SYSC 4005 A - Discrete Simulation / Modeling

Winter 2023

Simulation Project Final Deliverable

Completed By:

Favour Olotu	101130753
Ray Agha	101108060
Joseph Anyia	101117261

Due: April 12, 2023

1. Problem Formulation

A simulation study will be conducted to assess the performance of this manufacturing facility, partly based on observed historical data of the inspectors' and workstations' service times given in units of minutes. An additional objective is to possibly improve the policy that Inspector 1 follows when delivering C1 components to the different workstations to increase throughput and/or decrease the inspector's “blocked" time.

The manufacturing process creates 3 products, referred to as P1, P2, and P3. The products are created on corresponding workstations called W1, W2, and W3, with components referred to as C1, C2, and C3. The product composition is as follows: · P1 is composed of one C1 · P2 is composed of one C1, and one C2 · P3 is composed of one C1, and one C3 Creating products involves two inspectors, referred to as I1 and I2, inspect, clean, and repair components before they are sent to workstation buffers. The supply of components is infinite. Obtaining a component to inspect happens instantaneously. However, it takes a period of time to complete an inspection, and this time period will follow a specified distribution.

Inspector I1 is responsible for inspecting component C1, and inspector I2 is responsible for inspecting components C2 and C3. For now, inspector I2 will randomly select either C2 or C3 for inspection without considering other production conditions. Once an inspection is complete, the components are sent to one of the three workstations, W1, W2, and W3, which each assemble products P1, P2, and P3, respectively.

Each workstation has a buffer with a capacity of two components for each of the components it requires to assemble its respective product and removes its needed components from the buffers to begin assembly. A workstation can only assemble one product at a time, and assembly may begin as soon as all of its component buffers have a component to take. Each workstation will have assembly times that will follow a specified workstation-specific distribution. When an inspector completes an inspection, the inspector will try to place the component on the workstation buffer if it has an opening for that component type. If all buffers have an equal amount of free space, the buffer corresponding to the lowest workstation number will be chosen.

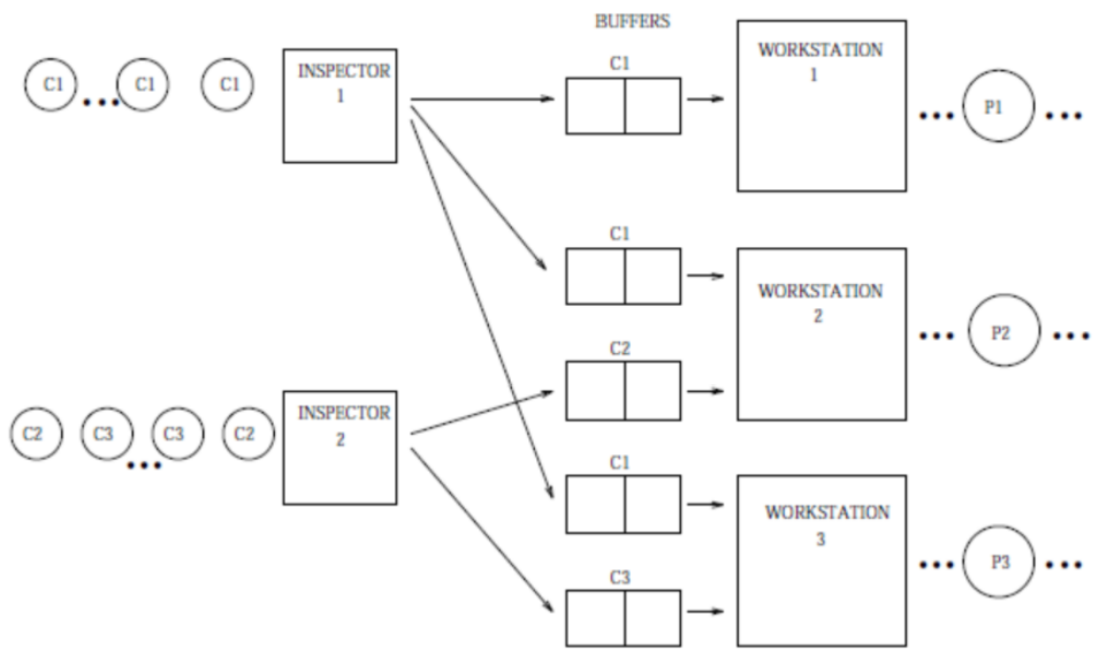


Figure 1: Schematic illustration of the manufacturing facility.

In the present mode of operation, Inspector 1 routes components C1 to the buffer with the smallest number of components in waiting (i.e., a routing policy according to the shortest queue). In case of a tie, W1 has the highest and W3 has the lowest priority.

2. Setting of objectives

The following are the objectives of this project:

1. Study the performance of the manufacturing factory
2. Increase the production protocol for inspector 1

The idea behind the simulation is to use inputs to generate probable output. In this project, the inputs used are the service times of both the inspectors and the workstation. The input data are units of time. (seconds or minutes converted to seconds). The outputs for the simulation are as follows;

I. Product per unit time: This is the number of products produced over a specified period of time. The accounts for all the products from all three workstations.
II. Probability a workstation is busy:
III. Average buffer occupancy of each buffer: The output value represents the average occupancy for each buffer used in the system.
IV. The probability that an inspector is blocked or idle: This would be a value expressed in percentage.

3. Model Conceptualization

This problem will be modelled using simulation software. The Simulation will use the object-oriented programming paradigm to create a clear separation of concern between distinct entities of the system. Random variable generators are used for simulating the random inputs of components for each inspector. The first Input will be all components 1, which will be connected to inspector 1. The second Input will be a random assortment of component 2 and component 3 connecting to inspector 2.

Each inspector will have additional inputs from data files with inspection times for each component. Inspector 1 will input times for component 1 and inspector two for components 2 and 3. The first inspector will output component 1 to three different buffers of component 1 that feed into each workstation. The inspector will prioritize the most open buffer, followed by the workstations in order from 1 to 3. The second inspector will give component 2 to a buffer on workstation 2 and component 3 a buffer on workstation 3.

Each workstation will have additional Input from a specific data file for time to assemble the final part. Once the workstations have taken the inputted time, they can export the final product and add more components. Within the functions for the inspectors, there will be additional coding to record the time it takes them to complete their processing time to determine if they become blocked. The primary function will also require code to record the system's throughput.

These processes will be initiated in a main Simulation class that will track the timings of events, translate these timings into metrics of interest, and then process statistics from these metrics accordingly.

4. Model Translation

The choice of simulation language for this project would be the python programming language. We used python because it is easy to code in. It provides libraries such as “numPy” to help with complex number operations, “random” for generating random numbers and data structures compatible with expressing the problem as a model.

Inspector Module:

The Inspector module contains two class implementations, one for each inspector. Each inspector class will calculate the delay time based on the data file corresponding to each component. Inspector1 has an “__init__” method to initialize the class with the mean of component 1. After this is the “generate_mean_from_data” method that calculates the mean from the component's data set. The last method of Inspector1 is “generate_inspect_time” and this uses the mean to generate a random delay within a distribution of the mean. This uses numpy.random.exponential with the mean as an input. Inspector2 has an identical structure and functionality as Inspector1. The only difference is that since Inspector2 works with two components, there is an added functionality for randomly selecting the component to generate a random delay.

Workstation Module:

The Workstation module encapsulates the logic for the Workstation entity represented as a class. The Workstation class receives components and calculates a delay based on the imputed distribution of data. The Class is set up with the __init__ file to initialize the mean of the production times and the product. The Next method is “generate_mean_from_data,” which will see what product is being worked on, select the corresponding data file, and calculate the mean for random number generation. Finally, the “get_delay_time” method generates a product assembly delay time from the mean.

Buffer Module:

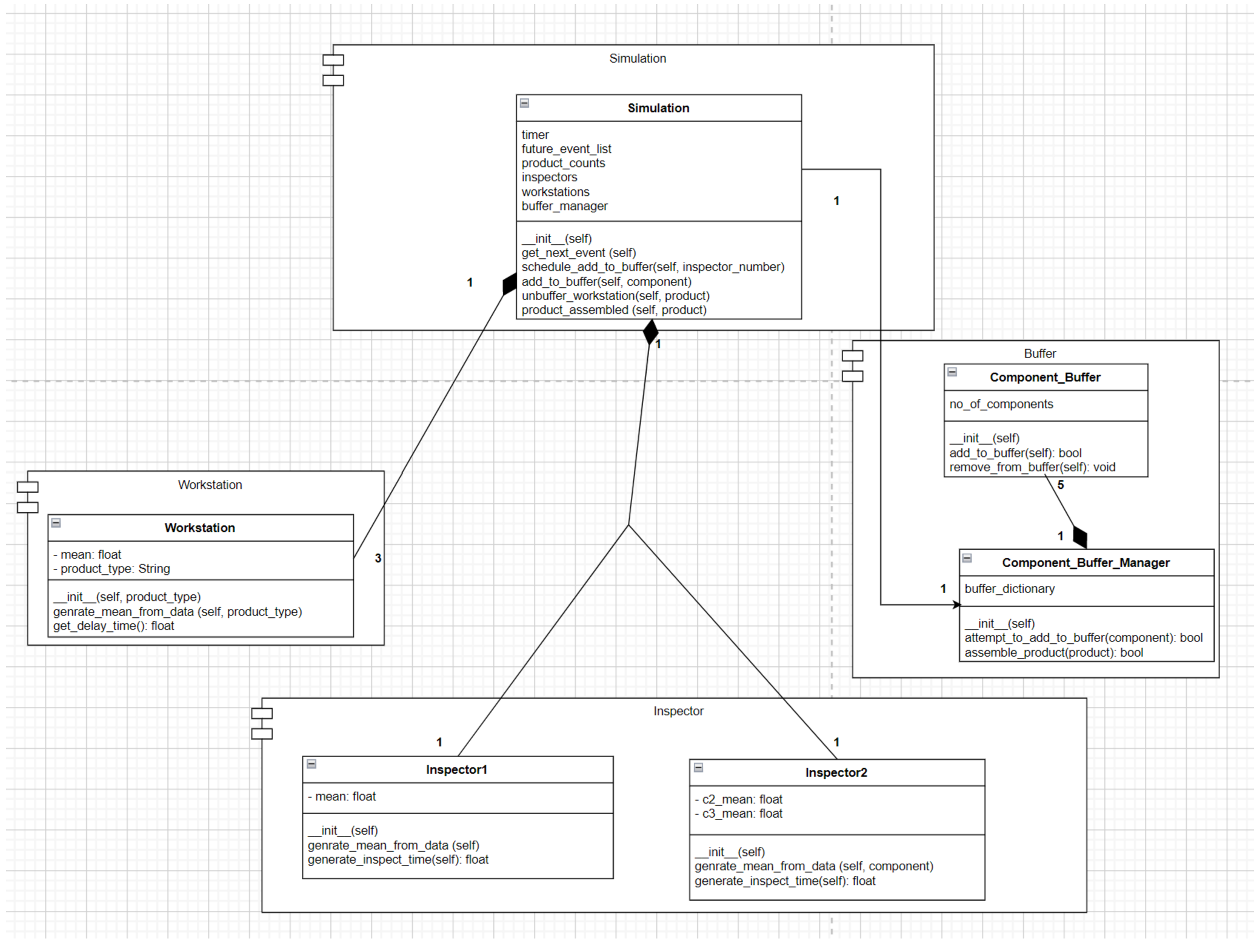
The Buffer module contains two class implementations a “Buffer” class and the “Buffer_Manager” class. The Buffer implements the buffer object
where components are stored and waiting to be moved into the workstation. The
Buffer has an “add_to_buffer” function which attempts to add a
component to the buffer object and returns true if successful and false otherwise. It also has a “remove_from_buffer” to remove components from the Buffer.
The Buffer_Manager class is a controller for all possible buffers, as described in the problem statement. It routes components to the appropriate Buffer and pushes components out of the Buffer to the workstations. The class consists of a constructor that initializes all the Component_Buffer objects needed with their associated component product pairs represented as a tuple. The “attempt_to_add_to_buffer” function takes in a Component as a parameter and attempts to find a buffer to send the component to. It returns True and the product if successful, and False and None otherwise. The “assemble_product” method takes a product as a parameter. It takes the needed components off the workstation buffer to assemble the product. It returns true if the needed components were provided from the buffers and false otherwise.

Simulation Module:

The Simulation module serves as the main file. It contains the Simulation class and the main script that runs the simulation. The simulation consists of a single event list to manage events until a maximum production number has been reached. The constructor method “__init__” is for setting up the necessary components for the simulation, i.e. inspectors, workstations and buffer manager. The “get_next_event” method identifies the closest event in the future event list based on the time it was added. It then returns and removes the event from the list to advance the simulation. The “schedule_add_to_buffer” method initiates an inspection process for a given inspector. It adds the estimated inspection completion time to the future event list. The inspection time is estimated using the inspection time generation mechanism from the inspector module.

The “add_to_buffer” method processes an attempt to add an inspected component to a buffer; if a buffer can accept the component, then the construction of a product will be immediately scheduled by adding a corresponding event to the future event list; else, nothing happens. The “unbuffer_workstation” method deals with product assembly in the corresponding workstation. It will try to retrieve the needed components from the buffer. If successful, a new event will be added to the future event list for the completion of the product assembly. The “product_assembled” method deals with keeping track of the products assembled.

