCG) algorithm given in [1] as :

$$x_i = a_1 x_{i-1} \bmod m \tag{1}$$

Such that $i = \mathbb{N}^+$ and x_i is the i^{th} number in the sequence and x_0 is defined as the seed. $a_1 \in \mathbb{N}^+$ and $a \in \mathbb{N}^+$ are both arbitrary but can be used to modify the sequence and sequence length. Further details can be explored at [1]. For the remainder of this report the rng given in Equation (1) shall be defined as r and $r(.)$ is the operation used to retrieve a random number from r where . signifies the ability to modify the seed.

This rng will provide a sequence of numbers that is repeatable at the start of every operation. The seed provided at the start of operation determines which sequence will occur. Thus for non-deterministic behavior the seed needs to change at the start of each use. This book also explains in more details some methods for handling the seed [1]. For the purpose of this project the user has access to the seed and can adjust to different values manually before each run. An easy example would be to use the system clock to create a seed.

A. Uniform Distributed Random Variable Generator

A Uniform distribution is required later in the derivation of the exponential distributed random number generator. It is also used to answer question 1 of the project in section 4. This distribution is quite straight forward given r from (1). The uniform distribution used here is given as $U[0,1]$. Therefore for the uniform random variable generator $X \sim U[0,1]$.

$$X(.) = r(.)/m \tag{2}$$

Equation (2) provides the functionality of the uniform generator where . here is defined as the ability to modify the seed at startup. Implemented as class Uniform in ./dists of source code given.

B. Exponential Random Variable Generator

Using Equation (2) a Exponentially distributed random variable generator E is defined using the algorithm provided by [2]. This method is known as the inverse transform technique [2]. Given Equation (2) and $E \sim Exp(\lambda)$ from [2]:

$$E(rate,.) = \frac{-1}{\lambda} ln(1 - X(.)) \tag{3}$$

Where $(rate,.)$ signifies a modular rate that can be set and again seed that can be set at start up. This is used to model the service for some Queues. Implemented as class Exp in ./dists of source code given.

C. Poisson Random Variable Generator

Following the structure of the previous subsections this will outline a Poisson random variable generator. The purpose is to use this to model the packet arrivals in a Queue model. Here knowledge of the Poisson inter-arrival time distribution will be leveraged as in [1] but implementation is slightly different. The approach is best descried through an algorithmic definition rather then Equation. Defined as $P \sim Poisson(\lambda)$, this generator is defined as $P(E(rate,.),T)$.

[1]Matthew A. Appel appel.60@osu.edu

Algorithm 1 Possion Random Variable Generator

```
1: procedure P(E(rate,.),T)
2:     t ← 0
3:     i ← 0
4:     while t < T do
5:         interArrivalTime ← E(rate,.)
6:         t ← t + interArrivalTime
7:         i++
8:     end while
9:     return i − 1
10: end procedure
```

The inter-arrival times can be procedurally generated using $E(rate,.)$. Thus once enough packets have arrived such that the current time in the loop has become larger then the interval T, that index $i-1$ is to be returned accounting for the last generation was outside the interval. Implemented as class Poisson in ./dists of source code given.

III. Queue

Here 2 Queue models will be derived. A $M/M/1$ where the arrivals are Poisson distributed with rate λ and service times are Exponentially distributed with rate μ. Also, a $M/E_k/1$ is model with the same arrival but the service is Erlang in both cases there is one infinite buffer and one server. Both Queues are implemented as clases in ./NetworkQueues. the tick() function is where the following algorithms are implemented.

A. M/M/1

The Queue will be a object class with an internal buffer. This will be updated periodically based on the step interval time. Packets added according to poisson arrival then parsed into and out of the system keeping track of time in the system. The Queue can me visualized as:

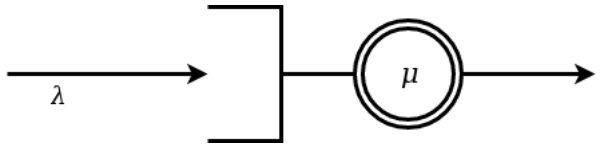

Fig. 1. ∞ Buffer 1 Server Queue

It can be seen that the algorithm needs to utilize $P(E(\lambda,.),T)$ and $E(\mu,.)$ where T can be chosen by the user based on the desired interval. The next few lines are a walk through of the algorithms operation. A packet is defined as having the following structure:

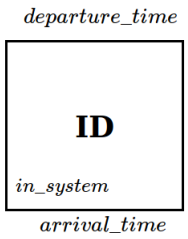

Fig. 2. Packet definition

Where $departuretime$ is the time the packet is set to leave the system, 0 is defined as not yet entered service. $arrivaltime$ is the time the packet is set to arrive into the system. $insystem$ is a flag to notify if the packet is currently in the system.

