Probability Mini Project #1

APMA 3100

Garrett Burroughs

gab8un

October 2022

Contents

1 Model

For this project, we will be simulating the calling process for the representative of a high-speed Internet provider. The simulation will be done with a Monte-Carlo simulation system. Before implementing the system, we must first define the notation and components of the model.

1.1 Notation

For this project we will define the following variables and values:

W - Random variable representing the total time spent spent by the representative on calling one customer

X - Continuous random variable representing the time it takes for the customer to answer the call. X has a mean of 12 seconds and has an exponential distribution

T_d - The time it takes to dial a number. $T_d = 6$ seconds

T_b - the time it takes to detect a busy signal. $T_b = 3$ seconds

T_u - the time it takes to wait for 5 rings (and conclude no that customer is unavailable). $T_u = 25$ seconds

T_e - the time it takes to end the call. $T_e = 1$ second

C_b - The event that the customer is using the line, and therefore busy

C_u - The event that the customer is unavailable, and will therefore not answer the phone.

C_a - The event that the customer is available and will pick up after X seconds

$P[C_b]$ - The probability of event C_b occurring when a call is made. $P[C_b] = 0.2$

$P[C_u]$ - The probability of event C_u occurring when a call is made. $P[C_u] = 0.3$

$P[C_a]$ - The probability of event C_a occurring when a call is made. $P[C_a] = 0.5$

This report also uses the abbreviation PMF when referring to the probability mass function of a random variable, PDF when referring to the probability density function of a random variable, and CDF when referring to the cumulative distribution function of a variable.

1.2 Cumulative Distribution Function

Because X is an exponential random variable, we know that $E[X] = \frac{1}{\lambda} = 12$ so $\lambda = \frac{1}{12}$ and the CDF for a exponential random variable is $F_X(x) = 1 - e^{-\lambda x}$ so:

$$\text{CDF: } F_X(x) = 1 - e^{-\frac{x}{12}} \tag{1}$$

The inverse of the CDF for an exponential is $F_X^{-1}(u) = -\frac{\ln(1-u)}{\lambda}$ so:

$$x = F_X^{-1}(u) = -12\ln(1-u) \tag{2}$$

1.3 Tree Diagram

The events of the calling process can be described by the following flowchart:

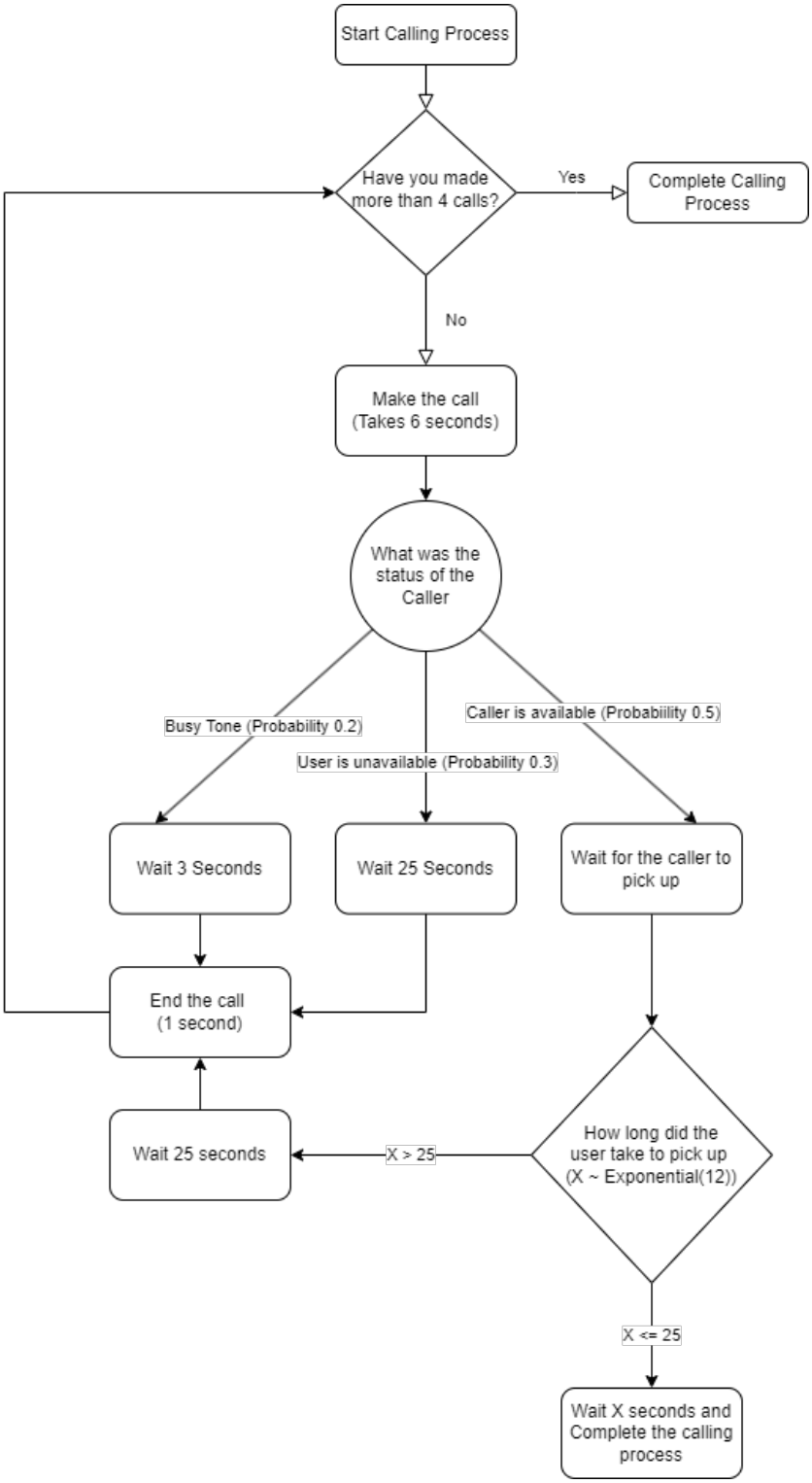

Figure 1: Flowchart describing calling process

2 Data Collection

To collect sample data on this problem, an ad-hoc experiment was run. Using an iPhone 11, calls were made where the participant was told to not answer, where the participant was unknowing of the call (to simulate a random wait time for an available user) and where the line being called was knowingly busy. To simulate dialing numbers, a random phone number was generated, and the time it took to pick up the phone and dial the number was recorded. The results of these experiments can be seen in the table below:

Trial	Dial (T_d)	Unavailable (T_u)	Pick up (X)	Busy (T_b)	End (T_e)
1	12	32	9	3	1
2	10	34	3	2	1
3	11	33	6	2	1
Average	11	33	6	2.33	1

Table 1: Results of Ad-Hoc Calling Process Experiment

3 Monte-Carlo Simulation Algorithm

In order to simulate this problem on a computer, a Monte-Carlo simulation system was used. The Monte-Carlo simulation system used a pseudo-random number generator, a discrete random variable generator, a continuous random variable generator, and implemented the calling process described in section 1.3. All code implementing the simulation system was written in the Rust Programming Language version 1.58.1 (https://www.rust-lang.org/).

3.1 Random Number Generator

For this simulation, a linear congruential random number generator was used. This algorithm uses the following recursive rules to define the output u_i for the ith random number generated based on some initial parameters:

$$x_i = (ax_{i-1} + c) \mod K \tag{3}$$

$$u_i = x_i/K \tag{4}$$

for $i \in \mathbb{Z}^+$

The initial parameters used in this project were

Starting Value	$x_0 = 1000$
Multiplier	$a = 42693$
Increment	$c = 3517$
Modulus	$K = 2^{17}$

Table 2: Random number generator starting values

These values were provided by the project specification, and used to achieve maximum cycle length for the pseudo-random number generation. In the code, this was implemented by creating a Rust struct to store all of the parameters of the algorithm, and calling equations 3 and 4 using those parameters and updating x_i internally, as well as returning the result of u_i.

The results of generating the first 3 random numbers using this algorithm with the mentioned parameters were:

u_1	0.41947174072265625
u_2	0.0425262451171875
u_3	0.12740325927734375

Table 3: First three random numbers generated by pseudo-random number generator

These values match up with the given values in the project specification, leading to the conclusion that the pseudo-random number generator is working correctly. The 51st, 52nd, and 53rd (u_{51}, u_{52}, u_{53}) were also generated:

u_{51}	0.5157089233398438
u_{52}	0.427276611328125
u_{53}	0.7681961059570313

Table 4: 51st, 52nd and 53rd values of the Pseudo-Random number generator

3.2 Random Variable Generators

For this simulation, we needed to create a Discrete Random Variable to represent the different customer availability states, as well as create a continuous random variable to represent how long it takes for an available customer to pick up the phone.

