SYSC4005A Discrete Simulation

Milestone **#3**

Project Group #42
Alina Ahmad - 101111867
Scharara Islam - 101149731
Nick Fejes - 101115761

April 5th, 2023

1.0 Problem Formulation

A manufacturing facility deals with the production of three different types of products, P_1, P_2, P_3, and there are two or more components, C_1, C_2, and C_3 for these products. Each product must be assembled at a workshop, W_1, W_2, and W_3 by inspectors I_1, I_2, and I_3. Product P_1 contains one C_1 component, Product P_2 contains two C_1 components, and Product P_3 contains two C_1 components. The pieces are examined, cleaned, and fixed by two inspectors. Inspector 1 is used on C_1 components. The C_2 and C_3 components are handled by Inspector 2 in a random order.

The components are delivered to their appropriate workstations after passing inspection. One buffer is available for each of the required component types, giving each workstation a two-component buffer capacity. The first step in building a product is waiting till all necessary components are accessible. The associated inspector who completed examining a component of a certain kind is deemed "blocked" until there is an opening, at which point the inspector can resume processing and forwarding components of that type if all workstation buffers for that type are empty.

2.0 Setting of Objectives and Overall Plan

2.1 Objective

The objective of this project is to conduct a simulation study in order to determine the efficiency of the manufacturing facility based on collected historical data for workshops and inspectors service times. The presented problem is that inspector 1 needs to distribute C_1 in a way which improves the idle time for inspectors and workshops and also maximizes the system throughput.

2.2 Project Plan

The team plans on completing this project by meeting on Tuesdays and Friday evenings to discuss and distribute work for each Milestone. As a group, we have decided to use the Python language for this project as it was one of the recommendations from the instructor and also because each member has experience with this programming language.

3.0 Model Conceptualization

The following figure shows the relationship between the different parts of the manufacturing facility. The components are available to every part of the facility in infinite quantities as they are always required and used instantaneously. There are two inspectors in this facility and they are always provided the components in order to inspect and then pass them to the buffers in the system. The main purpose of the buffer is to act as a queue which will be given a specific component by one of the inspector and provide this component to a workshop. As shown below, the inspectors have access to any of the five buffers and each buffer is able to hold at most, 2 components. Each workstation can only receive one component at a time from the

buffer and then the workstation takes the component as input and generates output which is the assembled product. The workstations only begin working once all the required components are passing in from the buffers. The components are passed to the buffer which has the same component type but if the buffer is full then the inspectors must be blocked from adding more components. The inspector will keep the components but it will be notified once the buffer of the same component type becomes available again and take the stored component.

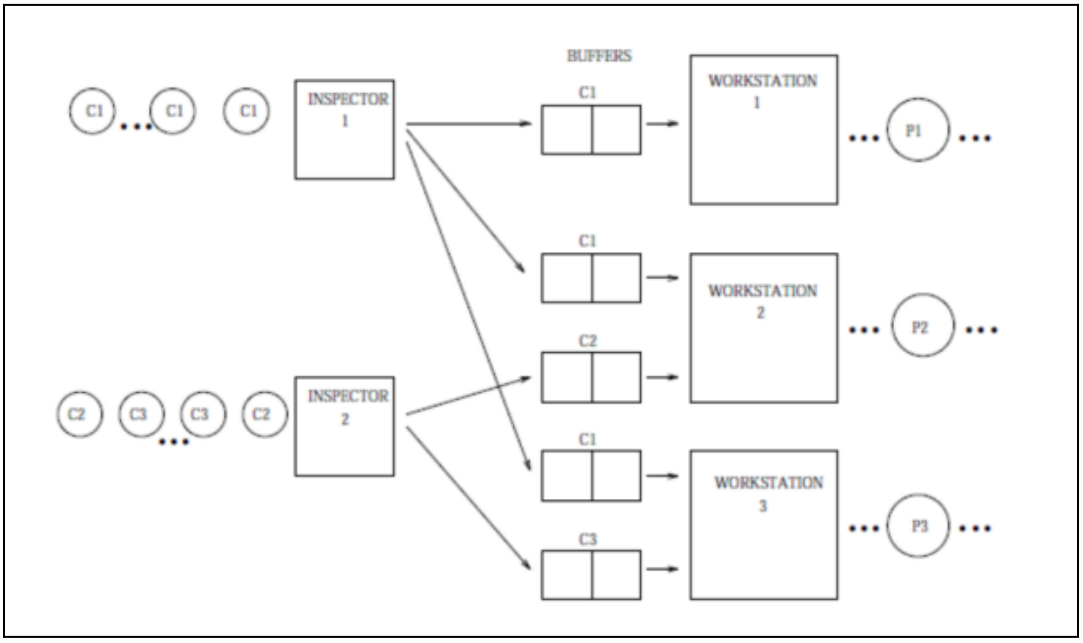


Figure 1. Manufacturing facility

4.0 Model Translation

For this project, the decision was made to use python for the simulation program language. Python has the advantages of being easy to implement multiprocessing, as the multiprocessing library is simple to implement and has many inbuilt functions that make a simulation of multiple parts easier to create. In addition, the library of numpy was used for the math involved in identifying the least full queues.

For this simulation, we have these python files and we will be describing more in detail about what each of these files will be doing and how it is interacting with each other with the diagrams as shown below. The main file are Components.py, Inspector.py, Workshop.py, main.py:

components.py: The components.py file contains all the required functions and classes required to simulate the components used in the system. It contains four classes which define a generic component class, and the three subclasses which define each of the components used in the simulation. One function is defined which is used to supply the inspectors with the required

components. When initialized as a process it is given a type which controls whether it creates type 1 or type 2 / type 3 components.

inspector.py: The inspector.py file contains the two function definitions that are used to create the two inspectors used in the simulation. Both inspectors receive components from their input queues, and then "work" for a random amount of time pulled from the supplied list of execution times. Once the inspector has completed its work it sends the component to the first valid output queue. For the first inspector it selects the least full queue available to send the inspected component to, the second inspector selects the valid queue for the type of component. For both if there is no valid queue to send to the inspector waits until one becomes valid.

workshop.py: The workshop.py file contains three function definitions used to simulate the workshops used in the system. Each workshop waits until it can receive the required components from their input queues, once the needed components are taken from the queue the workshop "works" for a random amount of time from the supplied times given. Once the time is complete the workshop consumes the components used and prints to the terminal that a product has been created.

main.py: The main.py file initializes all needed variables, functions, and processes that are required to run the simulation. It first creates an id list used to keep track of components made during the simulation. Then it creates a Lock object shared by all processes that synchronizes access to the terminal. After that all the queues are initialized and sorted into groups for use by the processes. Then the program initializes all processes in the simulation, two for the component supplies, two for the inspectors, and three for the workshops. Finally the program starts all the processes, and loops while the simulation is ongoing.