The algorithm to run the queue is run at each time step. There is a internal buffer that stores the packets that will arrive. At each time step given the simulation progresses by T. First, packets are added to the internal buf for the time interval, following a Poisson distribution, utilizing Algorithm 1. Algorithm 2 is the implementation of the queue object update of new packets arriving. The last packet in the buffer to be served will always be after interval. The reason the last packet after the time interval is not dropped will be shown later, but essentially it allows the algorithm to be ticked manually.

Algorithm 2 Update Queue Internal Buffer

```
1: mm1                                          ▷ the queue object
2: currentp = nextinline
3: procedure New Packets
4:     currenttime ← time
5:     nextarrival ← nextpacket.arrival
6:     while nextarrival < (currenttime + sim_step) do
7:         p = NULL                             ▷ create new packet
8:         delta = E(μ,.)
9:         p.arrival = delta + nextarrival
10:        currentarrival = currentarrival + delta
11:        mm1.add(p)
12:        cp = p
13:    end while
14: end procedure
```

Next, the algorithm iterates over all the packets adding packets to the system, removing packets and adjusting time as necessary. The goal is to put each packet in the system and advance and serve till this occurs or the interval is over. The program keeps internal track of when the next arrival and next departure will happen. Algorithm 3 covers each case when iterating over the packets at each time interval.

Algorithm 3 Update System Using Internal Buffer

```
n = numcustomers
mm1                                              ▷ the queue object
procedure UPDATE
    iterator ← buf.front                         ▷ buf is not packets in queue
    while iterator! = buf.end do                 ▷ iterate over packets
        mm1.arrival = iterator.arrivaltime
        if iteratoris not in system then
            if iterator.service! = 0 then
                if mm1.service > 0 then
                    if arrival before service then
                        add iterator to mm1
                        update length of time n in mm1
                    else
                        serve and update n
                    end if
                end if
            else
                add iterator to mm1
                assign iterator service time
            end if
        end if
        iterator++
    end while
end procedure
```

IV. Results

This section outlines the results of the algorithms developed above. Each distribution is compared to the calculated desired result. For each distribution the generator was run 10000 times. For the uniform distribution we get the following plot:

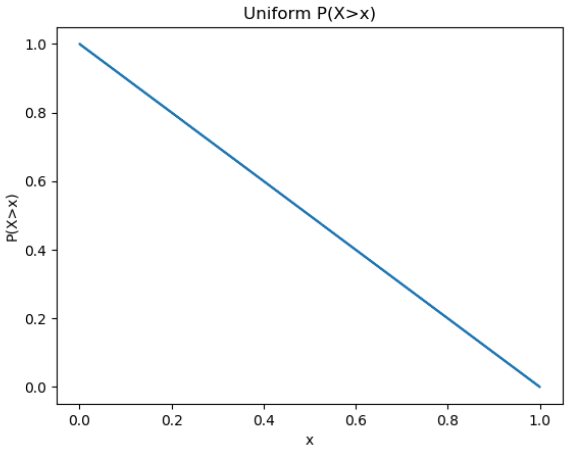

Fig. 3. Uniform Result

For the Poisson($\lambda = 2$) the $P(X > x)$ is given in the Figure below:

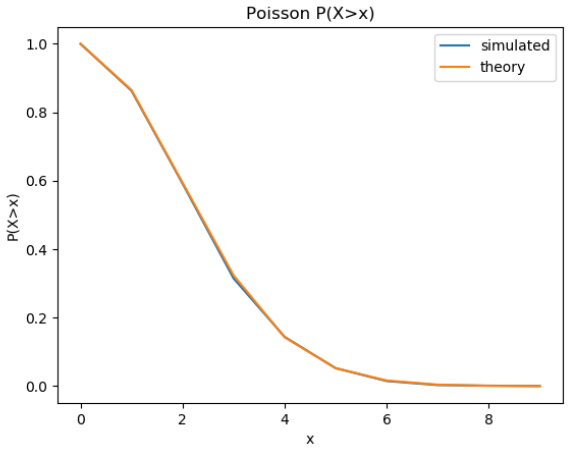

Fig. 4. Poisson result

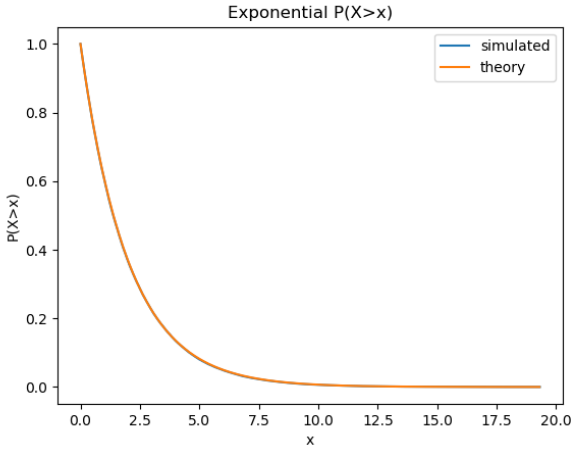

Fig. 5. Exponential result

And finally the Exponential result is given in 5.