UML Class Diagram:



Simulation
Simulation
timer
future_event_list
product_counts
inspectors
workstations
buffer_manager
__init__(self)
get_next_event (self)
schedule_add_to_buffer(self, inspector_number)
add_to_buffer(self, component)
unbuffer_workstation(self, product)
product_assembled (self, product)
1
1
1
Buffer
Component_Buffer
no_of_components
__init__(self)
add_to_buffer(self): bool
remove_from_buffer(self): void
5
1
Component_Buffer_Manager
buffer_dictionary
__init__(self)
attempt_to_add_to_buffer(component): bool
assemble_product(product): bool
1
Workstation
Workstation
- mean: float
- product_type: String
__init__(self, product_type)
genrate_mean_from_data (self, product_type)
get_delay_time(): float
3
Inspector
1
Inspector1
- mean: float
__init__(self)
genrate_mean_from_data (self)
generate_inspect_time(self): float
1
Inspector2
- c2_mean: float
- c3_mean: float
__init__(self)
genrate_mean_from_data (self, component)
generate_inspect_time(self): float

5. Data Collection and Input Modeling

The distribution of each of the data sets for the simulation entities, based on the shape of their histograms, is an exponential distribution as it starts at the peak and observes a continuous downward trend. Refer to Appendix A for the histograms. The histograms were plotted with a variable bin size. Although it is recommended to use the square root of the sample size for the bin width (from the textbook), it was difficult to get reasonable information with that bin width.

QQ Testing:

The shape of a histogram is necessary for selecting a distribution but not enough to justify the fit of a particular distribution. Knowing that information, the QQ plots are used to evaluate the distribution fit, and the Chi-squared test is used to check the goodness of the fit.

QQ and Chi-squared testing will be conducted using an exponential distribution to test the fit. If the exponential distribution is unsuccessful, a different distribution will be attempted. Then the inverse transform method of random variate generation will be conducted on the distribution.

A QQ or quantile-quantile plot is a graphical method of comparing two probability distributions by plotting their quantiles against each other. It provides a method to evaluate how well a distribution represents a data set. In the "Statistical_Analysis" excel workbook, we currently have implemented QQ testing for Exponential distributions because of the initial observations of the histograms. For the exponential distribution, the following equation is used as the inverse function:

$$F^{-1}(p;\lambda) = \frac{-\ln(1-p)}{\lambda}, \qquad 0 \leq p < 1$$

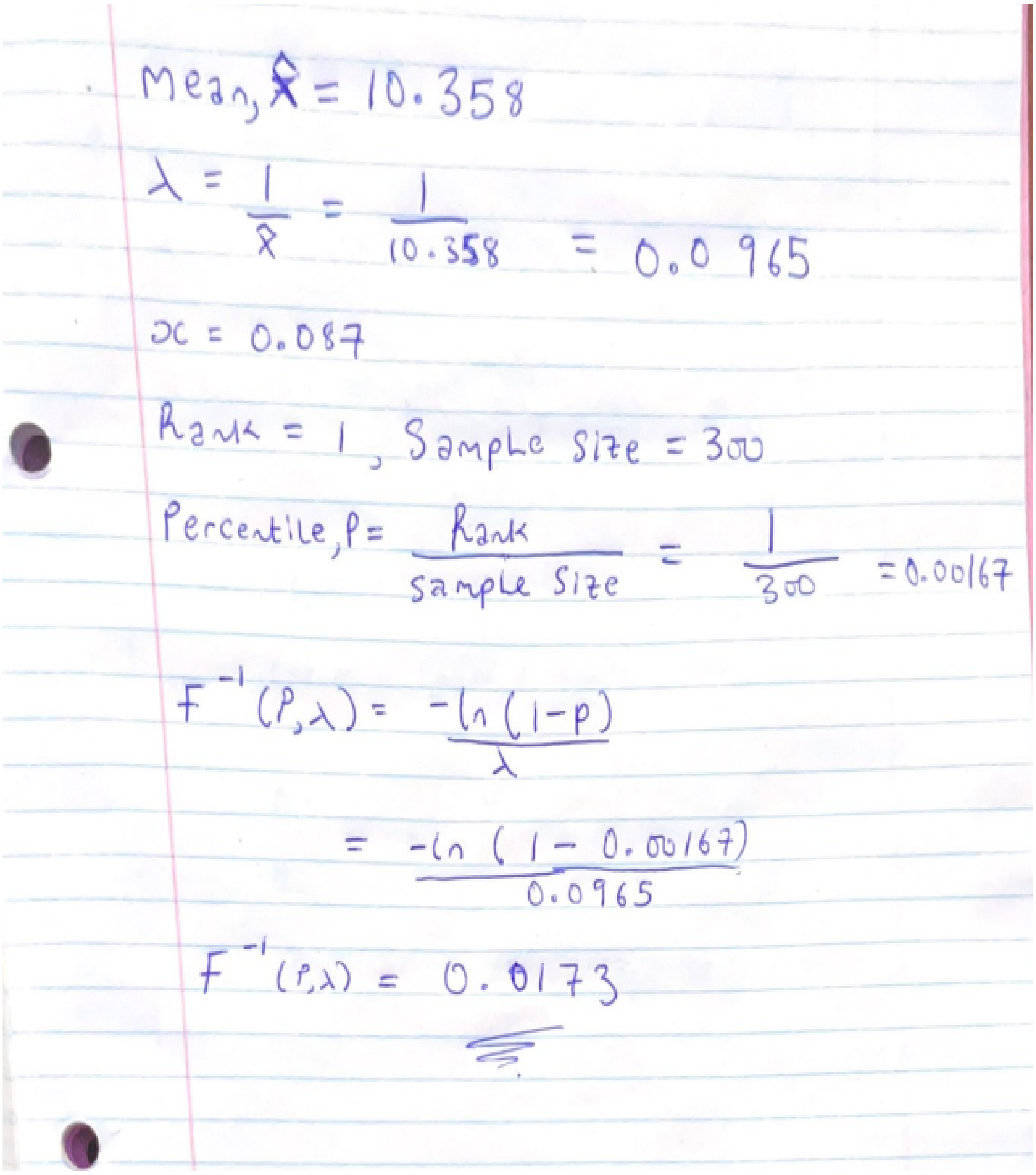
In this function, p represents the percentile varied between 0 and 1.

The value of lambda is 1 divided by the sample mean, as seen in the equation below:

$$\hat{\lambda} = \frac{1}{\bar{X}}$$

(From Table 3 in Chapter 9 of the Textbook)

Below is a sample calculation for the input from the inspector 1 data set:



Mean, $\hat{x} = 10.358$

$\lambda = \frac{1}{\hat{x}} = \frac{1}{10.358} = 0.0965$

$x = 0.087$

Rank = 1, Sample Size = 300

Percentile, $P = \frac{\text{Rank}}{\text{Sample Size}} = \frac{1}{300} = 0.00167$

$$F^{-1}(P,\lambda) = \frac{-\ln(1-p)}{\lambda}$$

$$= \frac{-\ln(1-0.00167)}{0.0965}$$

$$F^{-1}(P,\lambda) = 0.0173$$

The fit of the distribution is evaluated by how close data points are to a straight line on the graph. The closer the fit is to a straight line, the better the fit of the distribution to the data, and if, through visual inspection, we determine that the line on the QQ plot is sufficiently straight, then we will proceed to test the goodness of fit with Chi-Squared testing. The QQ plots made can be found in the appendix. Refer to the “Statistical_Analysis.xlsx” file, which contains the detailed calculations of the described steps above for all the sample data sets.

In conclusion, The QQ plots have the shape of a straight line with a slight tail showing that the hypothesis of an exponential distribution is correct. The slight tail's presence is because the extreme values' variance is higher than the middle's (Collected From Lecture Material Chapter 6 Slide 16).

Chi-Squared Testing:

A Chi-Squared test presents a null hypothesis stating that the distribution of a set of sample data points is the same as a specified hypothesized distribution, and the alternate hypothesis states that they are different. Testing if the input data fits a distribution using a chi-squared test is done by first sorting the input data from smallest to largest and then determining how many bins of what size to use to categorize the sample data. Once the bin count and sizes are known, the next step is to tabulate the observed frequencies in each bin in the same way it is done for making histograms. Then, an expected frequency number will need to be determined for each bin. This expected frequency number will be calculated by determining the difference between the CDFs of the upper and lower bounds of the current bin and multiplying the difference by the size of the dataset. Then for each bin, the formula below should be applied:

$$\frac{(O_i - E_i)^2}{E_i}$$

The Chi-Squared value is calculated using the formula:

$$\chi_0^2 = \sum_{i=1}^{k} \frac{(O_i - E_i)^2}{E_i}$$

This Chi-Squared value will be compared to a reference Chi-Squared value from a table whose value will depend on the number of bins and the alpha level of significance. If the table value is smaller than the calculated value, then the null hypothesis is rejected, and the distribution is not the same as the sample data. If the table value is larger than the calculated value, the null hypothesis will not be rejected. We can assume that the selected distribution is the same as the input data.

Assuming the distribution to be tested is continuous, the pdf or cdf can be computed as:

$$p_i = \int_{a_{i-1}}^{a_i} f(x)\,dx = F(a_i) - F(a_{i-1})$$

For the Chi-squared test for the exponential distribution with intervals of equal probability, selecting the class intervals is the first step. The recommendation from the textbook for a sample size greater than 100 should be between $\sqrt{n}$ to $n/5$. Given the sample size of 300 the number of class intervals, k will be between 17 and 60 (Table 5 Chapter 9). Based on that information let k = 20 and then each interval will have a probability, p = 0.05.

The cdf for the exponential distribution is given as:

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

Where ai is the endpoint at the ith interval, the value of lambda is 1 divided by the sample mean. The equation can be re-written as seen below because the cumulative area from zero to ai is eqaul to ip.

$$e^{-\lambda a_i} = 1 - ip$$

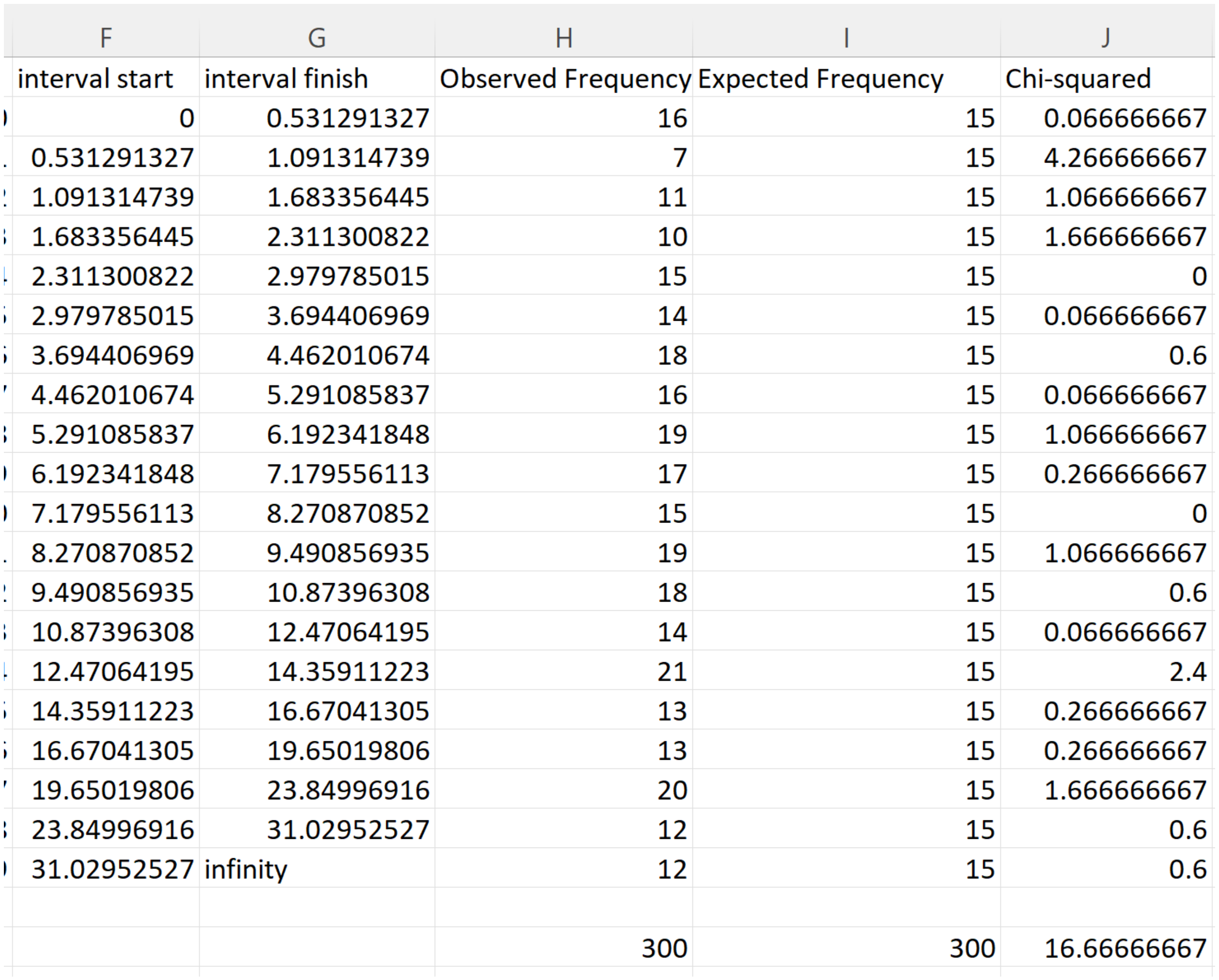
And ai can be calculated by taking the log of both sides.

$$a_i = -\frac{1}{\lambda}\ln(1 - ip), \quad i = 0, 1, \ldots, k$$

Regardless of the value of lambda a0 will always be zero and ak infinity.

The tables below correspond to the chi-squared testing tables for all 6 data files.