3.2.1 Discrete Random Variable Generator

The following mathematical model was used to implement a discrete random variable generator

Let D be a discrete random variable. Given:

- The PMF of D: $P_D(d)$
- The sample space of D: S_D
- A random or pseudo-random number: u_i

The cumulative distribution function of D is:

$$F(d) = \Sigma_{y \leq d} p(y), d \in S_D \tag{5}$$

And a realization d_i of D given u_i can be generated by:

$$d_i = \min\{d : F(d) \leq u_i\} \tag{6}$$

3.2.2 Discrete Random Variable Generator Implementation Details

A generalized discrete random variable generator was implemented in Rust by creating a struct that stores a reference to a random number generator, the PMF of D, $P_D(d)$, and the sample space of D, S_D. When generating a realization of D, the code generates a random number u_i from the provided random number generator and iterates over the values of $d \in S_D$ until $F_D(d) \geq u_i$, and returns the result d.

3.2.3 Testing Discrete Random Variable Generator

In order to test the discrete random variable generator, an example random variable Y was created with Sample space and PMF

$$S_y = \{1, 2, 3, 4\} \tag{7}$$

$$P_Y(y) = \begin{cases} 0.1 & \text{if } y = 1 \\ 0.2 & \text{if } y = 2 \\ 0.4 & \text{if } y = 3 \\ 0.3 & \text{if } y = 4 \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

This PMF and sample space was used to create a Discrete Random Variable Generator along with the pseudo-random number generator described in section 3.1. This generator was used to generate 10000 realizations of Y. The experimental probabilities were as follows:

P[Y = 1] = 0.1011
P[Y = 2] = 0.2008
P[Y = 3] = 0.3957
P[Y = 4] = 0.3024

These experimental results are consistent with the theoretical probabilities given by the PMF, leading to the conclusion that the algorithm is working properly.

3.2.4 Discrete Random Variable used in the Monte-Carlo simulation system

For this particular simulation system, a discrete random variable was created to represent the different outcomes of the customer availability. The sample space for this random variable was $S_A = \{1, 2, 3\}$ where 1 corresponds to C_u, the event that the customer is unavailable, 2 corresponds to C_b, the event that the customer is busy and 3 corresponds to C_a, the event that the customer is available. The PMF of this variable was defined as:

$$P_A(a) = \begin{cases} 0.2 & \text{if } a = 1 \\ 0.3 & \text{if } a = 2 \\ 0.5 & \text{if } a = 3 \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

3.2.5 Continuous Random Variable Generator

The following mathematical model was used to implement a continuous random variable generator

Let C be a continuous random variable. Given:

- The inverse of the CDF of C: $F_C^{-1}(u)$

A realization c_i of C given u_i can be generated by:

$$c_i = F_C^{-1}(u_i) \tag{10}$$

3.2.6 Continuous Random Variable Generator Implementation Details

A generalized continuous random variable generator was implemented in Rust by creating a struct that contained a reference to a random number generator, and a closure that contains the operation to perform the inverse CDF F^{-1}. When generating a realization of the random variable, the code generated a random number u_i from the random number generator, and then calculated $c_i = F^{-1}(u_i)$ by calling the closure with u_i as input and returning the result.

3.2.7 Testing of a continuous random variable

In order to test the continuous random variable generator, a random variable Z was created to test a continuous exponential random variable. Z is an exponential random variable with a mean of 12. The CDF and inverse CDF of Z is the same of that for X described in equations 1 and 2. The expected CDF and experimental CDF were calculated and compared. The experimental CDF was calculated by generating 10000 realizations of Z and then counting the number of non exceedance events ($P[Z \leq z]$) for the desired value of the CDF, and then dividing by the number of trials (in this case 10000). The CDF was tested at 1, 2, 6, 12, and 24. The results were:

- CDF of Z for 1, Expected: 0.07995558537067671, Actual: 0.0803
- CDF of Z for 2, Expected: 0.15351827510938587, Actual: 0.1529
- CDF of Z for 6, Expected: 0.3934693402873666, Actual: 0.3964
- CDF of Z for 12, Expected: 0.6321205588285577, Actual: 0.6311
- CDF of Z for 24, Expected: 0.8646647167633873, Actual: 0.864

These results all fall within a reasonable range of the expected value. This leads to the conclusion that the continuous random variable generator is working as intended.