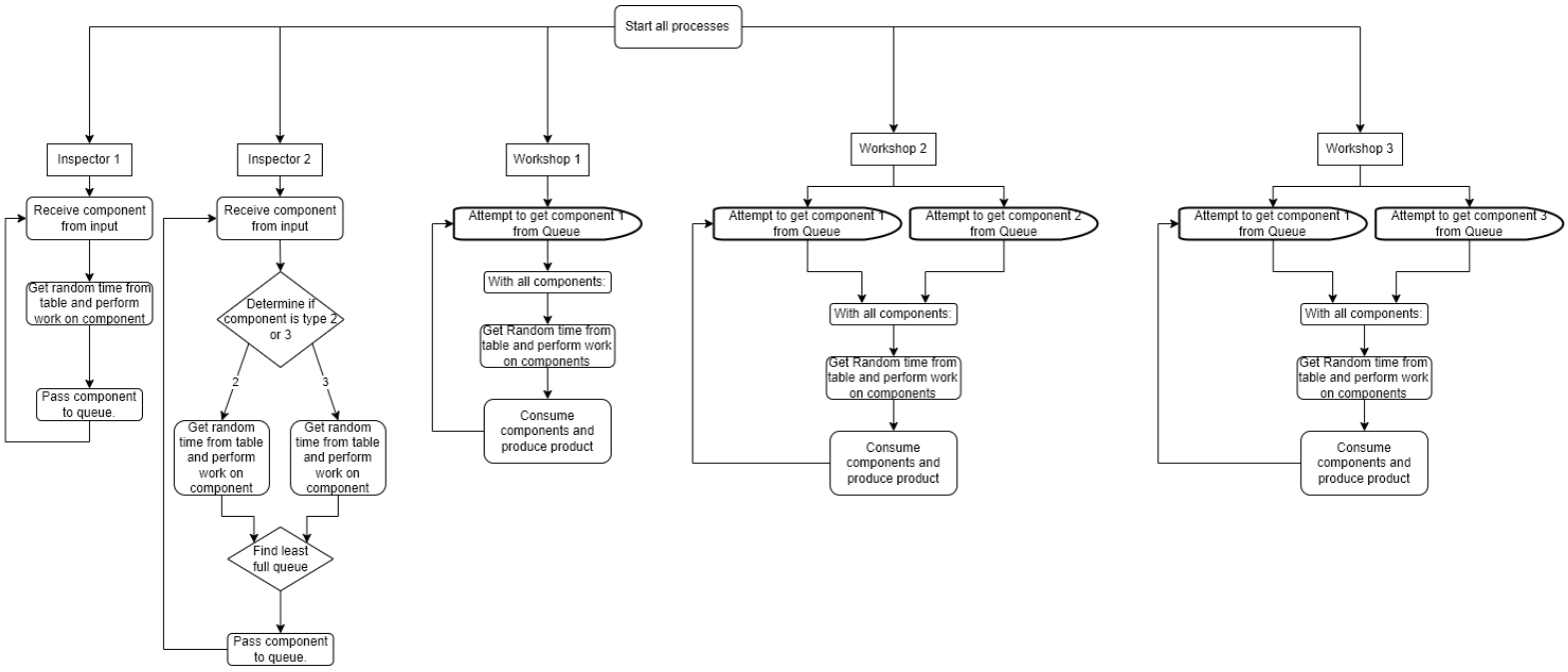


Figure 2. Flow chart.