1. Inspector 1 Table



F	G	H	I	J
interval start	interval finish	Observed Frequency	Expected Frequency	Chi-squared
0	0.531291327	16	15	0.066666667
0.531291327	1.091314739	7	15	4.266666667
1.091314739	1.683356445	11	15	1.066666667
1.683356445	2.311300822	10	15	1.666666667
2.311300822	2.979785015	15	15	0
2.979785015	3.694406969	14	15	0.066666667
3.694406969	4.462010674	18	15	0.6
4.462010674	5.291085837	16	15	0.066666667
5.291085837	6.192341848	19	15	1.066666667
6.192341848	7.179556113	17	15	0.266666667
7.179556113	8.270870852	15	15	0
8.270870852	9.490856935	19	15	1.066666667
9.490856935	10.87396308	18	15	0.6
10.87396308	12.47064195	14	15	0.066666667
12.47064195	14.35911223	21	15	2.4
14.35911223	16.67041305	13	15	0.266666667
16.67041305	19.65019806	13	15	0.266666667
19.65019806	23.84996916	20	15	1.666666667
23.84996916	31.02952527	12	15	0.6
31.02952527	infinity	12	15	0.6
		300	300	16.66666667

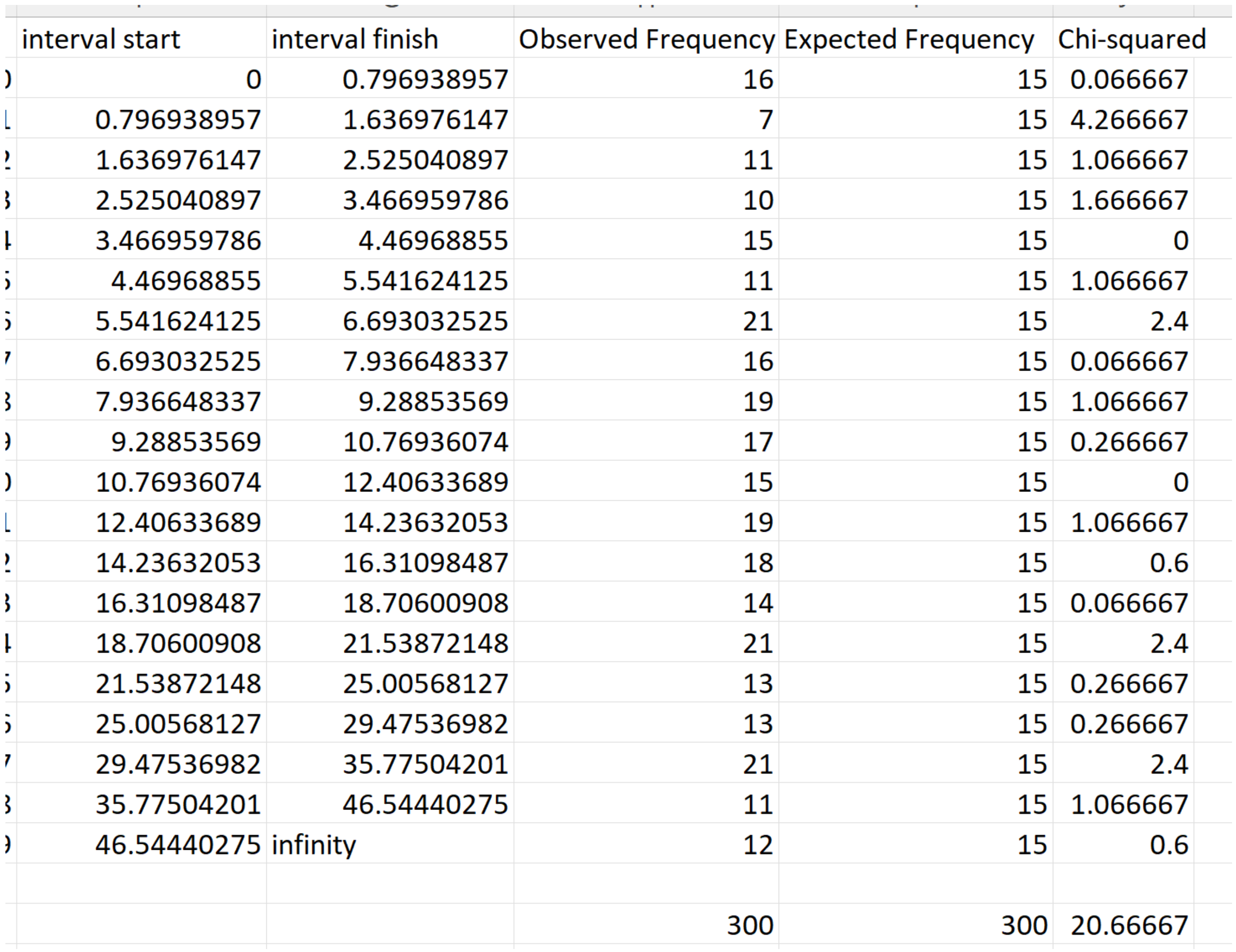
As shown in the table above x^2(0) = 16.67,

alpha = 0.05, s = 1, k - s - 1 = 18

x^2 @(0.05,18) = 28.869

x^2 @(0.05,18) > x^2(0), at 0.05 significance level the hypothesis is not rejected.

2. Inspector 2 Component 2 Table



interval start	interval finish	Observed Frequency	Expected Frequency	Chi-squared
0	0.796938957	16	15	0.066667
0.796938957	1.636976147	7	15	4.266667
1.636976147	2.525040897	11	15	1.066667
2.525040897	3.466959786	10	15	1.666667
3.466959786	4.46968855	15	15	0
4.46968855	5.541624125	11	15	1.066667
5.541624125	6.693032525	21	15	2.4
6.693032525	7.936648337	16	15	0.066667
7.936648337	9.28853569	19	15	1.066667
9.28853569	10.76936074	17	15	0.266667
10.76936074	12.40633689	15	15	0
12.40633689	14.23632053	19	15	1.066667
14.23632053	16.31098487	18	15	0.6
16.31098487	18.70600908	14	15	0.066667
18.70600908	21.53872148	21	15	2.4
21.53872148	25.00568127	13	15	0.266667
25.00568127	29.47536982	13	15	0.266667
29.47536982	35.77504201	21	15	2.4
35.77504201	46.54440275	11	15	1.066667
46.54440275	infinity	12	15	0.6
		300	300	20.66667

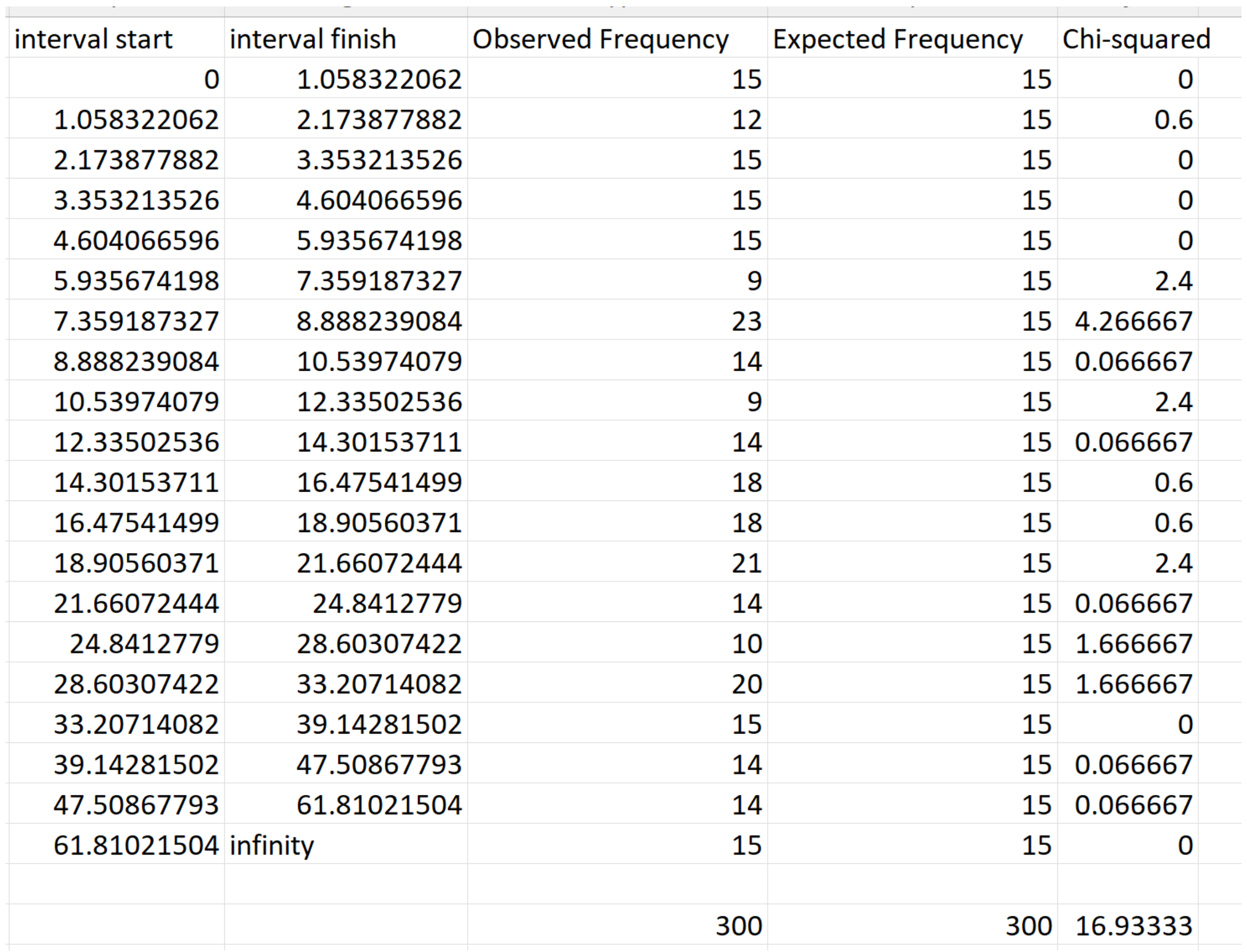
As shown in the table above x^2(0) = 20.67,

alpha = 0.05, s = 1, k - s - 1 = 18

x^2 @(0.05,18) = 28.869

x^2 @(0.05,18) > x^2(0), at 0.05 significance level the hypothesis is not rejected.

3. Inspector 2 Component 3 Table



interval start	interval finish	Observed Frequency	Expected Frequency	Chi-squared
0	1.058322062	15	15	0
1.058322062	2.173877882	12	15	0.6
2.173877882	3.353213526	15	15	0
3.353213526	4.604066596	15	15	0
4.604066596	5.935674198	15	15	0
5.935674198	7.359187327	9	15	2.4
7.359187327	8.888239084	23	15	4.266667
8.888239084	10.53974079	14	15	0.066667
10.53974079	12.33502536	9	15	2.4
12.33502536	14.30153711	14	15	0.066667
14.30153711	16.47541499	18	15	0.6
16.47541499	18.90560371	18	15	0.6
18.90560371	21.66072444	21	15	2.4
21.66072444	24.8412779	14	15	0.066667
24.8412779	28.60307422	10	15	1.666667
28.60307422	33.20714082	20	15	1.666667
33.20714082	39.14281502	15	15	0
39.14281502	47.50867793	14	15	0.066667
47.50867793	61.81021504	14	15	0.066667
61.81021504	infinity	15	15	0
		300	300	16.93333

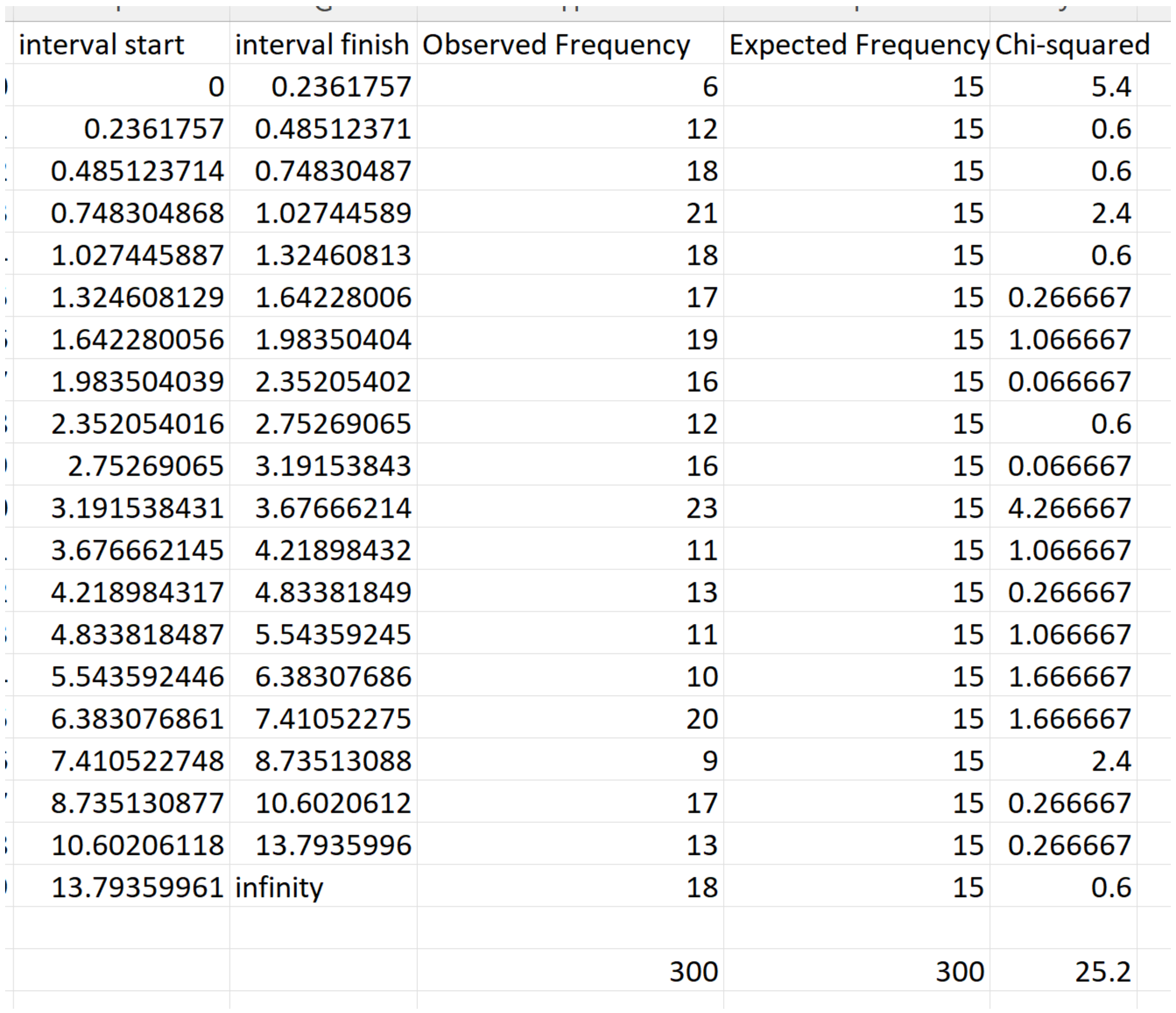
As shown in the table above x^2(0) = 16.93,

alpha = 0.05, s = 1, k - s - 1 = 18

x^2 @(0.05,18) = 28.869

x^2 @(0.05,18) > x^2(0), at 0.05 significance level the hypothesis is not rejected.