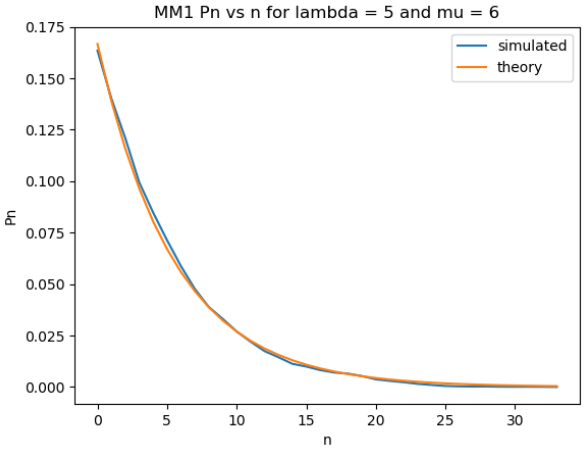

Fig. 6. MM1 result

As for the MM1 Algorithm 2 and Algorithm 3 resulted in the following in which the expectation is blue and the actual is orange in 6. For this simulation the expected delay was calculated to be 0.92691 seconds and the Expected number in the Queue was found to be 4.63724. Both where close to what was expected with some amount of difference due to the randomness of the simulation.

The final step was the Erlang. The same algorithm used for MM1 should work for the MEk1 with an adjustment in the service time distribution. For $k = 4$,$\lambda = 4$,$\mu = 6$ the following resulted:

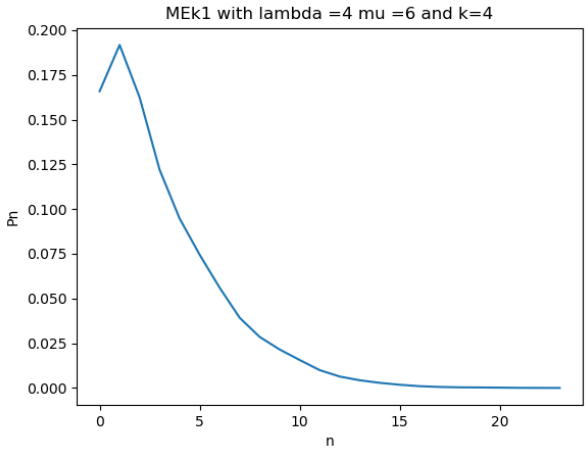

Fig. 7. Exponential result

With and Expectation of delay equal to 3.17489 and expected number in the queue to be0.63474. Showing improved overall performance with a modified behavior of a small bump at lower n.

This last graph shows the $M/E_k/1$ for $\lambda = 1$ to $\lambda = 5.9$ in increments of 0.1. k=40 and $\mu = 6$. This was used to visual the impact on the expected number in the system as the arrival rate increased.

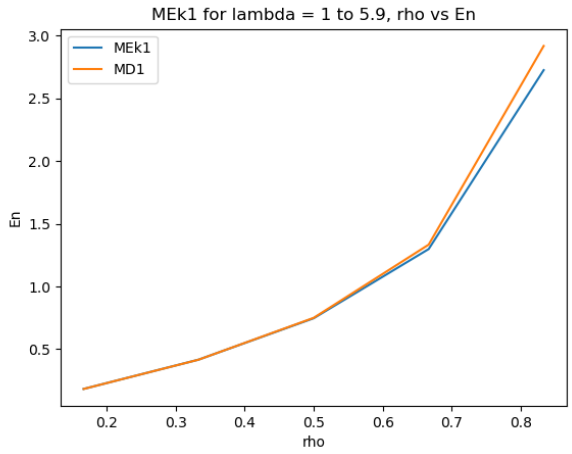

Fig. 8. M/Ek/1 vs M/D/1

Figure 8 shows that the Expected number in the deterministic system appears to be slightly worse as ρ increase to 1. The Theoretic value for the expected number in a deterministic system was found using Pollaczek–Khinchine formula:

$$E(N) = \frac{\rho}{1-\rho} * (1 - \frac{\rho}{2}) \tag{4}$$

It makes sense that as the shape of service distribution gets to be very large the service may start to appear to be more constant. This is easily visualized for plots of Erlang distributions with a large k.

REFERENCES

[1] D. E. Knuth, "The art of computer programming, volume 2, chapter 3," *Preliminary version available at http://www-cs-faculty. stanford. edu/ knuth/news. html*, 1973.

[2] "Generate random numbers according to a given distribution." [Online]. Available: http://www.ece.virginia.edu/ mv/edu/prob/stat/random-number-generation.pdf