3.2.8 Continuous Random Variable used in the Monte-Carlo simulation system

For this particular simulation system a continuous random variable X was created to represent the time it takes for the user to pick up the phone when available. X is an exponential random variable with a mean of 12. For the program, the inverse CDF for X that was derived in equation 2 was used.

3.3 Simulation System

To create a simulation system that modeled the calling process, a Rust struct was created that accepted all of the parameters defined in section 1.1. When running a simulation of the calling process, the process explained in the tree diagram in section 1.3 was used. The control flow of the program was coded using conditionals, loops, and the random variable generators explained in the previous sections of this report.

3.3.1 Testing Simulation System

In order to test the simulation system, an option for the calling process was coded to output each step of the process, as well as how long it took, and the variables being generated. While this was not able to rigorously test the probabilities of any one event occurring in the calling process, it was able to test if the operations were occurring as they should. This test was able to verify that there were 3 different availability states of the customer, and that the correct amount of time was waited in each of the scenarios.

3.4 Code details and Archive

As mentioned the code for this algorithm was written in Rust and allowed for a generalized Monte-Carlo simulation system to be implemented, as well as provided the specific implementation for the calling process described in section 1.3. The random number and variable generator code was split up into modules for the random number generator, the discrete random variable generator, and the continuous random variable generator. These implementations follow the descriptions in the previous sections, and are generalized and not specific to anything involving the calling process. An implementation of the calling process was created in the calling process module. The calling process module is where the concrete implementations of the random variable generation lies, as well as the other logic to conduct the calling process. Unit testing was implemented where applicable, however, comprehensive testing was difficult due to the random nature of the output. Most testing was done experimentally, and verified manually. An archive of this code can be seen at:
https://github.com/GarrettBurroughs/MonteCarloSimulation

4 Simulation

In order to run the simulation, a for loop was used to run the calling process 1000 times. The same pseudo-random number generator was used in all 1000 trails, and was not reset in between calls to provide for different outputs on all calls. The results of each call was stored in a vector for analysis. The results were also sorted in order of increasing time and written to a CSV file for external analysis. The raw CSV file can be seen at:
https://github.com/GarrettBurroughs/MonteCarloSimulation/blob/master/results.csv

5 Estimations

The following values were calculated after running the simulation, based on the sample of 1000 runs.

Mean	40.871
First Quartile	13.551
Median	30.130
Third Quartile	60.601
$P[W \leq 15]$	0.279
$P[W \leq 20]$	0.39
$P[W \leq 30]$	0.496
$P[W > 40]$	0.42
$P[W > 75]$	0.186
$P[W > 100]$	0.081
$P[W > 125]$	0.022

Table 5: Estimated values from running the simulation 1000 times

The mean was calculated by summing up the total of all values and dividing by the number of results in the population. The first quartile, median, and third quartile were calculated by sorting the results by increasing order and finding the values at the index $\frac{1}{4}$, $\frac{1}{2}$, and $\frac{3}{4}$ of the population respectively. The probabilities $P[W \leq w]$ were found by counting the number of results that were less than or equal to w and dividing the total number of results. Probabilities $P[W > w]$ were calculated by calculating $1 - P[W \leq w]$. The values 75, 100, and 125 were used to show the right tail of the CDF of W because the max time the calling process could take is 128 seconds, in the case that the representative has to wait the full 25 seconds for 5 rings on every call. This would give a calling time of 6 seconds to dial, 25 seconds to wait, and 1 second to hang up, four times in a row or $32 * 4 = 128$.

6 Analysis

6.1 Comparison of the median to the mean

The mean of the distribution (40.871) is much higher than the median (30.130). This means that the average call length is longer than most of the calls that will be made. This indicates that the PDF of the calling process, is skewed towards shorter calls, however, a few events will take far longer than the mean. These fewer number of longer calls are able to increase the mean as compared to the median.