4. Workstation 1 Table



interval start	interval finish	Observed Frequency	Expected Frequency	Chi-squared
0	0.2361757	6	15	5.4
0.2361757	0.48512371	12	15	0.6
0.485123714	0.74830487	18	15	0.6
0.748304868	1.02744589	21	15	2.4
1.027445887	1.32460813	18	15	0.6
1.324608129	1.64228006	17	15	0.266667
1.642280056	1.98350404	19	15	1.066667
1.983504039	2.35205402	16	15	0.066667
2.352054016	2.75269065	12	15	0.6
2.75269065	3.19153843	16	15	0.066667
3.191538431	3.67666214	23	15	4.266667
3.676662145	4.21898432	11	15	1.066667
4.218984317	4.83381849	13	15	0.266667
4.833818487	5.54359245	11	15	1.066667
5.543592446	6.38307686	10	15	1.666667
6.383076861	7.41052275	20	15	1.666667
7.410522748	8.73513088	9	15	2.4
8.735130877	10.6020612	17	15	0.266667
10.60206118	13.7935996	13	15	0.266667
13.79359961	infinity	18	15	0.6
		300	300	25.2

As shown in the table above x^2(0) = 25.2,

alpha = 0.05,

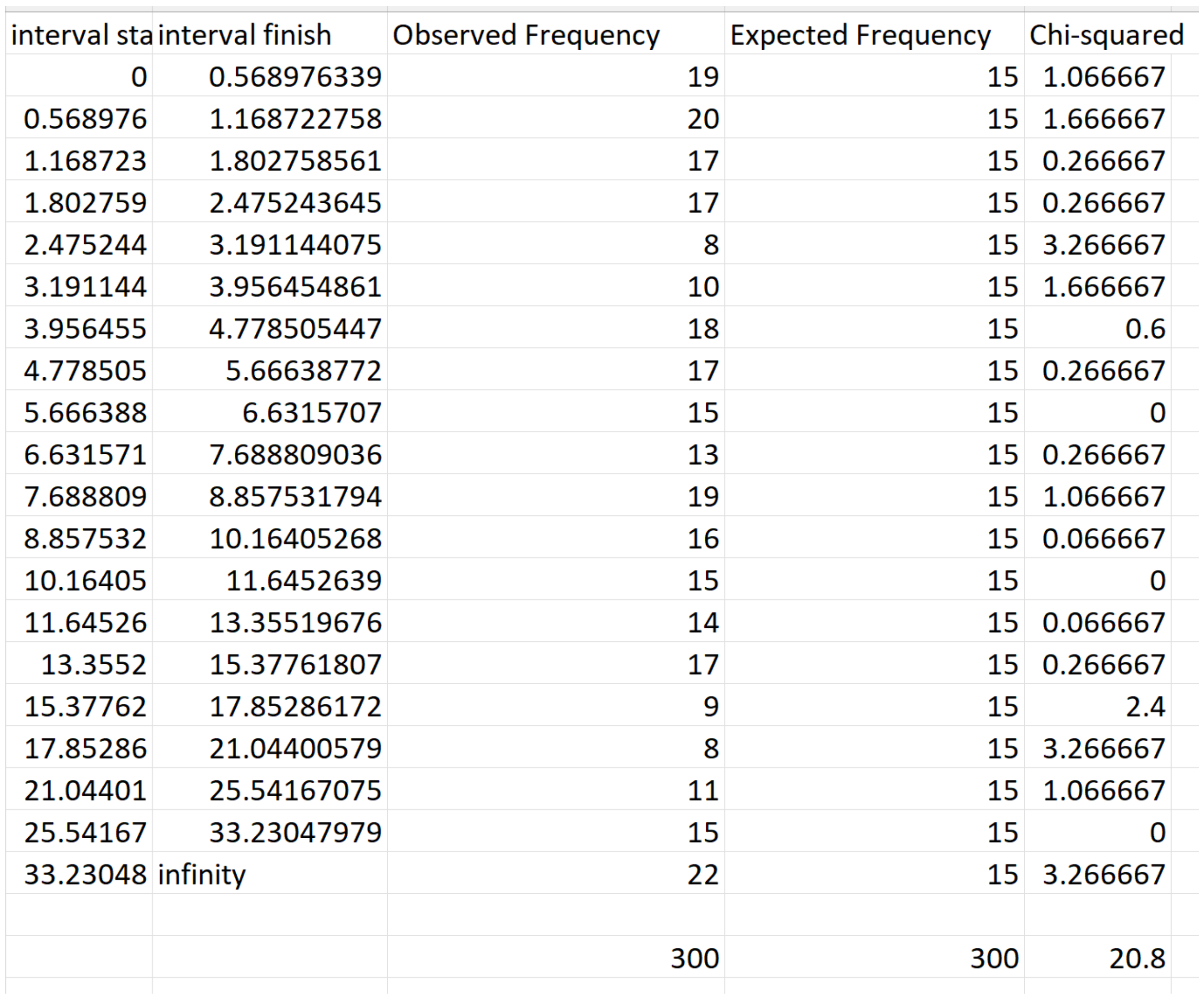
s = 1

k - s - 1 = 18

x^2 @(0.05,18) = 28.869

x^2 @(0.05,18) > x^2(0), at 0.05 significance level the hypothesis is not rejected.

5. Workstation 2 Table



interval sta	interval finish	Observed Frequency	Expected Frequency	Chi-squared
0	0.568976339	19	15	1.066667
0.568976	1.168722758	20	15	1.666667
1.168723	1.802758561	17	15	0.266667
1.802759	2.475243645	17	15	0.266667
2.475244	3.191144075	8	15	3.266667
3.191144	3.956454861	10	15	1.666667
3.956455	4.778505447	18	15	0.6
4.778505	5.66638772	17	15	0.266667
5.666388	6.6315707	15	15	0
6.631571	7.688809036	13	15	0.266667
7.688809	8.857531794	19	15	1.066667
8.857532	10.16405268	16	15	0.066667
10.16405	11.6452639	15	15	0
11.64526	13.35519676	14	15	0.066667
13.3552	15.37761807	17	15	0.266667
15.37762	17.85286172	9	15	2.4
17.85286	21.04400579	8	15	3.266667
21.04401	25.54167075	11	15	1.066667
25.54167	33.23047979	15	15	0
33.23048	infinity	22	15	3.266667
		300	300	20.8

As shown in the table above x^2(0) = 20.8,

alpha = 0.05,

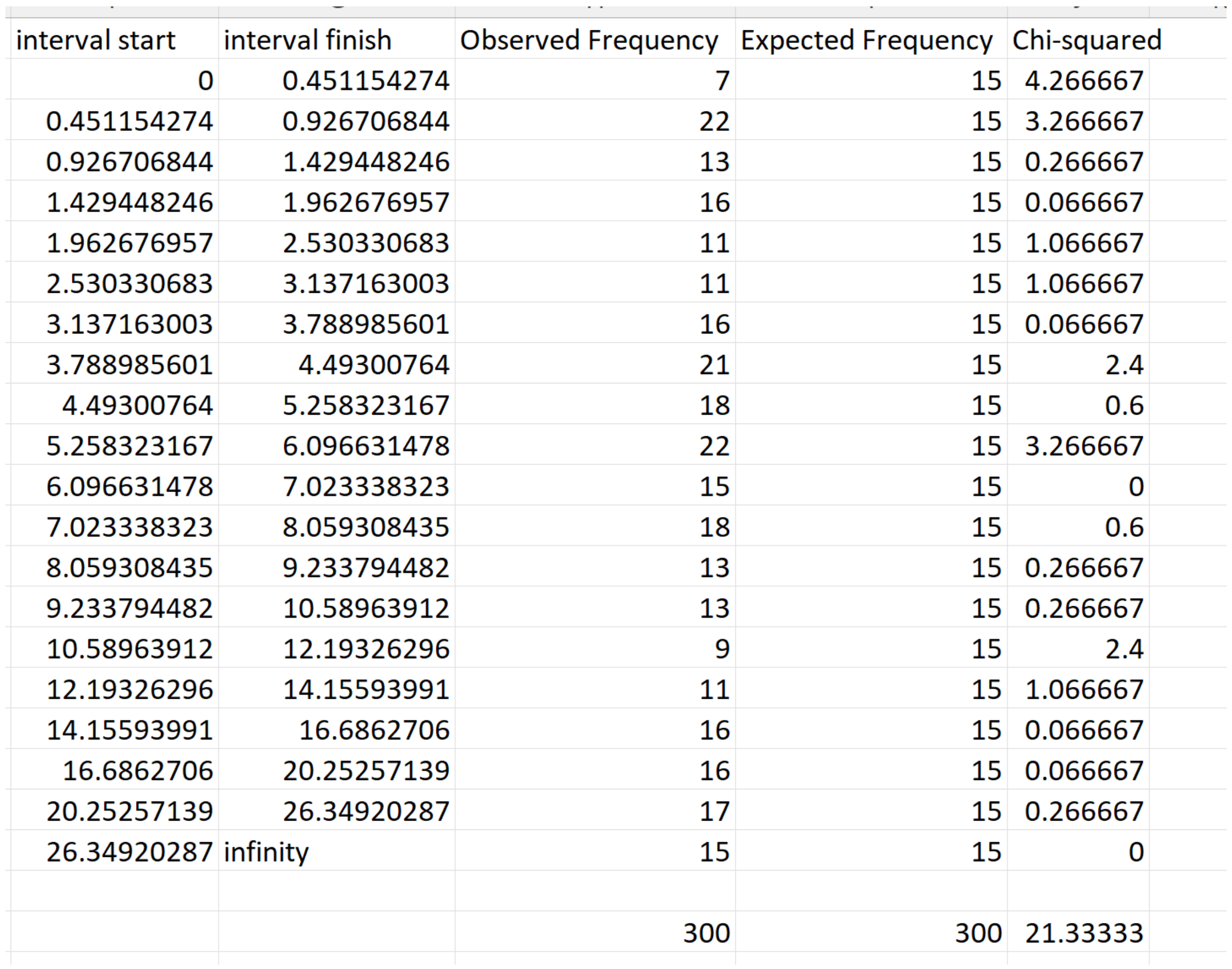
s = 1

k - s - 1 = 18

x^2 @(0.05,18) = 28.869

x^2 @(0.05,18) > x^2(0), at 0.05 significance level the hypothesis is not rejected.

6. Workstation 3 Table



interval start	interval finish	Observed Frequency	Expected Frequency	Chi-squared
0	0.451154274	7	15	4.266667
0.451154274	0.926706844	22	15	3.266667
0.926706844	1.429448246	13	15	0.266667
1.429448246	1.962676957	16	15	0.066667
1.962676957	2.530330683	11	15	1.066667
2.530330683	3.137163003	11	15	1.066667
3.137163003	3.788985601	16	15	0.066667
3.788985601	4.49300764	21	15	2.4
4.49300764	5.258323167	18	15	0.6
5.258323167	6.096631478	22	15	3.266667
6.096631478	7.023338323	15	15	0
7.023338323	8.059308435	18	15	0.6
8.059308435	9.233794482	13	15	0.266667
9.233794482	10.58963912	13	15	0.266667
10.58963912	12.19326296	9	15	2.4
12.19326296	14.15593991	11	15	1.066667
14.15593991	16.6862706	16	15	0.066667
16.6862706	20.25257139	16	15	0.066667
20.25257139	26.34920287	17	15	0.266667
26.34920287	infinity	15	15	0
		300	300	21.33333

As shown in the table above x^2(0) = 21.33,

alpha = 0.05,

s = 1

k - s - 1 = 18

x^2 @(0.05,18) = 28.869

x^2 @(0.05,18) > x^2(0), at 0.05 significance level the hypothesis is not rejected.

The following table shows a summary of the chi-squared tests.:

	Lambda	Chi Total	Fail to Reject Null Hypothesis?	Distribution Type
Inspector 1	0.096545	16.67	Yes	Exponential
Inspector 2_2	0.064363	20.67	Yes	Exponential
Inspector 2_3	0.048467	16.93	Yes	Exponential
Workstation 1	0.217183	25.2	Yes	Exponential
Workstation 2	0.09015	20.8	Yes	Exponential
Workstation 3	0.113693	21.33	Yes	Exponential

Testing all six of the data sets results in the Null hypothesis not being rejected, and we can assume that the distribution does match. This list of 300 random variables using the parameters was compared to the distribution using the same parameters. These tests were also successful in confirming and reinforcing our distribution parameter selections.

6. Input Generation

Generating Random Numbers

To generate randomized numbers for the simulation's component inspection times and workstation build times, a random number generation system was manually implemented and tested. For a generated number to be considered random, it should be independent and uniformly distributed. The random number generation model would be the multiplicative congruential model (MCM), based on the linear congruential model with a zero incrementor.

Multiplicative Congruential Model:

Random number generation with the multiplicative congruential model is done with a recursive method as shown below:

$$X_i = (aX_{i-1}) \% m$$

Where X represents a randomly selected number, a is the multiplier parameter, m is the modulus parameter, and % is the modulus operator. Since the current random number to generate depends on the previously generated number, the user must define an initial seed value to set as the initial 'X_{i-1}' value so subsequent numbers can be generated. Once a number is generated, the value for 'X_{i-1}' will be changed to X_i. The numbers generated are integers in the range of [0, m-1].

The integers are converted to random numbers using the equation below:

$$R_i = \frac{X_i}{m}$$

To ensure this generator is good, a large integer for modulus m will be chosen. Refer to the "Multiplicative_Congruential_Model.py" module in the source code for implementation.