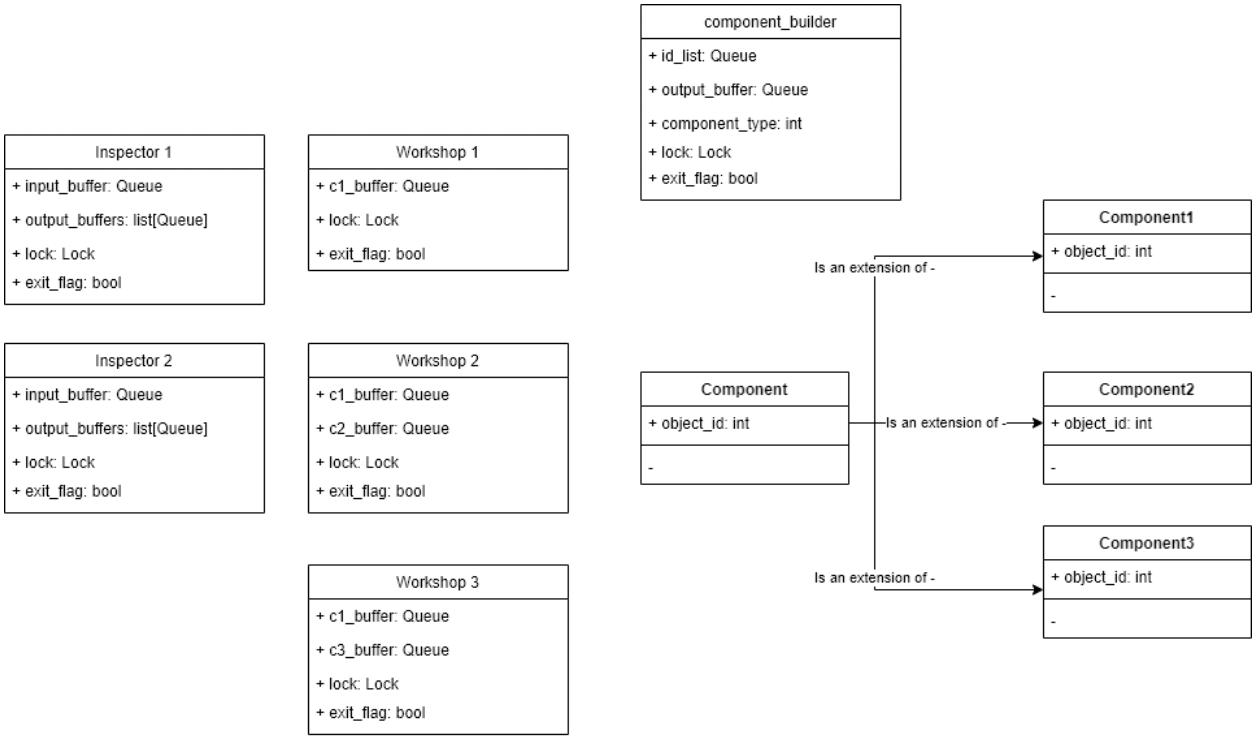


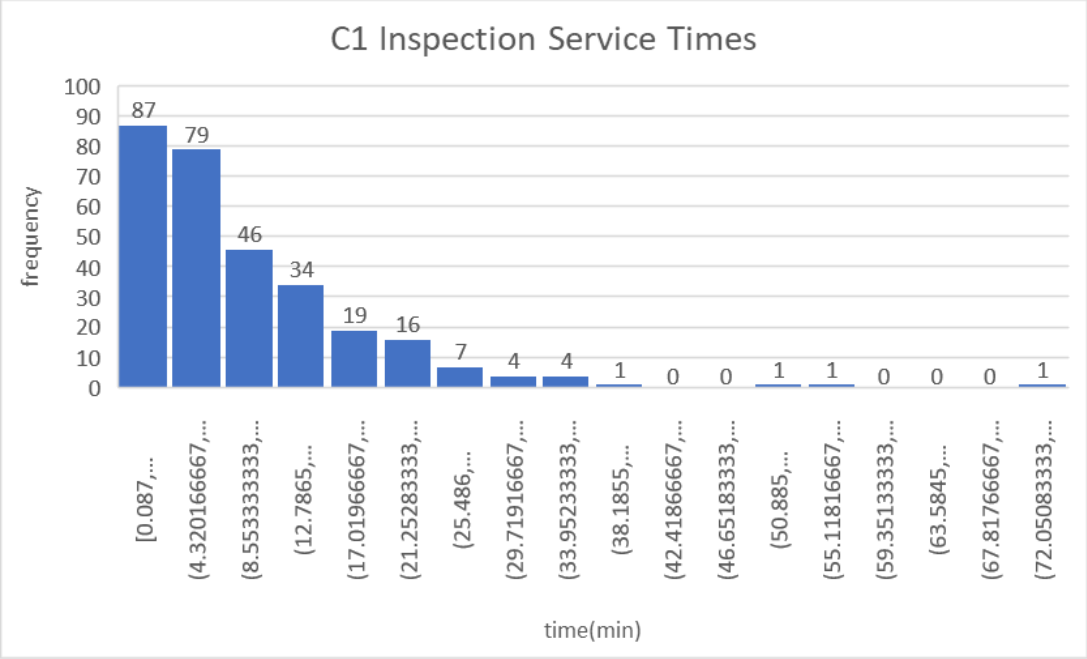
Figure 3. UML class diagram

5.0 Data Collection and Input Modeling

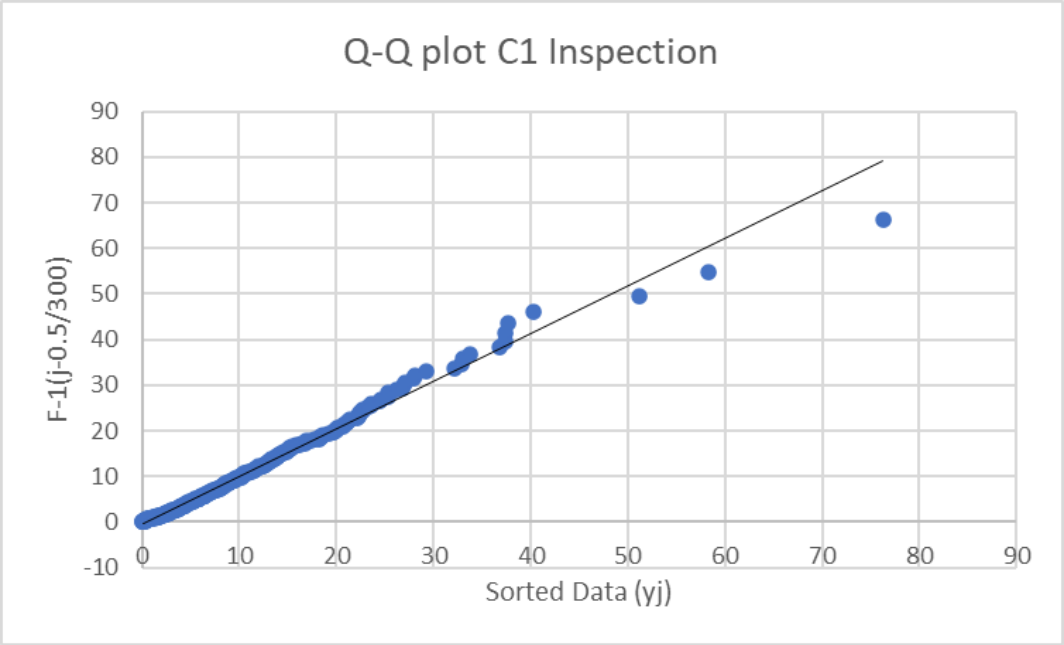
The given dat files contain the service times for the inspection time for each component and workstation are used to plot the histograms to determine the distribution for each sample in order to determine random samples for the simulation.

5.1 Component 1 Inspection Service time

The bin size for the histogram shown below is 18 since the total number of data points is 300 and $\sqrt{300} = 17.33 = 18$ bins.



The histogram shape suggests that the distribution is exponential however it cannot be confirmed until quantile-quantile plot and chi-square test pass for the sample data. The following quantile-quantile plot is produced using the sorted data and the inverse of the exponential distribution at each data point.



For all of the graphs that have been plotted, the trend line is linear and the plotted data points are all aligned which also suggests that our assumption is correct for the sample distribution. The x-axis was the data points sorted and the y-axis is the cdf equation for exponential distribution shown below.

$$F(x) = 1 - e^{-\lambda x}$$

The inverse of the equation above is given by the following expression.

$$F^{-1}(x) = -\frac{ln(1-x)}{\lambda}$$

$$\lambda = 1/\text{mean}$$

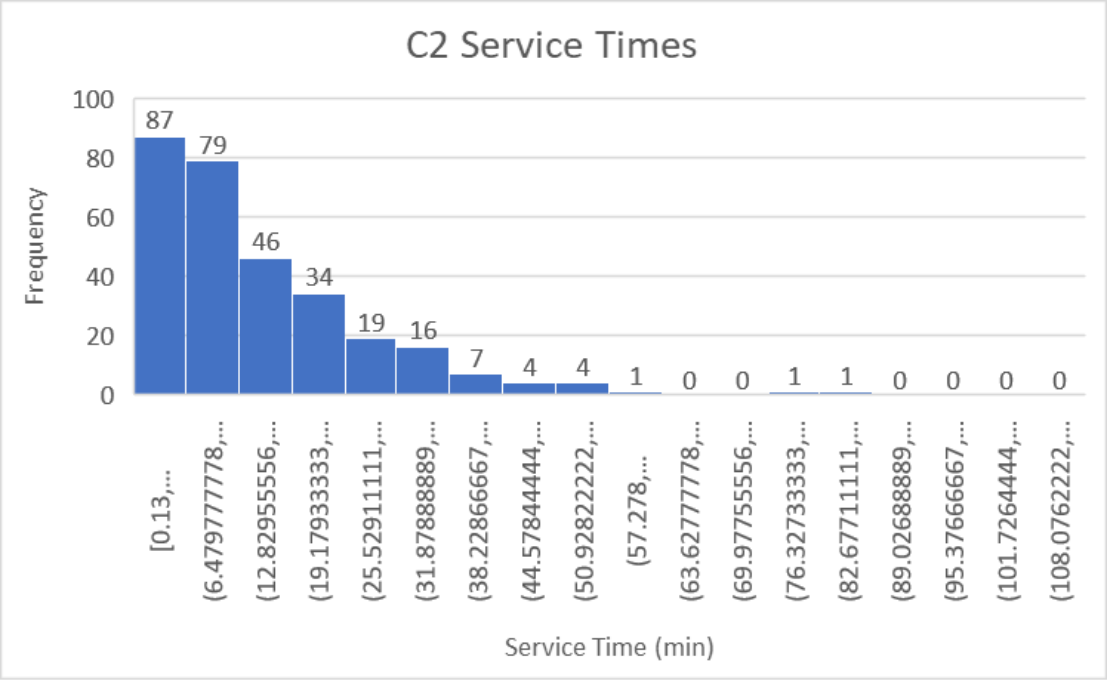
$$x = (\text{j-0.5/300})$$

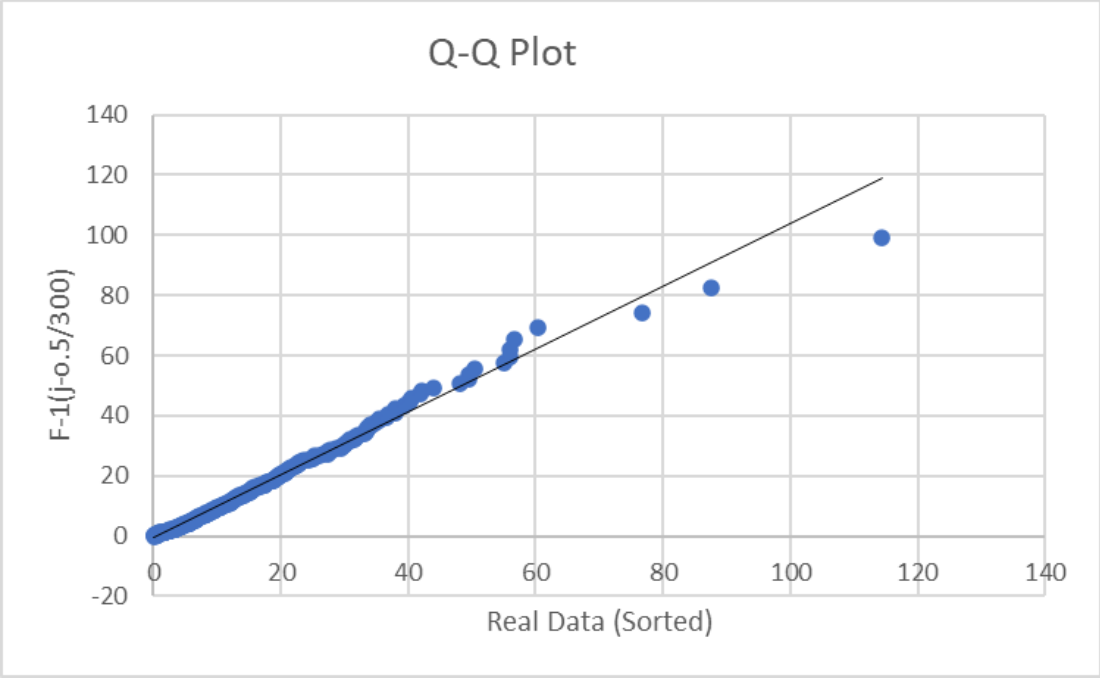
The lambda value in the inverse equation was calculated by finding the mean of all the data points and dividing one by the calculated mean. The x value represents the quantile percentile which is also calculated in the provided file.