6.2 Determining the sample space for W

To determine a sample space for the mean, we can take a look at the shortest and longest possible calling processes. The best case scenario for any one call is occurs when the customer is available. In the case that the customer is available, the sample space for the amount of

time that the representative waits is $[0, 25]$. In the best case scenario, the representative waits 0 seconds. With a wait time of 0 seconds, this means that the total time to make the call will be 6 seconds, the time it takes to dial the call. After the customer picks up, the calling process is complete so the total time is 6 seconds. The worst case scenario for the waiting time occurs when the customer is unavailable or does not pick up within 25 seconds. These both give a wait time of 25 seconds. With a wait time of 25 seconds, and no response, the total call time is 36 seconds (6 to dial, 25 waiting, 1 to hang up). In the worst case scenario, this can happen a max of 4 times, giving a total calling process wait time of 128 seconds. Because the time waiting on a successful call is X, which is a continuous random variable between 0 and 25, W can take on any value between the max and the min. This means that the sample space of W is $[6, 128]$

6.3 Graphs of the CDF of W

The following graph of the CDF of W can be used by plotting the points of the values of the CDF calculated in section 5.

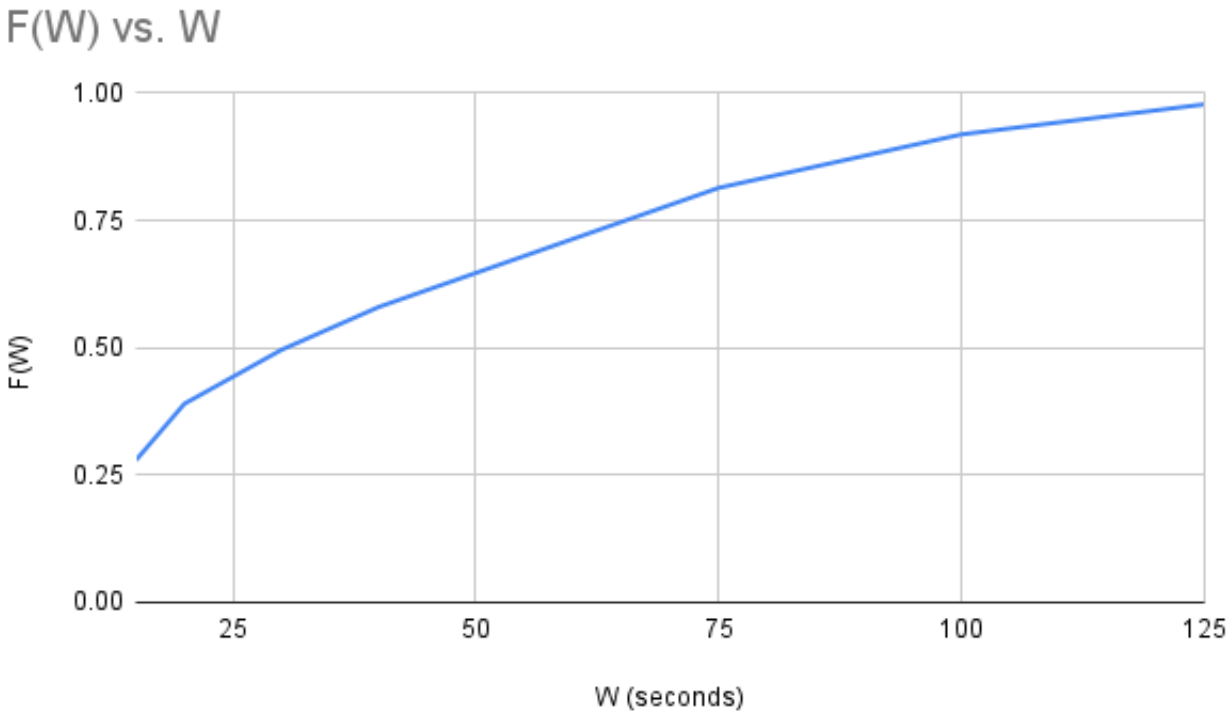

Figure 2: Graph of some calculated experimental values of CDF of W vs values of w

A graph was also generated using all 1000 values of the simulation that were ran.