Testing Random Number Generation

To ensure that the randomly generated numbers are truly random, we shall test for uniformity and independence with 95% confidence. Independence will be tested using an Autocorrelation test, and uniformity will be tested using a Kolmogorov-Smirnov test.

Autocorrelation Testing:

Autocorrelation is used to determine if the random number generation process produces independent numbers. The test assesses if subsequently selected random numbers tend to follow a pattern. For example, are low numbers followed by higher numbers, or check if high numbers follow high numbers, etc. The null hypothesis of this test states that the numbers are independent, and the alternate hypothesis is that they are not. This process first involves setting and calculating the below variables, and the values used for testing this project are shown below:

- $i = 1$, the *starting index in the set of random numbers*
- $m = 1$ (*lag* value from lecture), *how many numbers to skip between each choice*
- $M = 898$, *Largest integer that makes* $i + (M + 1)m \leq N$ *true*
- $\alpha = 0.05$, *confidence*. ($\alpha/2$) from *Z-Table* equals the *value to beat*
- $N = 900$, *Total number of random numbers to generate*

The formula retrieved from the textbook used is given below:

$$\hat{\rho}_{i\ell} = \frac{1}{M+1}\left[\sum_{k=0}^{M} R_{i+k\ell}R_{i+(k+1)\ell}\right] - 0.25$$

$$\sigma_{\hat{\rho}_{i\ell}} = \frac{\sqrt{13M+7}}{12(M+1)}$$

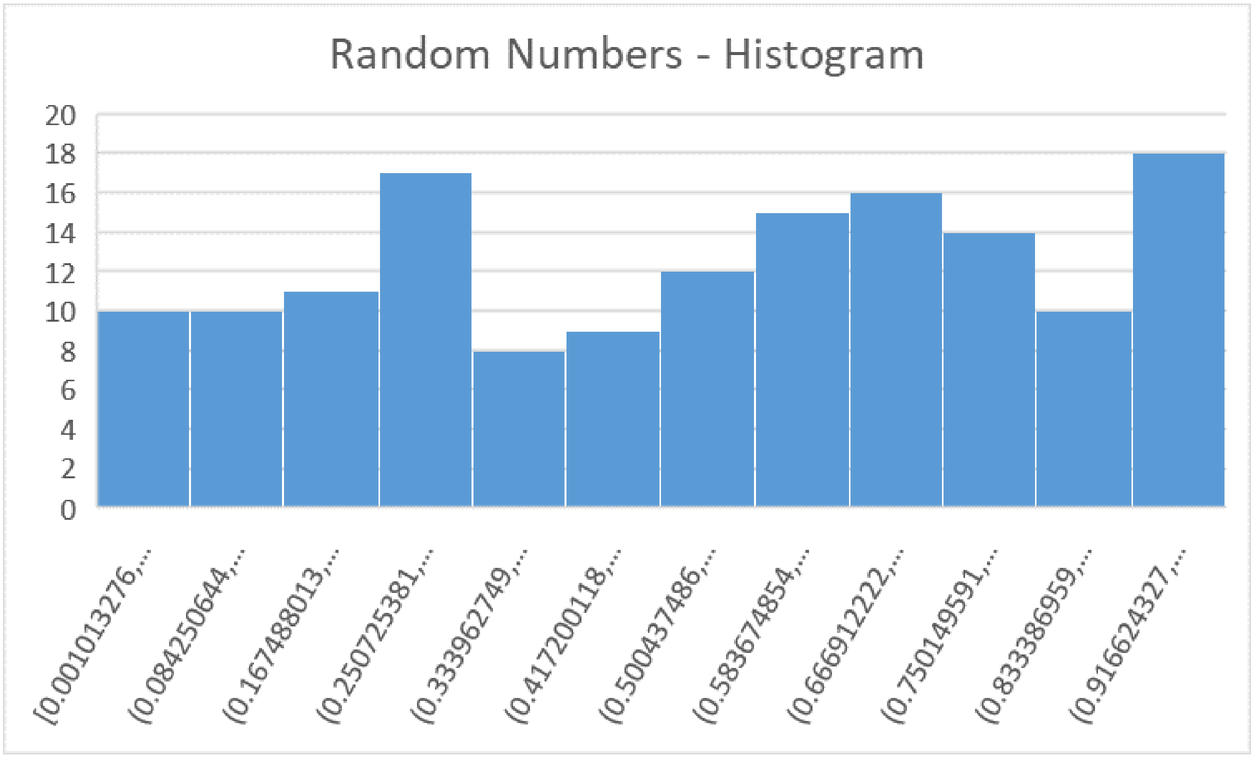
$$\frac{\rho}{\sigma} \leq Z_{table} = 1.96$$

If the value of calculated above, is less than the value from the z table then the null hypothesis is not rejected and then we can conclude the numbers are independent with a confidence of 95%. Alternatively, the null hypothesis is rejected, and the numbers are dependent. Refer to the “Multiplicative_Congruential_Model.py” module in the source code for the implementation of the tests.

Kolmogorov-Smirnov Testing:

To determine if the random number generation process produces uniform values, a Kolmogorov-Smirnov test will compare a set of randomly generated numbers to a uniform distribution. This test will compare a uniform distribution's continuous cumulative distribution function to the empirical cumulative distribution function observed from a set of random samples. The null hypothesis of this test states that there is no difference between the distribution of the random numbers and uniform distribution, and the alternate hypothesis states that they are different. The first step is to generate a set of random numbers. 150 random numbers will be generated for the testing in this project, and this set of random numbers shall be sorted from smallest to largest. Another set of 150 numbers will represent the expected cumulative distribution function values, ordered from smallest to largest. Given that there are 150 random samples to match, this list will also have 200 values, starting at 0.01, with each element being 0.01 larger than the previous, [0.01, 0.02, 0.03, . . . 0.98, 0.99, 1.0].

The figure below is a histogram of randomly generated variables.



Given these sorted lists of random numbers and expected cumulative distributions, two more lists will be made, taking in corresponding values from the two lists. The first list will be called the D_plus list and each element will be the calculation result of the

(current expected value - current sorted random number).

The second list will be called D_minus, and each element of the list will be calculated as

(current random number - previous expected value).

The first element of D-minus will use previousExpectedValue=0, and the second element in D_minus will be

(2nd smallest random number) - (0.01 * previous expected value)

Once the D_plus and D_minus lists are made, the most significant contained value between both of them will be retrieved and will be referred to as D_max. This D_max value will then be compared to the table-based D_alpha value and using alpha=0.05, D_alpha=1.36/sqrt(150)

If the D_max value is less than the D_alpha table value, then the null hypothesis will not be rejected and it can be assumed that the generated numbers are uniformly distributed. Otherwise, the null hypothesis will be rejected and it can be concluded that the randomly generated values are not uniformly distributed.

After performing the test on the random number generator, the null hypothesis was not rejected, meaning the generator is uniform. Refer to the "Multiplicative_Congruential_Model.py" module in the source code for the implementation of the test.

Inverse-Transform Technique:

For generating random variates, the preferred method is the inverse-transform technique. The inverse-transform technique requires an invertible cumulative distribution function for the distribution to generate from. Once the inverse of a distribution's cumulative distribution function is determined, a randomly selected number from a uniform distribution will become the input parameter, and the calculation output will be a random variate. This technique will be used for uniform, exponential, Weibull, and triangular distributions, and given that all distributions were determined to be exponential, this technique will be used throughout.

The implication to Simulation Code:

Upon the completion of all these analyses, The randomly generated service times for each entity of the system to be simulated the inverse transform techniques for generating random variates will be adopted. The exponential inverse CDF function is used for generating services times in the simulation as shown below:

```
# Lambda = 1 / sample_mean
```

```
time = expo_inverse_cdf(self.rand_generator.get_next_r(), 0.096544573)
```

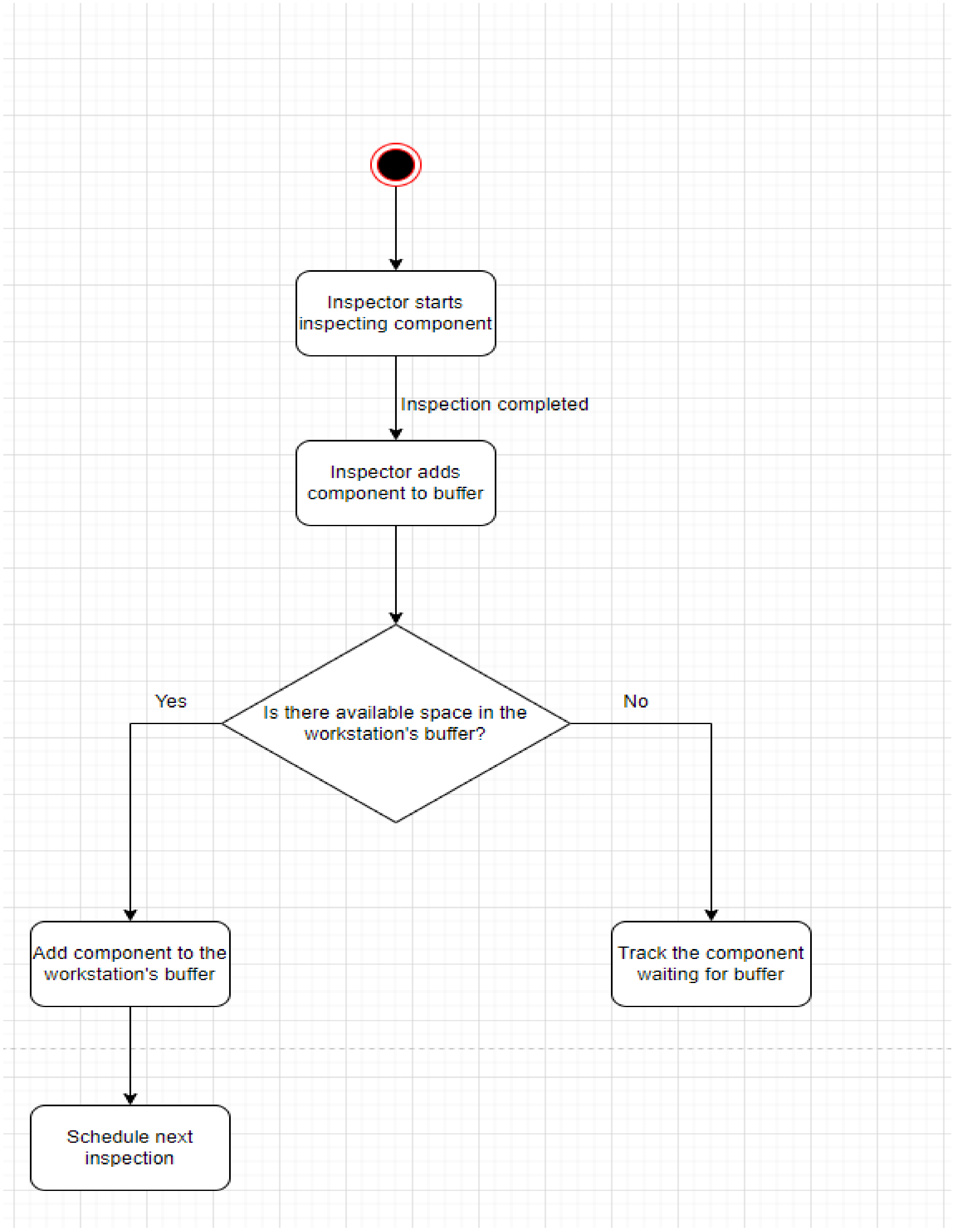
The "rand_generator" refers to the multiplicative congruential model for generating random variables. "0.096544573" is the lambda value calculated by using the inverse of the sample mean. This functionality is applied to the inspectors and workstation modules for generating a random delay time. They are biased by the mean from the sample data set. Refer to lines 51, 116 and 118 of the "Inspector.py" module and line 54 of the "Workstation.py" module for implementation.

7. Verification

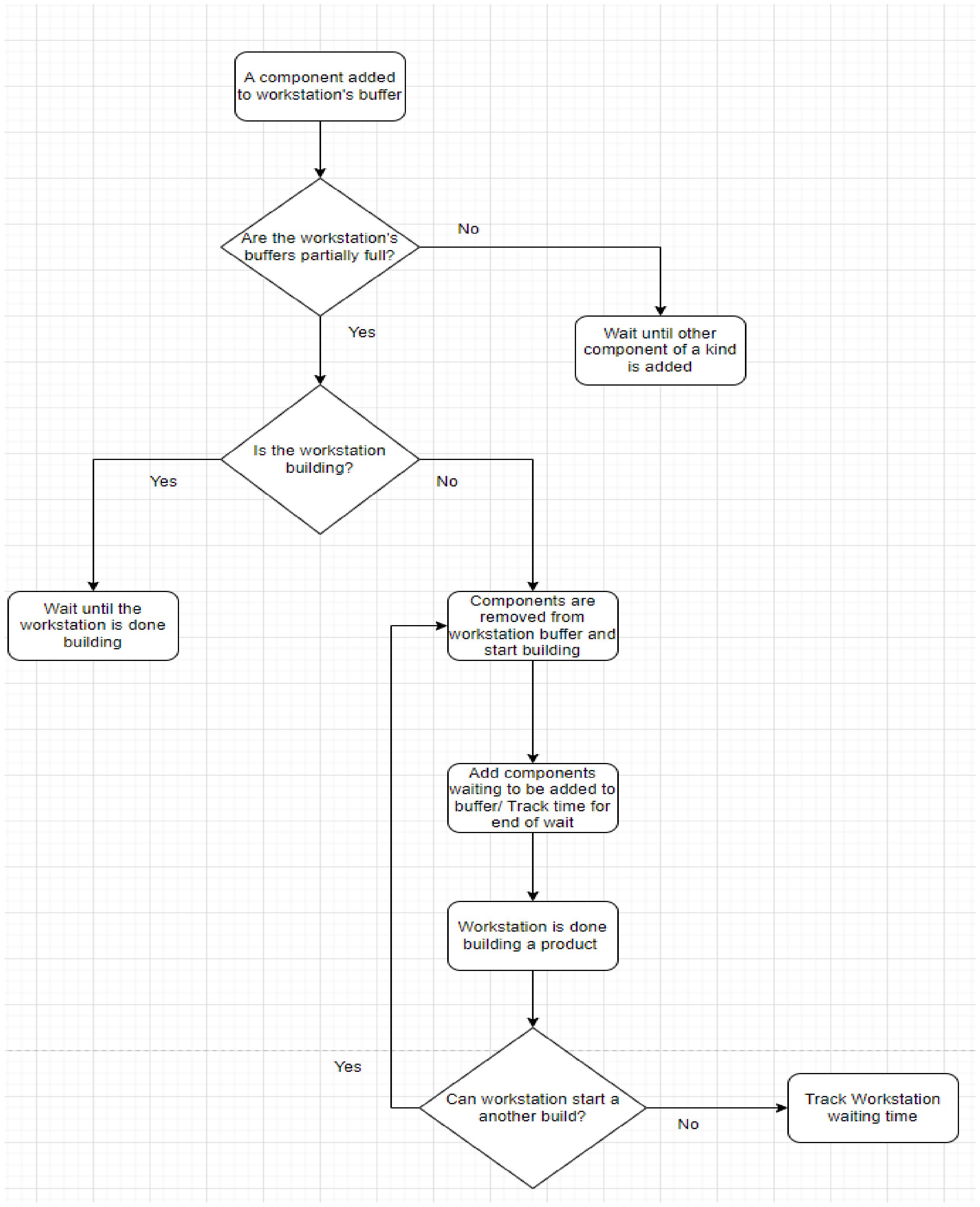
Model verification was done to ensure the conceptual model is reflected accurately in the operational model. This is to verify that the simulation is operating as intended. The conceptual model and operational model will be independently verified to ensure that the model accurately represents the manufacturing process described in the project description.