The histogram and quantile-quantile plot both suggested that the sample distribution is exponential but to confirm the results the chi square test was also conducted. The chi-square test for exponential distribution is given by the following equation.

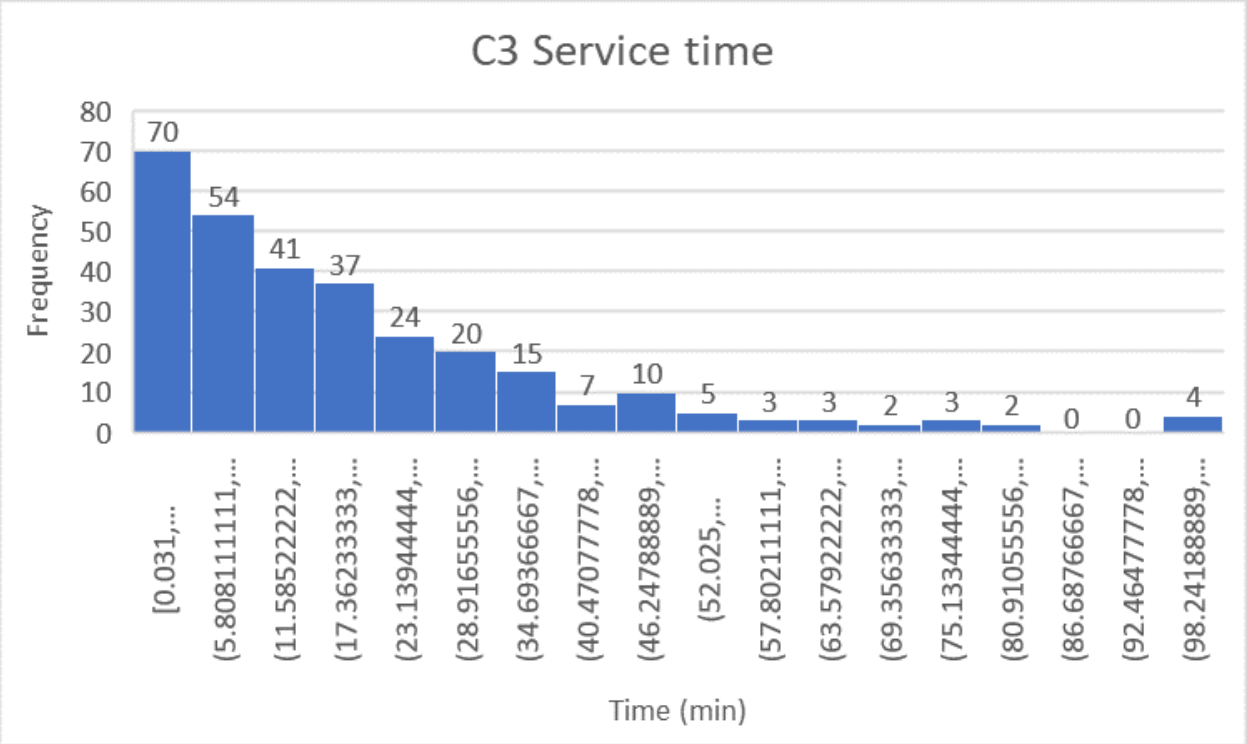
$$X_c^2 = \sum \frac{(O-E)^2}{E}$$

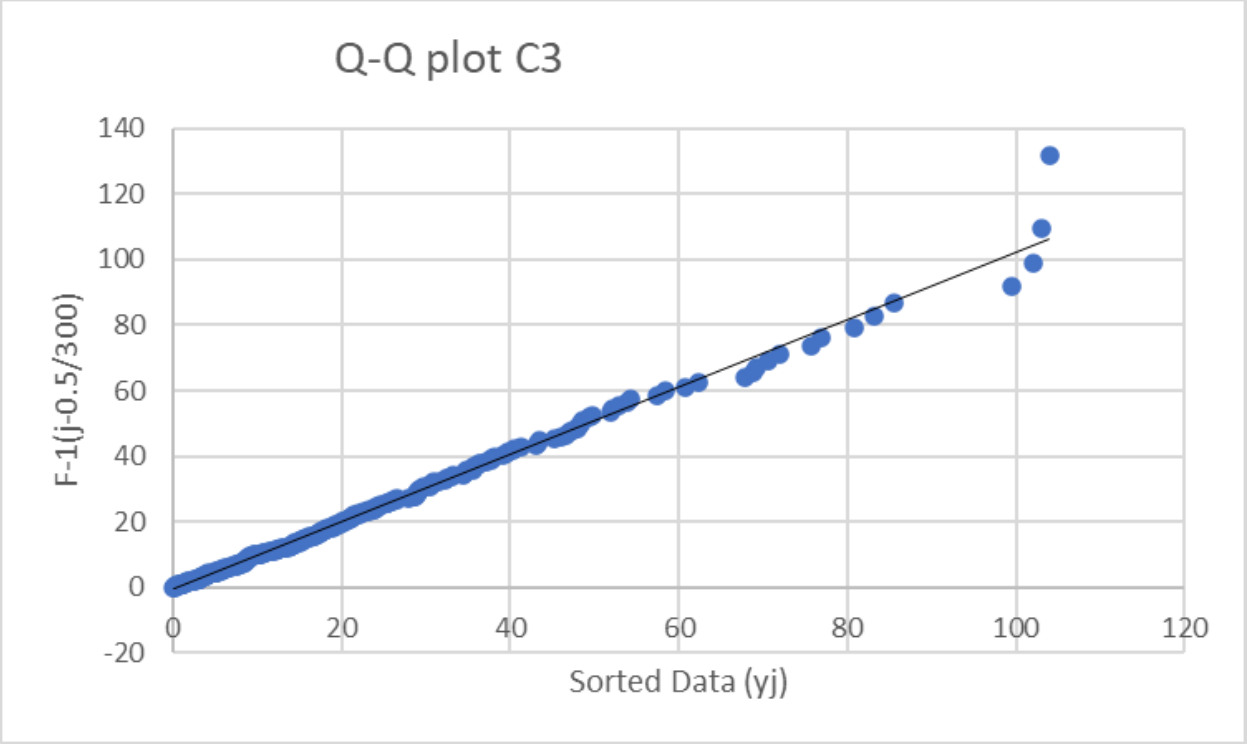
5.2 Component 2 Inspection Service time



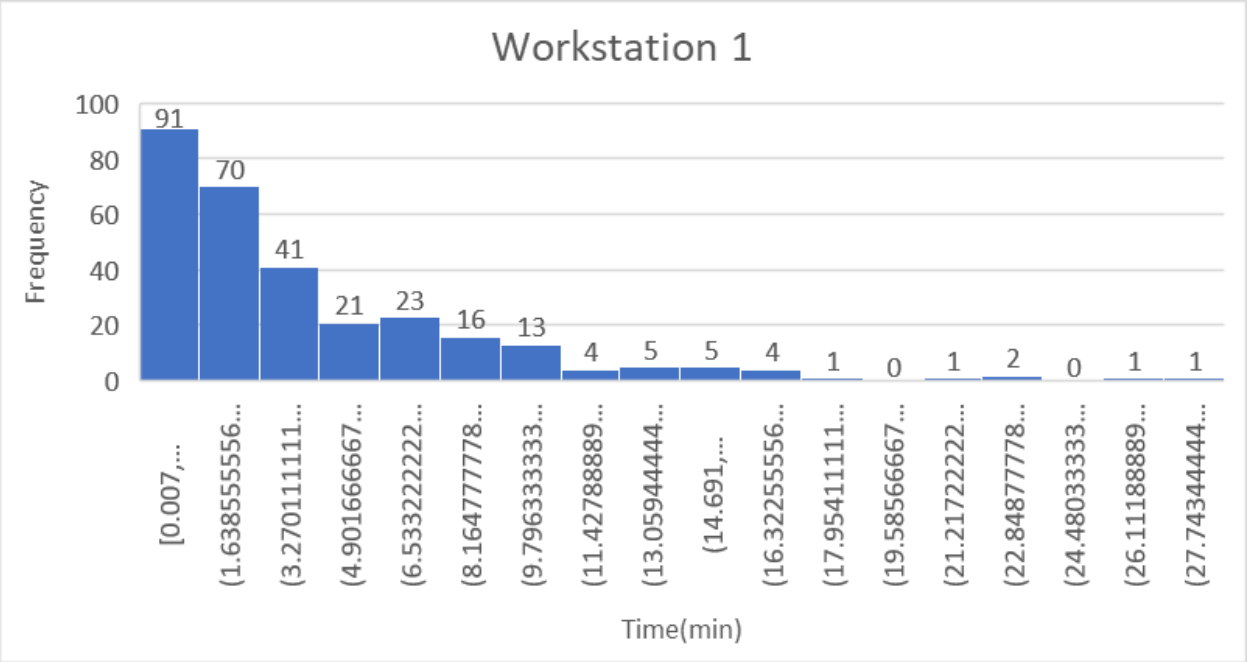


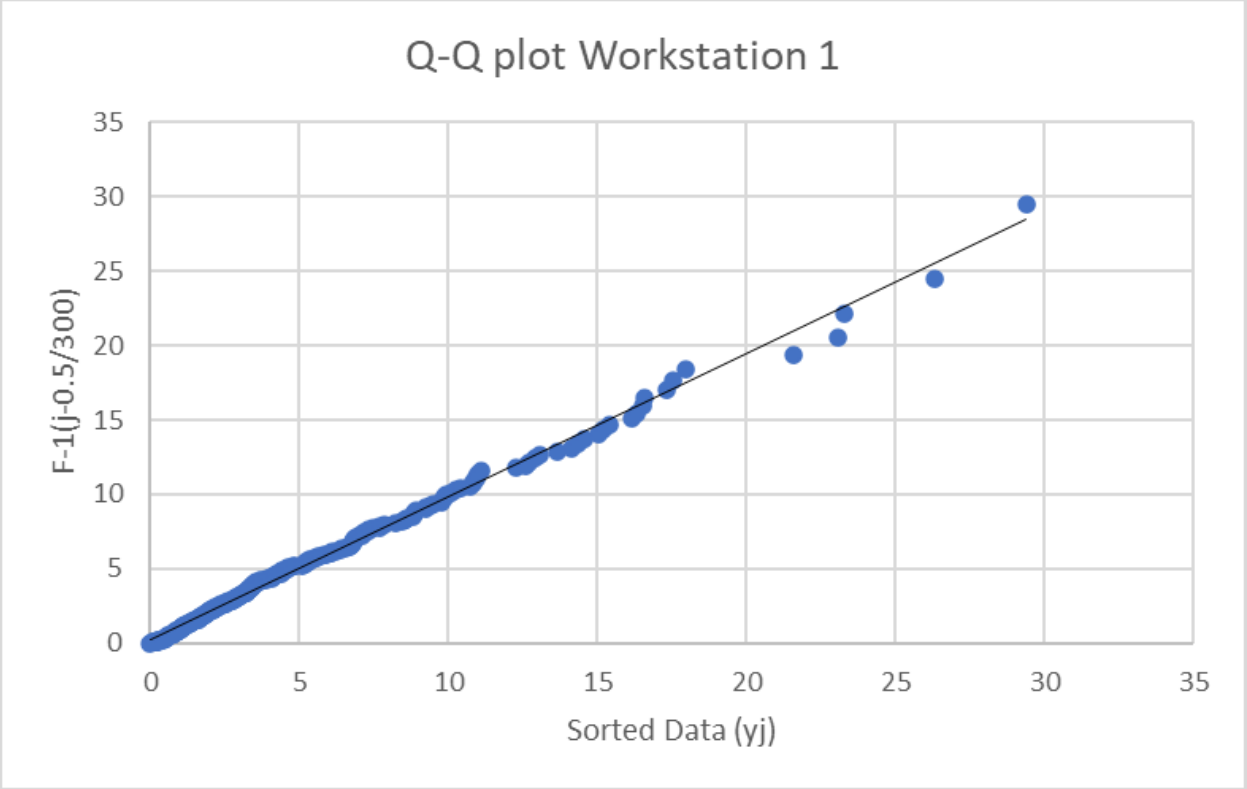