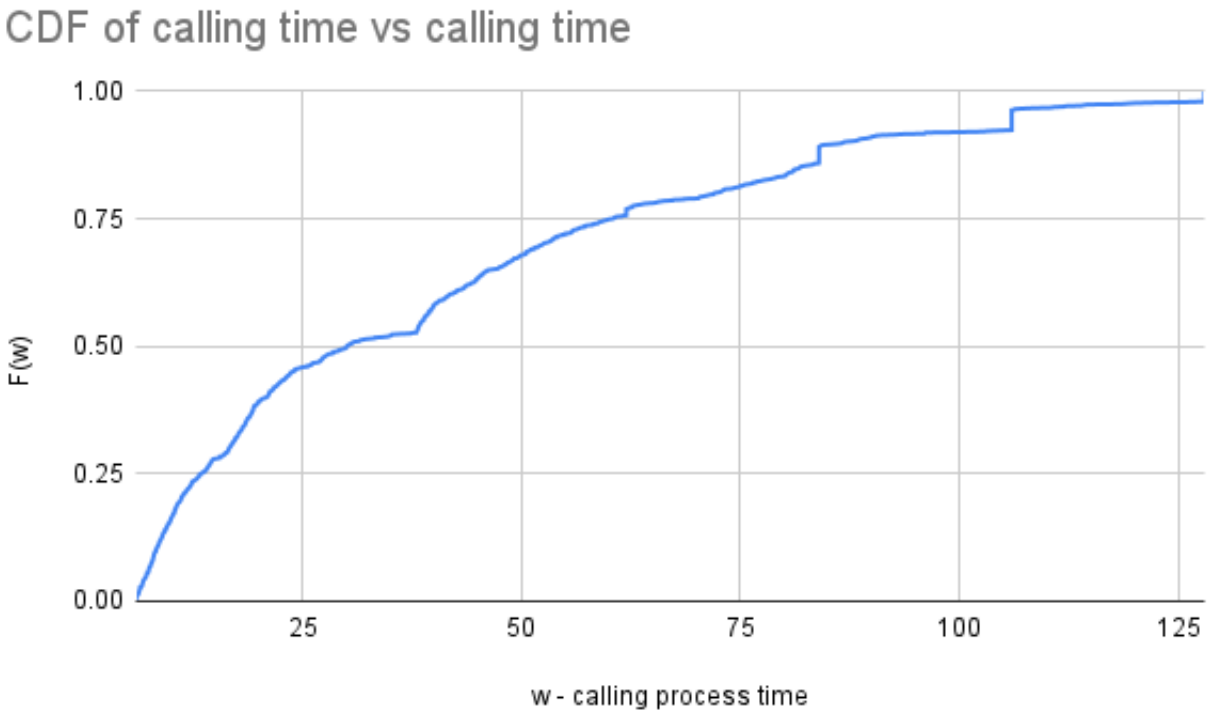

Figure 3: Graph of all calculated experimental values of CDF of W vs values of w

Based on the shape of the graph of the CDF of W, it closely resembles that of an exponential random variable. This is because the largest effect on the wait time (in 50% of the wait times, and the largest variance) is the time to wait for a customer X, which is a random variable. Because of this heavy dependence on X, W can be modeled relatively well as an exponential random variable.

While W, can be modeled relatively closely by an exponential random variable, there are noticeable discontinuities, and deviations from a true exponential random variable. There is a spike in call length at about 38, 62, 84, and 106 seconds. These spikes are most likely caused by the constant wait time needed to dial a phone call and hang up, which are not modeled exponentially. These spikes are also influenced by the nature of the discrete number of calls made (1, 2, 3 or 4), as well as the possibility of a constant waiting time in half of the scenarios (the customer line is busy or unavailable). Due to these deviations from a true exponential random variable, W is not a perfect random variable, even though it can be closely modeled as one.

An archive of the data used for this analysis can be seen at:
https://tinyurl.com/gab8un3100data

7 Comments

Throughout this project, various portions varied in time requirement and level of difficulty. Initial difficulty arose when trying to understand the context of the problem, and all of

the variables involved. There are many different components that needed to be understood in order to implement the simulation algorithm, including understanding how to generate random variables, understanding all variables at play in the simulation, and creating a model for the simulation. Once all the parts were understood, writing the code to simulate and test the system was rather straightforward. Similarly, initially writing the code did not take much time, and was the least time consuming portion of the project, however, testing and figuring out all of the bugs in the code took the longest.

At the conclusion of this project, I am fairly certain (within 90%) that my simulation correctly modeled the calling process, and produced reasonable values. Throughout the test methods described in previous sections, the individual components of the program were able to be tested with high certainty. An overall holistic evaluation of the entire system matches with how the system should act, and there are no reasons that would indicate error in the system. Even though there is a high certainty that the system is implemented and working correctly, it is extremely difficult to verify with 100% certainty that the system is working correctly due to the large number [1] permutations of inputs to the system.

Honor Pledge

On my honor, I pledge that I have neither given nor received aid on this assignment.

_____

[1]Most functions used the f64 type in Rust which has 2^{64} different possible values. Multiple functions within this algorithm use multiple f64 parameters to create their output. This range of permutations is far too large to be reasonably tested and verified