The first verification technique is having the model reviewed independently. This was accomplished by having each team member verify that the model was implemented correctly in the python code by reviewing the parts written by other team members. This helped to eliminate any discrepancies while developing the model.

To verify the conceptual model, a flowchart has been created to show how future simulation events shall be generated based on the initial events, and this flowchart can be found below. The figure below shows the flow of execution on the side of the inspector:


Inspector starts inspecting component
Inspection completed
Inspector adds component to buffer
Is there available space in the workstation's buffer?
Yes
No
Add component to the workstation's buffer
Track the component waiting for buffer
Schedule next inspection

The next figure below shows the flow of execution of the workstation logic of the simulation:


A component added to workstation's buffer
Are the workstation's buffers partially full?
No
Wait until other component of a kind is added
Yes
Is the workstation building?
Yes
Wait until the workstation is done building
No
Components are removed from workstation buffer and start building
Add components waiting to be added to buffer/ Track time for end of wait
Workstation is done building a product
Can workstation start a another build?
Yes
No
Track Workstation waiting time

Once the flowchart’s process, assumptions, abstractions, simplifications, and parameters are conceptually verified, we shall proceed to the operational model verification process. This was done by running the simulation and analyzing the line-by-line execution of event handling to make sure that the progression of the simulation program is consistent with the conceptual model flow chart.

8. Validation

The process of validating a simulation model involves comparing the model outputs to real-life observations. This determines if the model is a consistent and accurate representation of the real system. The project models a manufacturing facility, with the given collections of sets of historical data for component inspection and product build times, it can be assumed that the manufacturing facility can be used to obtain a historical dataset that corresponds with the output variables of interest in the simulation model.

A group of simulation-generated output variables can be used to calculate a simulation mean & standard deviation given the reported real values for the desired output variables. In order to match the output variables' confidence intervals to the measured values, this is done. The model can be approved if the computed confidence range from the simulation contains the observed value and the worst-case error is less than or equal to the epsilon error level. More simulation replications will be needed to reach a determination if the worst-case error is higher than the epsilon error cutoff value with the confidence interval comprising the real value.

Since the actual version of the manufacturing process detailed in this project is hypothetical, there is currently no way to know the actual observed values of the output variables, thus the above method of using simulation confidence intervals to reach a validation conclusion will only be partially possible. The simulation output variable confidence intervals will still be generated, but the validation will be done by ensuring that the output variables are reasonably close to each other between different simulation runs. This will ensure the output variables are reasonably accurate and accessing it could be deferred to a later task when the actual output variables are known. Given a confidence interval of the simulation output parameters, the width can provide insight into what epsilon error threshold values can lead to accepting the model’s implementation. The confidence intervals for the simulation's output variables with the standard operating policy can be found in the below section comparing the standard and alternate operating policies.

9. Production Runs and Analysis

Following the assumption that the implementation of the simulation has been verified and validated, we can then proceed to run multiple simulation replications so that confidence intervals for the output.

Number of Replicants:

To determine the appropriate number of replications, after 100 replications for each component, the means and standard deviations of the waiting times were determined for each component and the following equation was used to estimate the ideal number of replications, R for this simulation.

$$R \geq \left(\frac{z_{\alpha/2} S_0}{\epsilon} \right)^2$$

Where R is the number of replications, z is the z-value at alpha divided by 2, alpha is 0.05, So is the standard deviation of the data and epsilon is the error threshold of the data. This equation was applied to the mean and standard deviation of the blocked times of the simulation and given a 10% error value. It was concluded that the maximum replication value needed for this simulation has an estimated value of 632.

Initialization Phase:

While initializing a simulation, there may be a bias as a result of the initial conditions of the simulation being an unreliable representation of the model's actualities. To mitigate this bias, the simulation is divided into an initialization phase and a data collection phase. Data is not gathered during the initialization process, which is when the simulation can build up to the regular operation. Data gathering can be relied upon to start once the process has ramped up to steady-state circumstances. To implement this initialization phase, an event could be handled at the precise moment the initialization phase ends and the production phase begins to reset all of the output variables and exclude all events that occurred during the initialization phase from consideration for any subsequent adjustments.

As the duration of each simulation repetition grows, the effect of initialization bias on the output variables becomes less important, it was determined that for this project the effort and time funding needed to execute such a system would have limited value. Currently, a single simulation replication stops after 50,000 products are made, and given that the maximum difference between the initial condition of 5 empty buffers and the max capacity condition of 10 items across all buffers is very small by comparison. The initialization period was deemed unnecessary for the simulation. This design decision might result in wider confidence intervals for the output variables due to a higher degree of output variable variance, but the breadth of the confidence interval can be decreased by raising the number of replications.

Confidence Interval:

We evaluated the length of the initialization phase and the number of replications needed to achieve 95% confidence intervals with widths that did not exceed 20% of the predicted values in to determine the steady-state estimations of the variables of interest. After that, we carried out the production runs and determined the 95% confidence interval for the desired attributes.

10. Alternate Operating Policy

The alternate operating policy used in this project involves updating how inspector 2 behaves while selecting which component to inspect next. Rather than randomly selecting the next component to inspect it makes an informed selection based on the last inspection. Inspector 1's logic will remain unchanged as the regular operating policy used in previous iterations is already operating properly. We believe doing this will decrease the wait time for Inspector 1 as more efficient inspections by Inspector 2 will improve the production rates of Workstation 1.

This alternate operating policy was implemented by adding a new method to the "Inspector.py" module called get_alternate_policy_inspect_time which returns component C2 and a simulated inspection delay if component C3 was inspected last by Inspector 2 and vice versa. During initialization, the first component to be inspected by Inspector 2 is still randomly selected similar to how it is in the regular policy, but every subsequent one is based on the previously inspected.

In the "Simulation.py" module, there is a boolean flag to decide whether or not the alternate policy is in use by the simulation. If it is set to False then the simulation runs based on the default parameters. This flag is used to determine the method of generating inspection time and components for inspector 2. This means either using the "generate_inspect_time" or "get_alternate_policy_inspect_time". Implementing this flag makes it easy to switch between the default and alternate operating policies. Using the same simulation metrics and variables derived above the different operating policies can be compared to determine if the performance of the alternate operating policy will increase manufacturing efficiency.

[Please Refer To Source Code For Discussed Changes]

11. Operating Policy Comparison Using CRN

The following below are the outputs derived from running the simulation with the same parameters on the two operating policies.

Default Operating Policy:

```
C1 Average waiting time:
Mean: 1316.14
Standard deviation: 804.72
n: 632

C2 Average waiting time:
Mean: 59340.31
Standard deviation: 57033.20
n: 632

C3 Average waiting time:
Mean: 8361.26
Standard deviation: 3340.49
n: 632
```

Alternate Operating Policy:

```
C1 Average waiting time:
Mean: 816.84
Standard deviation: 562.54
n: 632

C2 Average waiting time:
Mean: 42274.06
Standard deviation: 52725.14
n: 632

C3 Average waiting time:
Mean: 5327.54
Standard deviation: 2060.36
n: 632
```

Comparing these output variables using the common random numbers (CRN) method, involves using the following equations to determine the confidence intervals of their differences:

$$\bar{Y}_{.1} - \bar{Y}_{.2} \pm t_{\alpha/2,\upsilon} s.e.(\bar{Y}_{.1} - \bar{Y}_{.2})$$

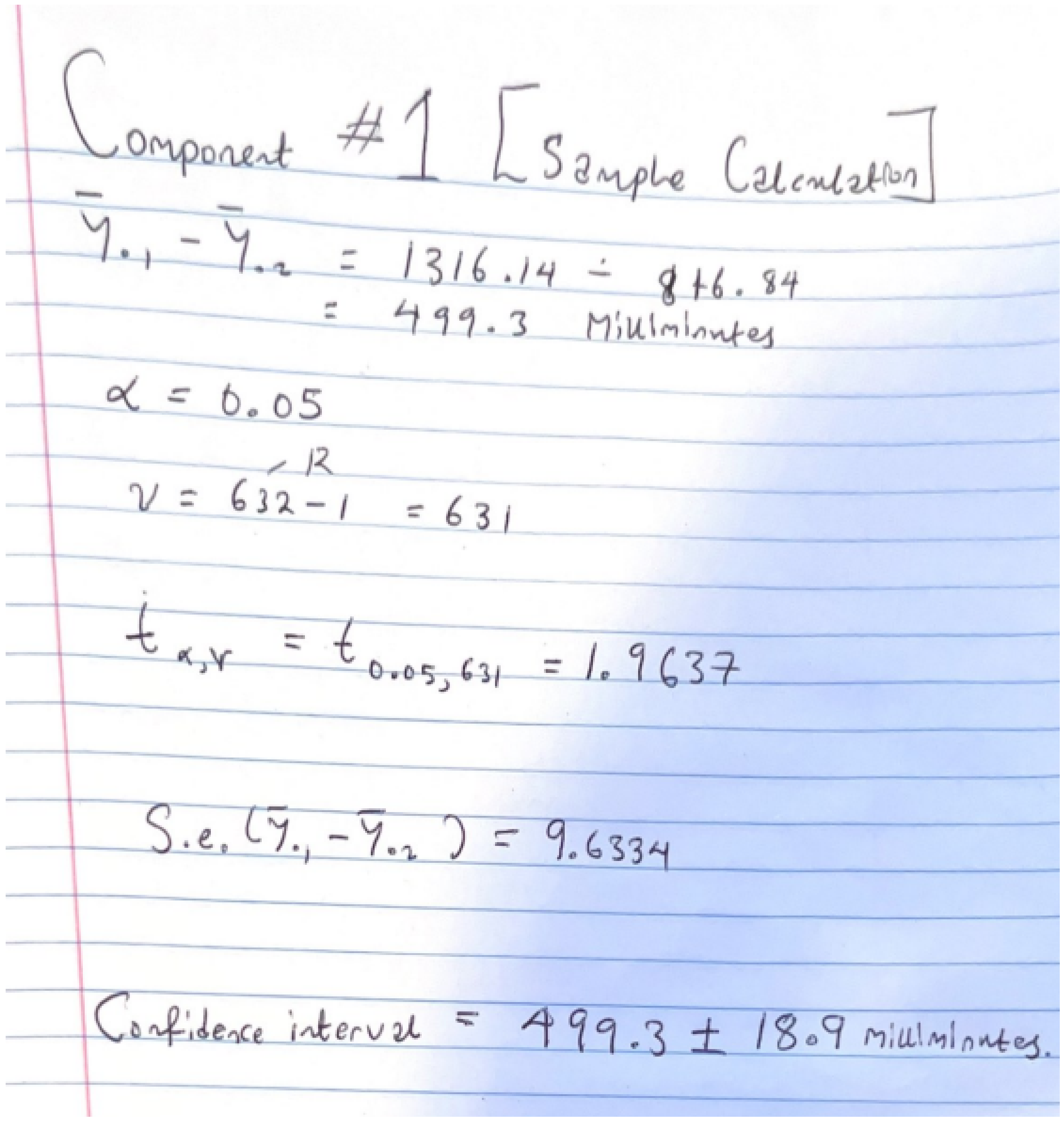
Assuming the independent sampling the equation below is used to calculate the standard error, “s.e.”:

$$s.e.(\bar{D}) = s.e.\left(\bar{Y}_{.1} - \bar{Y}_{.2}\right) = \frac{S_D}{\sqrt{R}}$$

The degrees of freedom, v can be calculated using the equation below where R is the number of replicants:

$$v = R - 1$$

Confidence Intervals:



Component #1 [Sample Calculation]

$\bar{y}_{\cdot 1} - \bar{y}_{\cdot 2} = 1316.14 \div 816.84$

$= 499.3$ Milliminutes

$\alpha = 0.05$

$\nu = \overset{R}{632} - 1 = 631$

$t_{\alpha,\nu} = t_{0.05,631} = 1.9637$

$s.e.(\bar{y}_{\cdot 1} - \bar{y}_{\cdot 2}) = 9.6334$

Confidence interval $= 499.3 \pm 18.9$ milliminutes.