5.3 Component 3 Inspection Service time



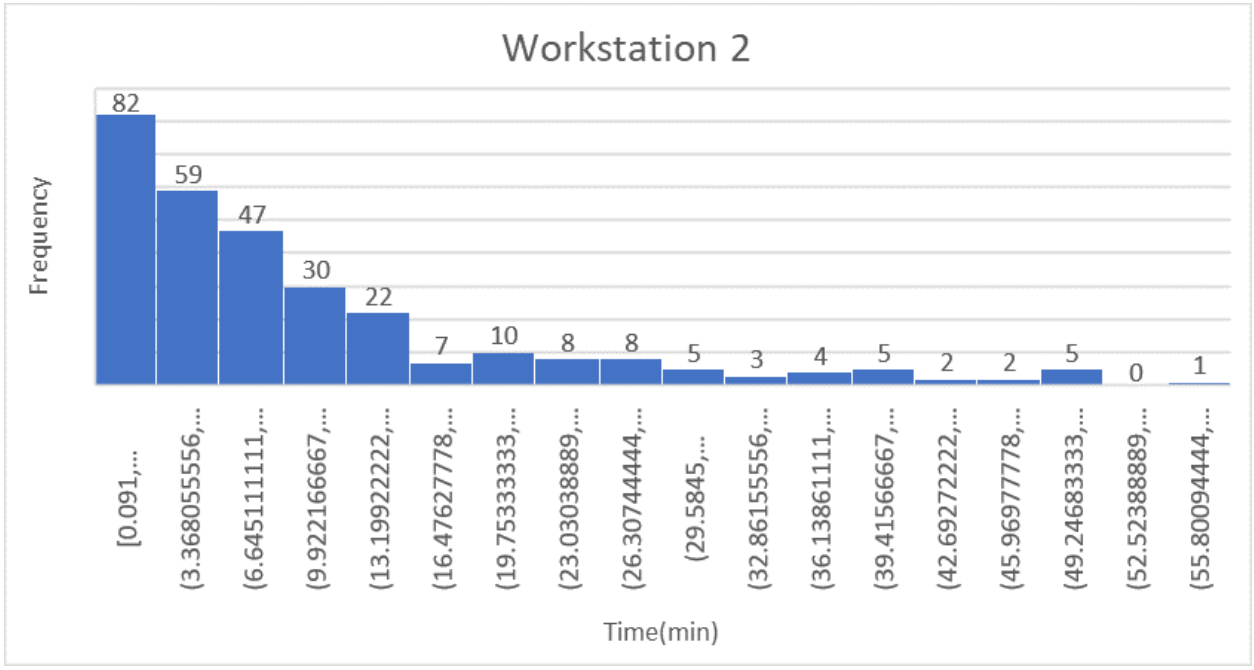


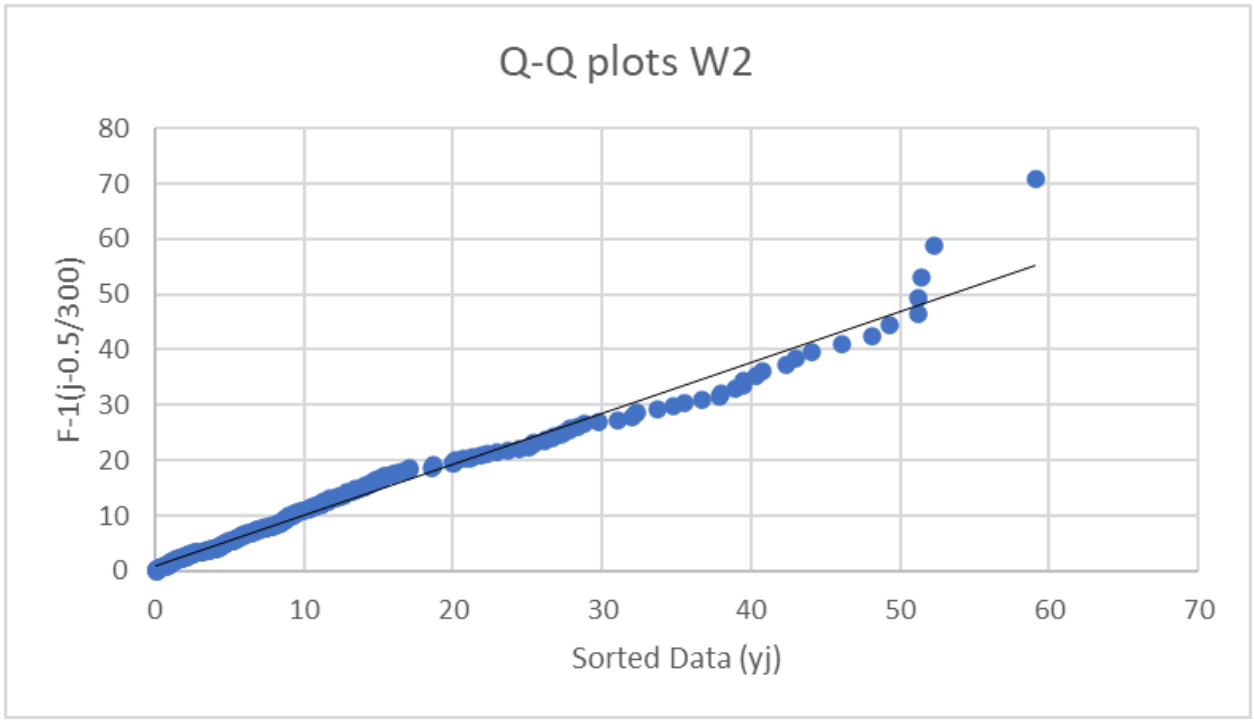
5.4 Workstation 1 Service time



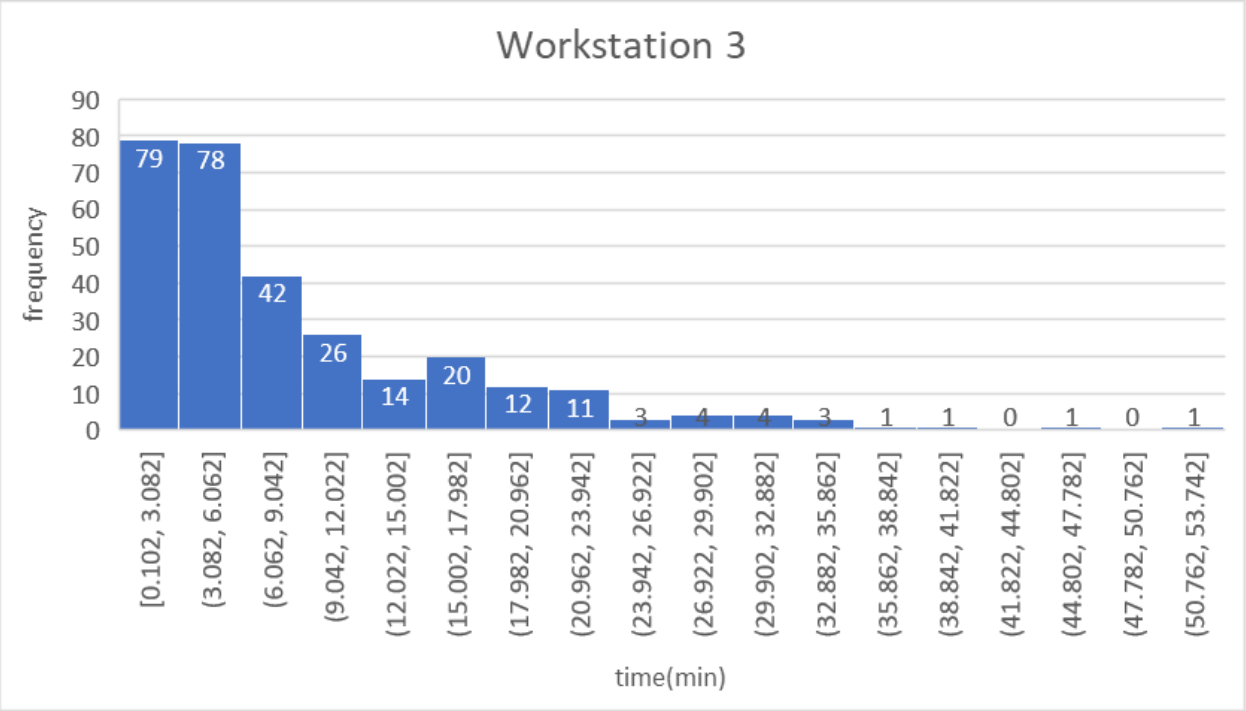


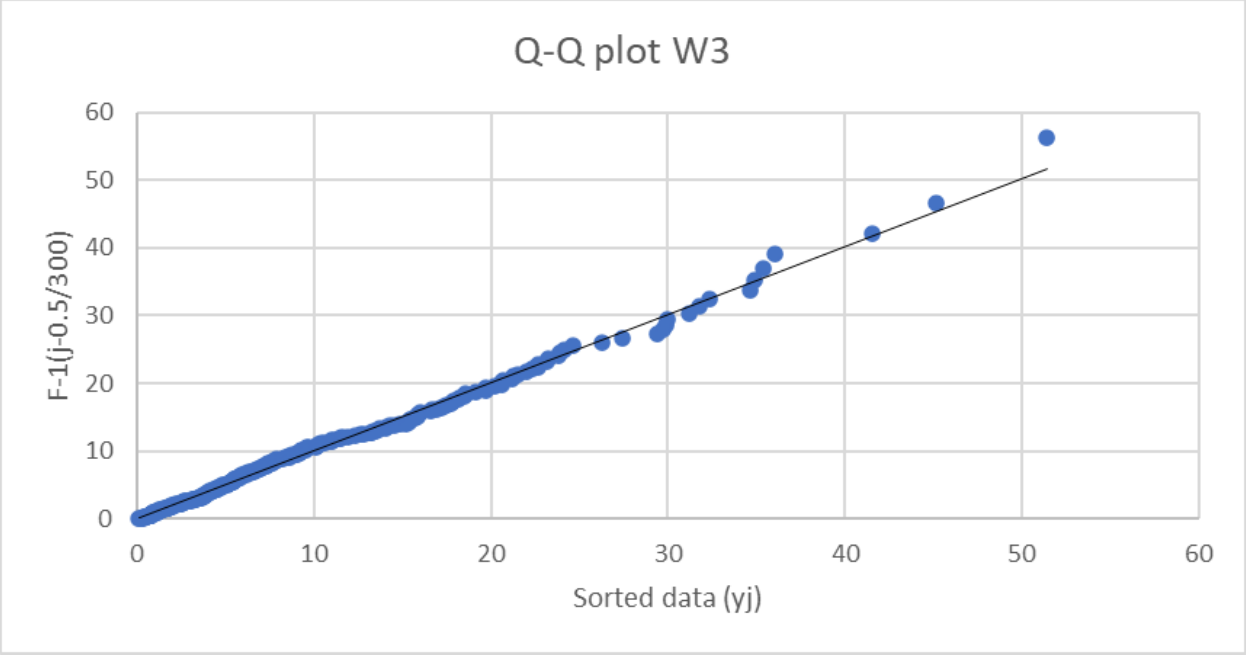
5.5 Workstation 2 Service time


Workstation 2
Frequency
82
59
47
30
22
7
10
8
8
5
3
4
5
2
2
5
0
1
[0.091,...
(3.368055556,...
(6.645111111,...
(9.922166667,...
(13.19922222,...
(16.47627778,...
(19.75333333,...
(23.03038889,...
(26.30744444,...
(29.5845,...
(32.86155556,...
(36.13861111,...
(39.41566667,...
(42.69272222,...
(45.96977778,...
(49.24683333,...
(52.52388889,...
(55.80094444,...
Time(min)


Q-Q plots W2
F-1(j-0.5/300)
0
10
20
30
40
50
60
70
80
Sorted Data (yj)

5.6 Workstation 3 Service time





Histogram is not enough when it comes to plotting the QQ plot or finding the fit for the distribution, it is because the number of data points that is given can be different for every plot that is being made.