Component 2 C. I = 17066.25 +/- 336.51 milliminutes
Component 3 C. I = 3033 +/- 99.93 milliminutes

Based on the above confidence intervals, the ranges are completely positive. This implies that the alternate operating policy's mean component block timings are substantially lower than the regular operating policy's mean component block time.

12. Conclusion

As shown in section 11 above, There is an increase in performance while using the alternate operating policy. We believe this is because Inspector 2 will no longer randomly guess what component to inspect. Instead, make an informed decision based on the state of the simulation variables. The alternative policy can prevent the possibility of inspector 2 choosing the same component repeatedly, which would completely block one of the workstations from building components.

There would most certainly be expenses connected with giving Inspector 2 the resources it would need to adapt its operating policy to reflect the alternate operating policy created in the simulation model. To link the outcomes of this model to the "real life" manufacturing facility, It can be assumed that the manufacturing process will continue for a significant amount of time, produce a large number of finished products and that the price of the finished products is adequate given that the manufacturing facility typically operates with very substantial capital investment and the need to operate on a large scale to be financially sustainable. The economic benefits obtained from the increased efficiency of the alternate operating policy implementation will grow as the manufacturing process is continued for longer periods of time. These economic benefits should eventually outweigh the resource costs of implementing the alternate operating policy, justifying its implementation.

In conclusion, it is also important to note that the above result is based on a very simplified simulation of a hypothetical manufacturing facility. An actual Manufacturing facility at a large scale is a complicated process comprising many unknown variables and non-linear actions. Simulation results and conclusions when it comes to making business decisions, should take precedence over empirically measured performance data.

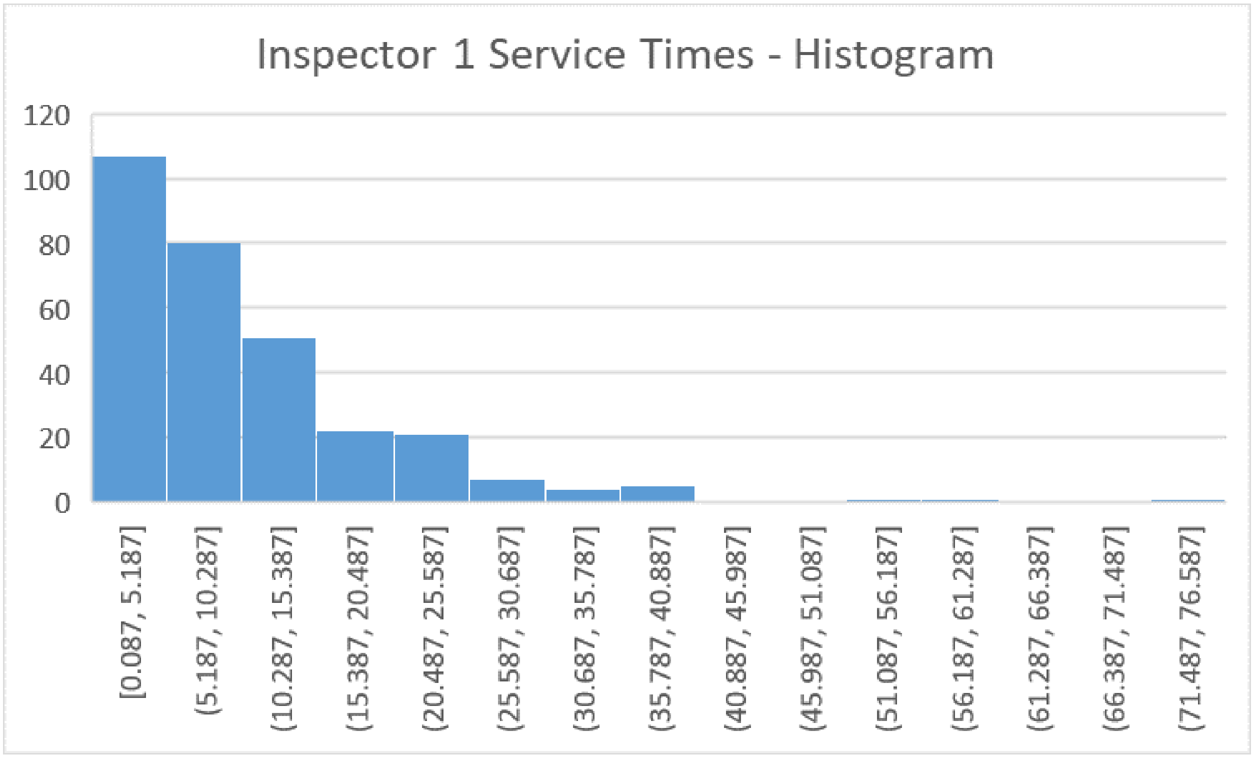
Appendices:

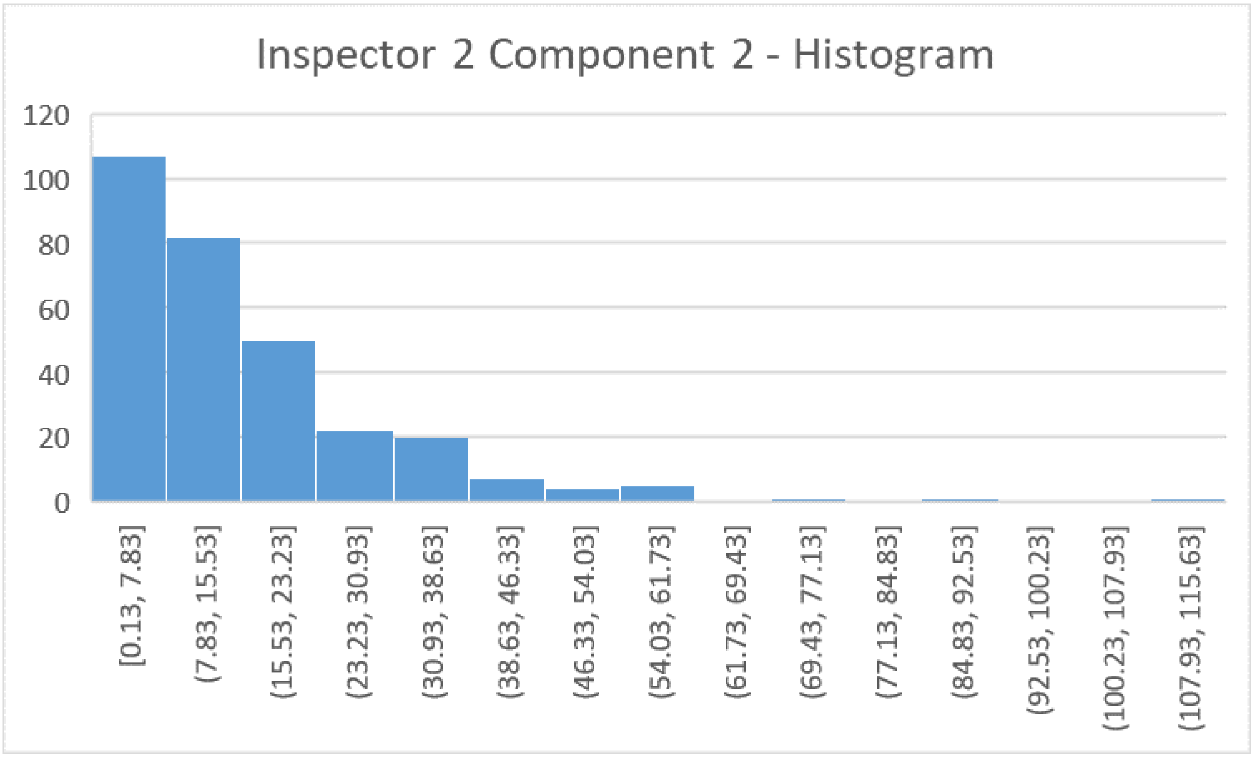
Appendix A (Analysis of Data Files for Simulation Entities)

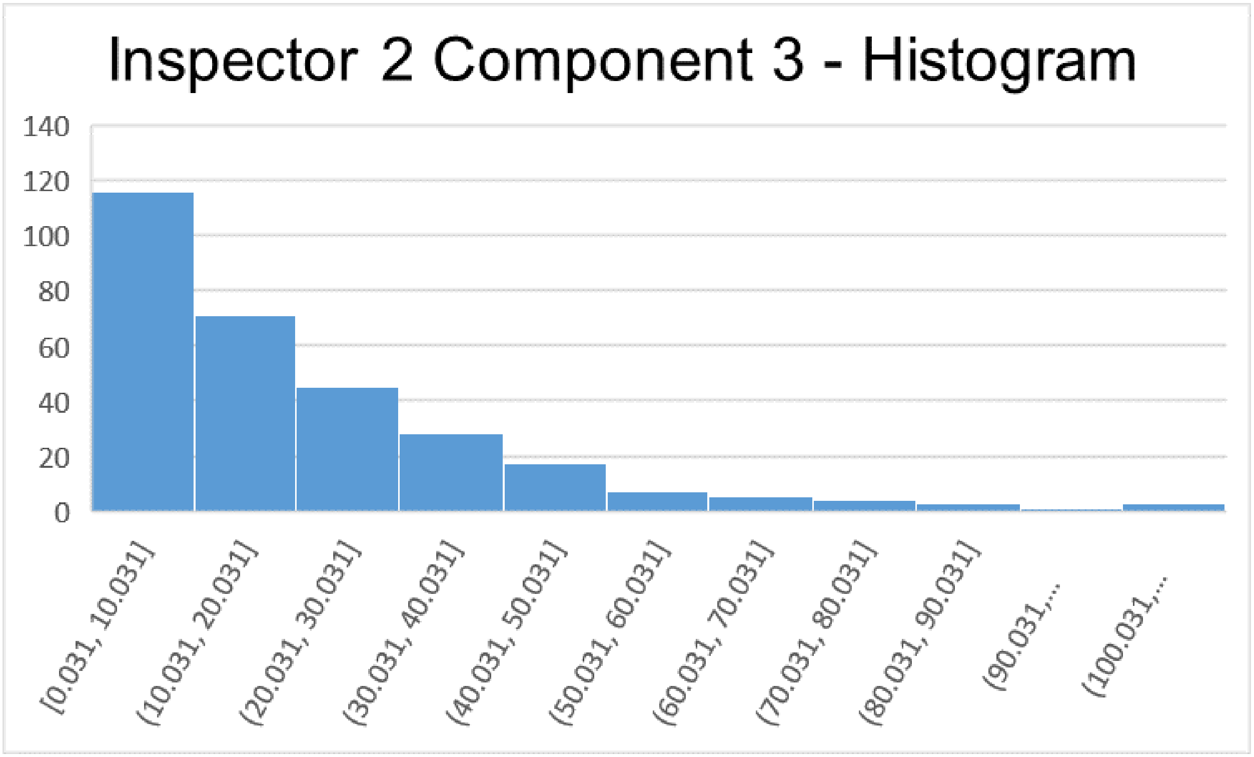
The following information corresponds to the histogram and Q-Q plots from the sample data for each simulation entity as part of the input modelling process.