6 Chi-Squared Test

Chi Square Test						
	mean = 1/lamda =		10.35791			
	variance =	95.84207				
	lamda = 1/	0.096545				
	start	**end**	**Pi**	**Oi**	**Ei**	**X02**
	0.087	4.32	0.332662693	87	99.79881	1.641397
	4.32	8.553	0.221064798	79	66.31944	2.424577
	8.553	12.786	0.146904496	46	44.07135	0.084402
	12.786	17.019	0.097622647	34	29.28679	0.758509
	17.019	21.2528	0.064883237	19	19.46497	0.011107
	21.2528	25.486	0.043108671	16	12.9326	0.727536
	25.486	29.719	0.028645416	7	8.593625	0.295526
	29.719	33.9523	0.01903687	4	5.711061	0.512642
	33.9523	38.1855	0.012649989	4	3.794997	0.011074
	38.1855	42.418	0.008405027	1	2.521508	0.918096
	42.418	46.6518	0.005587063	0	1.676119	1.676119
	46.6518	50.882	0.003709936	0	1.112981	1.112981
	50.882	55.118	0.002468772	1	0.740632	0.09083
	55.118	59.3513	0.001639255	1	0.491777	0.52522
	59.3513	63.5845	0.001089284	0	0.326785	0.326785
	63.5845	67.8176	0.000723836	0	0.217151	0.217151
	67.8176	72.0508	0.000481016	0	0.144305	0.144305
	72.0508	**77.5698**	0.000393559	1	0.118068	6.587794
				300		

Chi-Squared Test is best used when testing the hypothesis that uses a random sample size and follows the specific distributional form which is the chi-square goodness-of-fit test. It is to compare between the observed data and actual data.

7 Input Generation

In order to have our simulation be as accurate as possible to the measured system, we need our service times to be similar to the measured values. If we were to use randomly generated arbitrary values, even

bounded within the measured bounds, it would not create an accurate representation of the system as service times would not be representative of the real system. Knowing the mean we can use it to bias our pseudorandom number generator to match our observed distribution.

We want a RN that gives maximum density and maximum period. The maximum density is that since each R_i is a discrete value, and by having a large m value, we can have a large set of values to alleviate the problem, as the approximation is not much of an issue.
For the maximum period, we want the longest period possible as it allows us to achieve maximum density and avoid the problem of cycling, where the same value appears over and over.

A single LCM is insufficient for this problem, as it creates an output of even distribution which may have low independence due to its inputs. However, by using a CLCG, we are able to create a far more even distribution by combining the outputs of multiple LCM generators. With this far more even distribution, by running the output through an inverse CDF function we can transform our random numbers from a uniform distribution to an exponential distribution. This distribution can then be biased to match the various exponential distributions of service times of our six different service time tables.

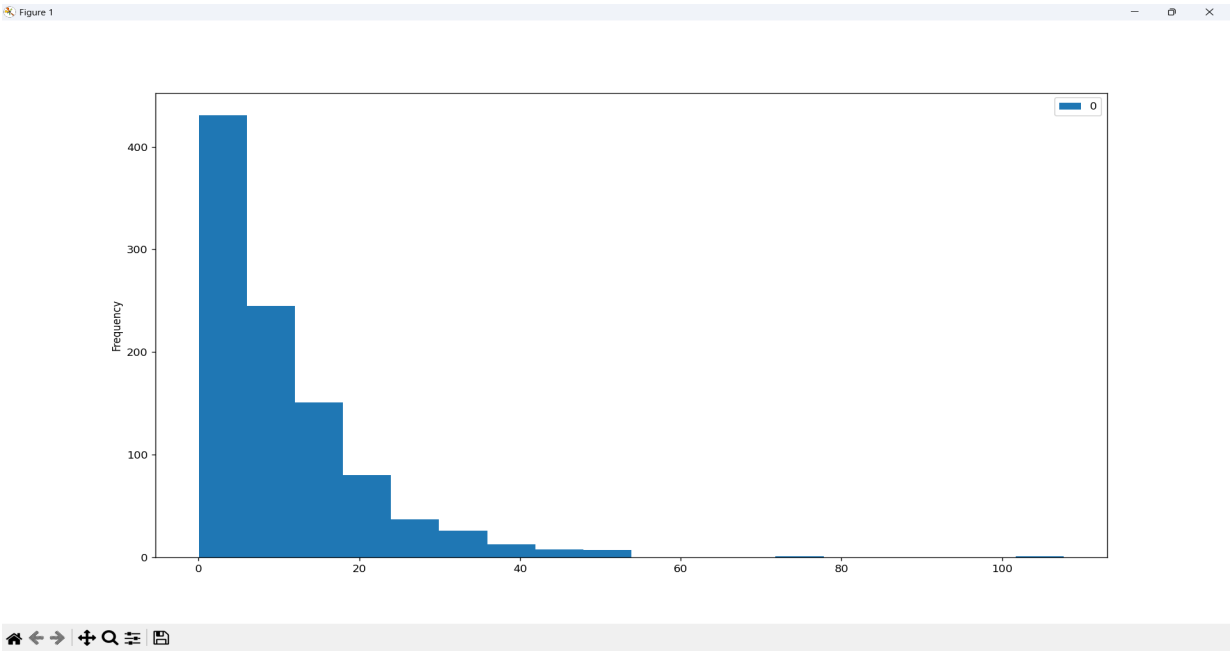


Figure: Unbiased Exponential Random Number output as histogram.

These Biased systems were tested with the chi-squared test for independence and the Kolmogorov-Smirnov test for uniformity. All the distributions passed these tests, showing that the generations were high quality.

Below is the Code used to generate the three LCM CLCG and the function used to transform it into an Exponential distribution.

```
"""
This file is responsible for the generation of random values to be used in the simulation.
```

```
"""
import math
import matplotlib.pyplot as plt
import pandas

class LCMclass():
    """
    Defines a LCM for use in the CLCG.
    """
    def __init__(self, Xo, m, a, c, noOfRandomNums=500):
        self.Xo = Xo
        self.m = m
        self.a = a
        self.c = c
        self.noOfRandomNums = noOfRandomNums

    def LCM(self):
        """
        Function to generate random numbers
        """
        # Initialize the seed state
        randomInts = [0] * (self.noOfRandomNums)
        randomInts[0] = self.Xo
        # Traverse to generate required
        # numbers of random numbers
        for i in range(1, self.noOfRandomNums):
            # Follow the linear congruential method
            randomInts[i] = ((randomInts[i - 1] * self.a) + self.c) % self.m
        return randomInts

def CLCG(LCM_list):
    """
    This simulates a CLCG.
    """
    LCM_output1 = LCM_list[0].LCM()
    LCM_output2 = LCM_list[0].LCM()
    LCM_output3 = LCM_list[0].LCM()
    output_list = [0] * 1000
    for i in range(len(LCM_output1)):
        output_list[i] = (LCM_output1[i] - LCM_output2[i] + LCM_output3[i]) % LCM_list[0].m - 1
    return output_list

def rn_generator_insp1(input_list, m1):
    """
    This generates a RN field for inspector 1.
    """
    cdf_conversion = [0] * 1000
    randomNums = input_list
```

```
    for i in range(len(randomNums)):
        randomNums[i] = randomNums[i]/m1
    for i in range(len(randomNums)):
        cdf_conversion[i] = -((math.log(1 - randomNums[i])/(1/10.35791)))
    return cdf_conversion

# Driver Code
if __name__ == '__main__':
    # Seed value
    Xo = 1234
    # Modulus parameter
    m1 = 65536
    m2 = 32768
    m3 = 16384
    # Multiplier term
    a = 101427
    # Increment term
    c = 321
    # Number of Random numbers
    # to be generated
    noOfRandomNums = 1000
    # Function Call
    LCM1 = LCMclass(Xo,m1,a,c,noOfRandomNums)
    LCM2 = LCMclass(Xo,m2,a,c,noOfRandomNums)
    LCM3 = LCMclass(Xo,m3,a,c,noOfRandomNums)
    LCM_list = [LCM1, LCM2, LCM3]
    CLCG_list = CLCG(LCM_list)
    output = rn_generator_insp1(CLCG_list, m1)
    # Print the generated random numbers
    for i in CLCG_list:
        x = i/m1
        print(x, end = " \n")
    df = pandas.DataFrame(output)
    ax = df.plot.hist(bins=18)
    plt.show()
```

This is a Python script that generates random numbers using a linear congruential generator (LCG) and then applies a transformation to generate another set of random numbers. The generated random numbers are then plotted in a histogram. The LCG used in this script is defined by the LCMclass() function. It takes four parameters - Xo (the seed value), m (the modulus), a (the multiplier term), and c (the increment term). The LCM() method generates a list of random numbers using the LCG method. The CLCG() function simulates a Combined Linear Congruential Generator (CLCG) by generating three sets of random numbers using different LCGs and then applying a formula to generate a new set of random numbers. The rn_generator_insp1() function takes the output of the CLCG() function and applies a

transformation to generate a new set of random numbers. In the main block of code, three instances of LCMclass() are created with different values of the modulus parameter. The CLCG() function is then called with these instances and the number of random numbers to be generated. The output of CLCG() is then passed to rn_generator_insp1() to generate a new set of random numbers.

8.0 Model Verification and Validation

Verification is a step we must take to ensure that the conceptual model and the operational model are producing almost the same results. This step can be done by setting a different set of inputs and observing the output values under a variety of input parameters. On the other hand a model needs to be validated using an iterative process. The accuracy of the output values are compared to the previous iterations and this process continues until the differences between the actual and expected values is reasonable enough for the model to be judged. We can utilize a variety of validation techniques, including face validation, code validation, input validation, output validation, and cross-validation, to confirm the model. In cross-validation, the accuracy and dependability of the model are verified using several datasets. Face validation is the process of asking experts to evaluate a model to see if it makes sense and is realistic. Verifying the completeness and correctness of the inputs is known as input validation. To verify the model's correctness and dependability, output validation entails comparing the model's results to actual observations.

In order for us to verify the conceptualized model we needed to get output values because only the input values are provided. To accomplish this the python script is written to implement a random number generator for inspectors and workshops. It defines six functions that generate random numbers for different scenarios. Each function takes a list of random numbers, a modulus parameter, and the number of random numbers to generate as input, and returns a list of RN values. Now that random numbers are generated we can verify our model in ‘input_validator.py’ file. This code is used to validate that the random number generators in the model module produce input values that are equivalent in range and mean to the real data. There are six functions defined, one for each component of the manufacturing process. Each function calculates the mean of the real data and the mean of the random number field generated by the corresponding function in the model module for a given n value (number of iterations). Then it prints the actual mean, random mean, and the difference between them as a percentage. The print_mean function is a helper function to print the means and the difference between them in a formatted way. Finally, in the main block, each function is called with n=2000. This means that the random number generators will be run 2000 times to generate input values, and the means of the real data and the random number field will be calculated and compared for each function.

Implementing the Little's law to verify the model proved to be difficult however the way this law would work is by following the steps listed below: Collect data on the system's average arrival rate and the average service time. Use Little's law to calculate the expected number of items in the system and the expected time spent in the system. Run the simulation model and collect data on the actual number of items in the system and the actual time spent in the system. Compare the expected values calculated using Little's law to the actual values obtained from the simulation model. If the values are close, the model is validated. Currently, the 'model_validator.py' contains a function ***littles_law_validation(data)*** which prints the data. This data is then calculated for the littles law validation manually. In the calculations we find that the model passes the little law test, showing that the simulation has all items which enter the system will exit as products.

9.0 Production Runs and Analysis

We have created a python file called replication_variable.py. This code allows to store and manage data for each replication of a simulation. It has two classes which are the ReplicationVariables and LittleLawVariables. The ReplicationVariables allows updating these attributes such as storing service times, idle times, block times, the number of products created, and queue occupancy with the new data as the simulation progresses. Each time a replication is completed, the replication's data is outputted to a list containing each replication's data. The default number of replications is 1000 in order to generate a large set of simulations to help reduce the effects of any bias introduced by our random number generators. Each replication also runs for a default time of 30000 seconds, or 8.3 hours. This generates a large collection of data for each replication while also creating a very large collection of data for the entire simulation, more than enough to generate effective averages for our measured quantities.

In order to reduce the effects of initialization bias (bias introduced by the initial state of the random number generators) it is best to drop a section from the beginning of each replication in order to avoid making assumptions on data based on initialization bias. In our program, based on multiple test runs, the initialization phase lasted between 500 to 1000 seconds, and showed a downward bias. To alleviate this, we treated the initialization phase as 10% of total replication time, and thus culled the first 10% of each replication from our recorded data.

(Note: There is a recurring bug in the code that causes portions of the output to not print to console, I have done everything I could think of to fix this but it just won't be fixed. If the code is run and the output is missing the required quantities for analysis, it must be rerun.)

In the Model_Validator.py, we used the scipy and numpy libraries to calculate the confidence interval. The function claculate_confidence_interval calculates the interval based on a given dataset using the t-distribution. The CI of the buffer occupancy is shown below.

```
The buffer occupancy for workshop 1 component 1 is 0.33379955138591805 ±9.795639441193049e-05, the real value is 0.28
The buffer occupancy for workshop 2 component 1 is 0.47646232017295453 ±0.00049059768824329, the real value is 0.41
The buffer occupancy for workshop 3 component 1 is 0.23650353268675892 ±0.00044541157588627144, the real value is 0.60
The buffer occupancy for workshop 2 component 2 is 0.36290989704219484 ±0.0006605216525759945, the real value is 0.32
The buffer occupancy for workshop 3 component 3 is 1.3356489239971163 ±0.0007776461918213853, the real value is 1.75
```

The CI was calculated at 95% confidence, and as can be seen, none of our real values fell within the confidence intervals of our simulated values. The values of workshop 1 component 1, workshop 2 component 1, and workshop 2 component 2 all regularly fall within 10-15% of the real values, while workshop 3 components 1 and 3 regularly vary at the shown values, approximately 50-60% off. This shows that the issue is likely being caused by how the simulation is done for workshop 3, and that output validation would likely pass if this issue could be resolved. I believe the issue to be with the random number generator for the workshop 3 service times, however varying the logarithmic curve did not substantially change these values.

Because of this it implies that the issue is instead with how the simulation is performed, or with how the occupancy data is collected. Currently occupancy data is collected by polling for the occupancy level every time it changes during the simulation. It is possible that this method is introducing an unknown bias into the output data which is affecting the results, this would also explain the offsets in the other occupancy data. Another sign this may be the issue is that the error margin is extremely small on the confidence interval, showing that the vast majority of values are the same, this may be due to the scale of the output (0,1,2). Unfortunately in the time allowed we were not able to resolve this error.