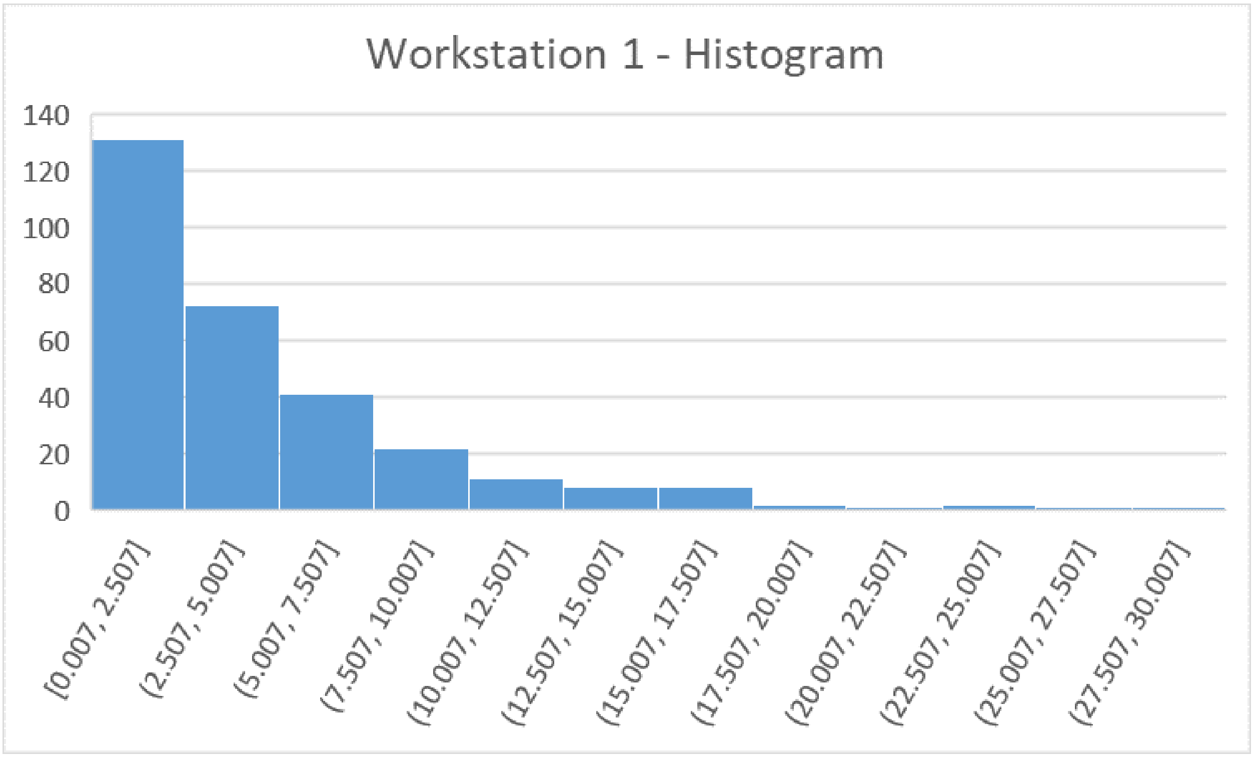
Histogram Figures

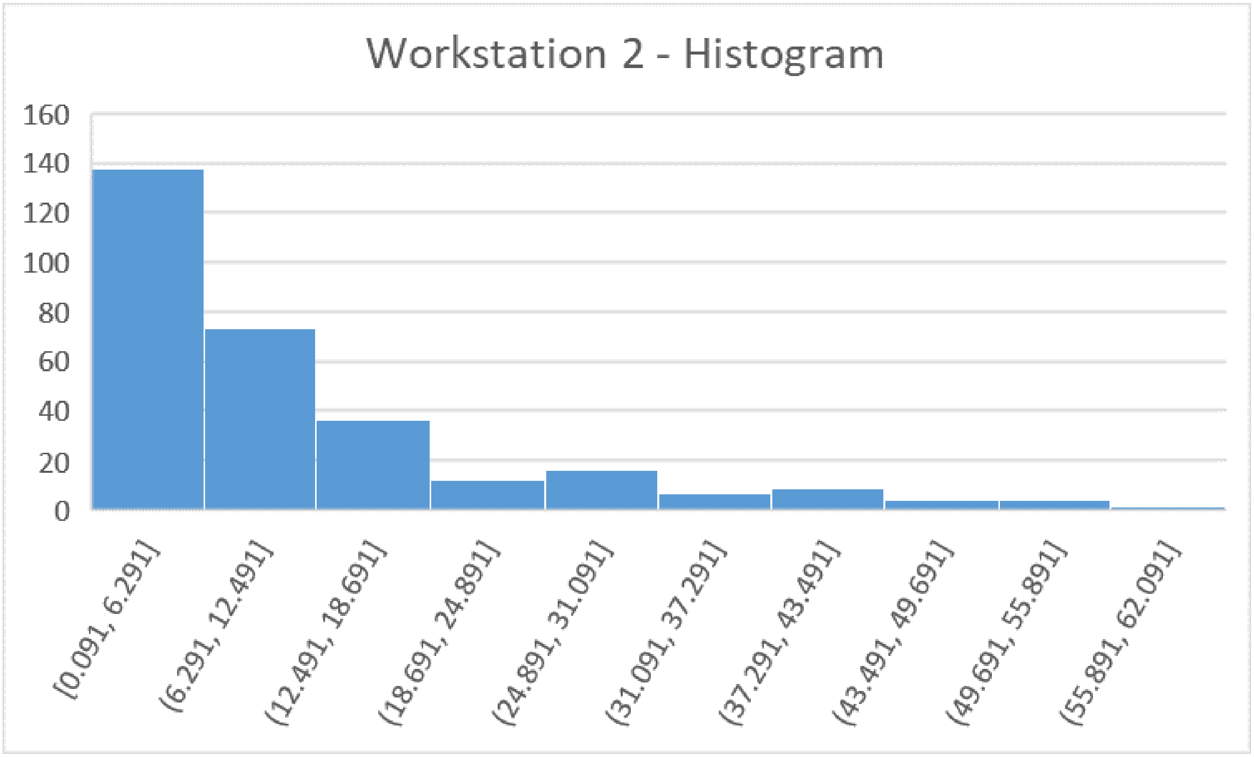
As shown in the histograms below, the distributions displayed follow an exponential distribution as it starts at the peak and observes a continuous downward trend.

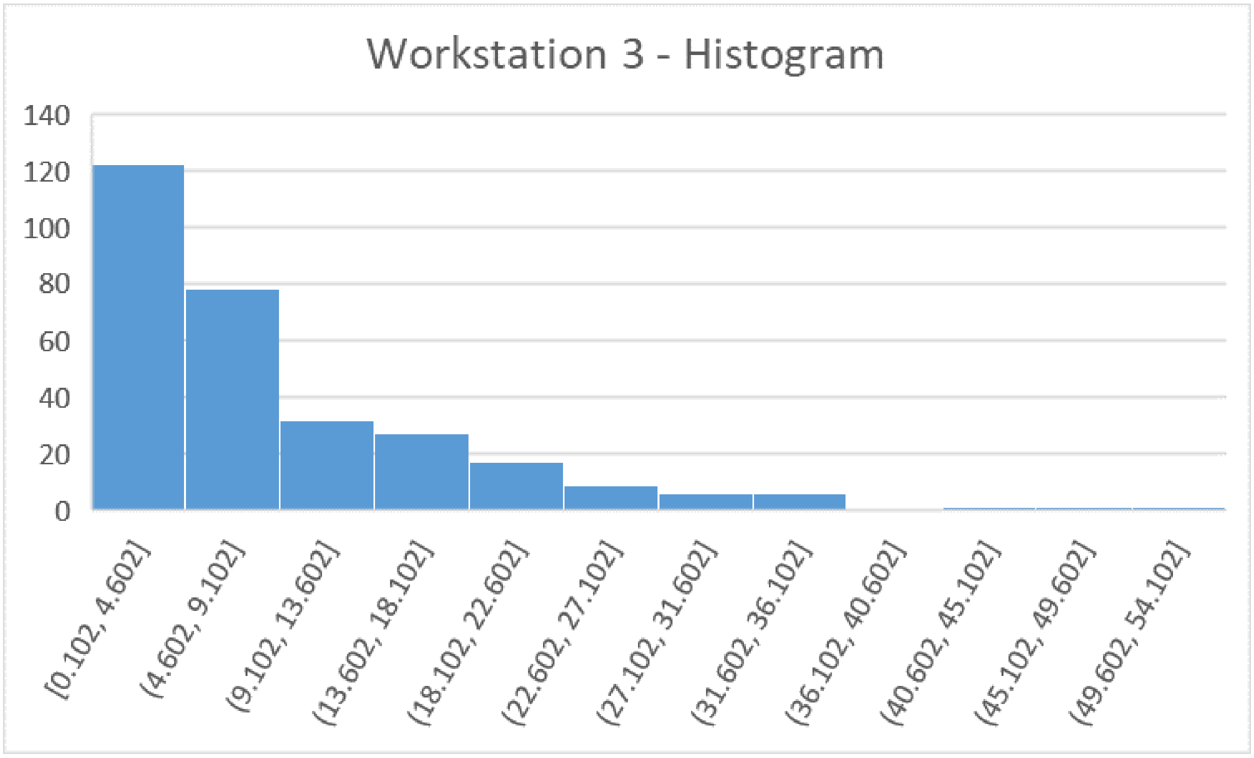



Inspector 2 Component 2 - Histogram
120
100
80
60
40
20
0
[0.13, 7.83]
(7.83, 15.53]
(15.53, 23.23]
(23.23, 30.93]
(30.93, 38.63]
(38.63, 46.33]
(46.33, 54.03]
(54.03, 61.73]
(61.73, 69.43]
(69.43, 77.13]
(77.13, 84.83]
(84.83, 92.53]
(92.53, 100.23]
(100.23, 107.93]
(107.93, 115.63]


Inspector 2 Component 3 - Histogram
140
120
100
80
60
40
20
0
[0.031, 10.031]
(10.031, 20.031]
(20.031, 30.031]
(30.031, 40.031]
(40.031, 50.031]
(50.031, 60.031]
(60.031, 70.031]
(70.031, 80.031]
(80.031, 90.031]
(90.031,...
(100.031,...

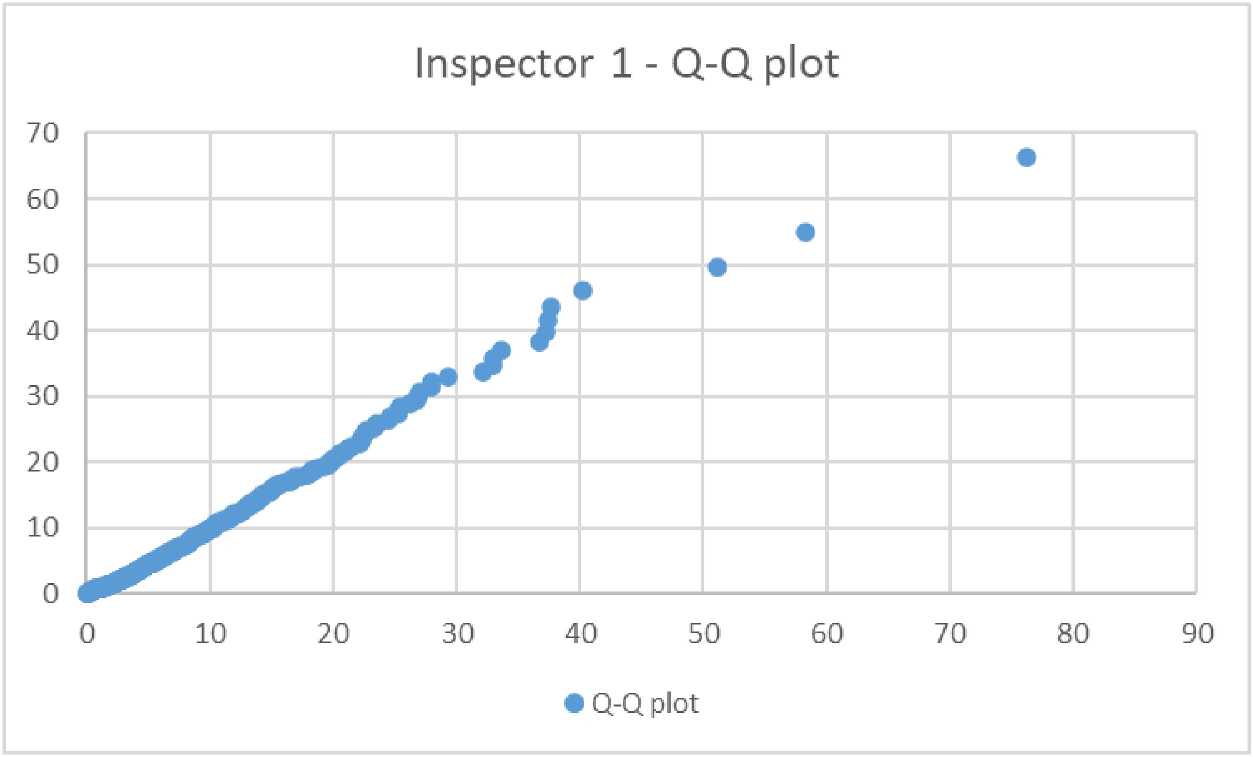

Workstation 1 - Histogram
140
120
100
80
60
40
20
0
[0.007, 2.507]
(2.507, 5.007]
(5.007, 7.507]
(7.507, 10.007]
(10.007, 12.507]
(12.507, 15.007]
(15.007, 17.507]
(17.507, 20.007]
(20.007, 22.507]
(22.507, 25.007]
(25.007, 27.507]
(27.507, 30.007]

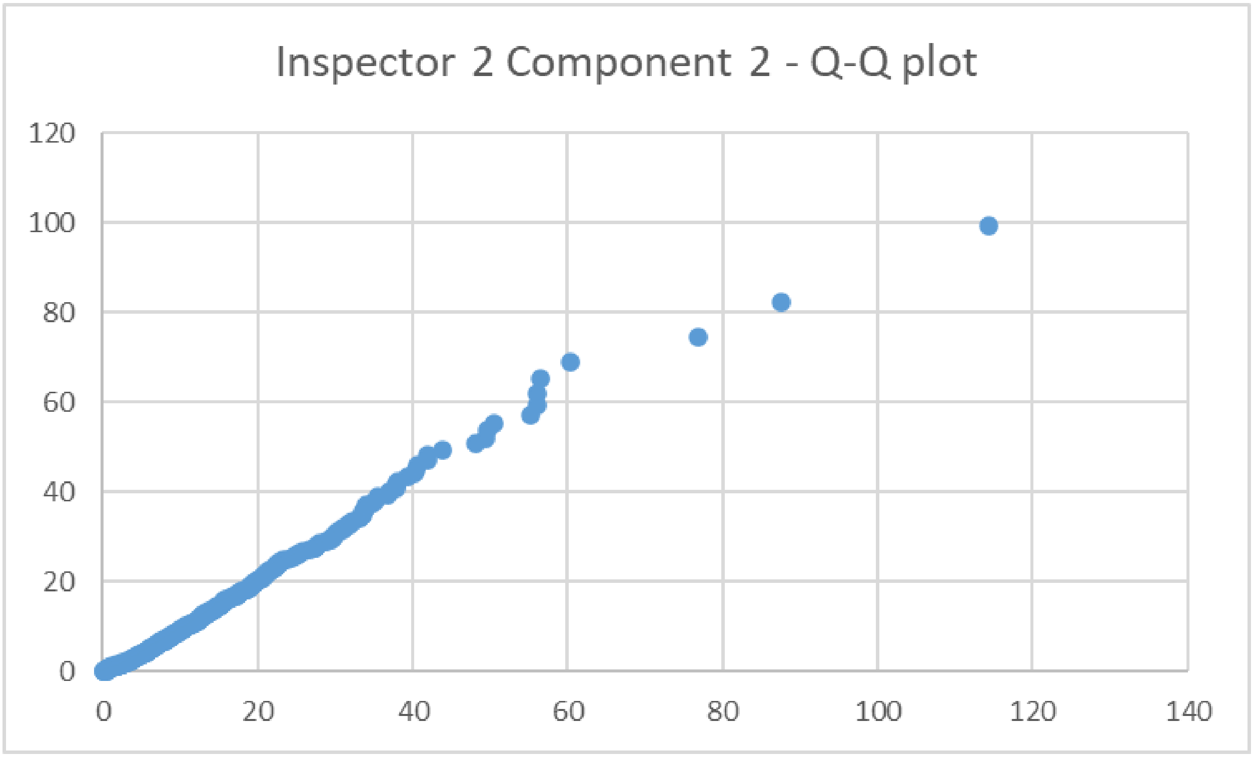

Workstation 2 - Histogram
160
140
120
100
80
60
40
20
0
[0.091, 6.291]
(6.291, 12.491]
(12.491, 18.691]
(18.691, 24.891]
(24.891, 31.091]
(31.091, 37.291]
(37.291, 43.491]
(43.491, 49.691]
(49.691, 55.891]
(55.891, 62.091]

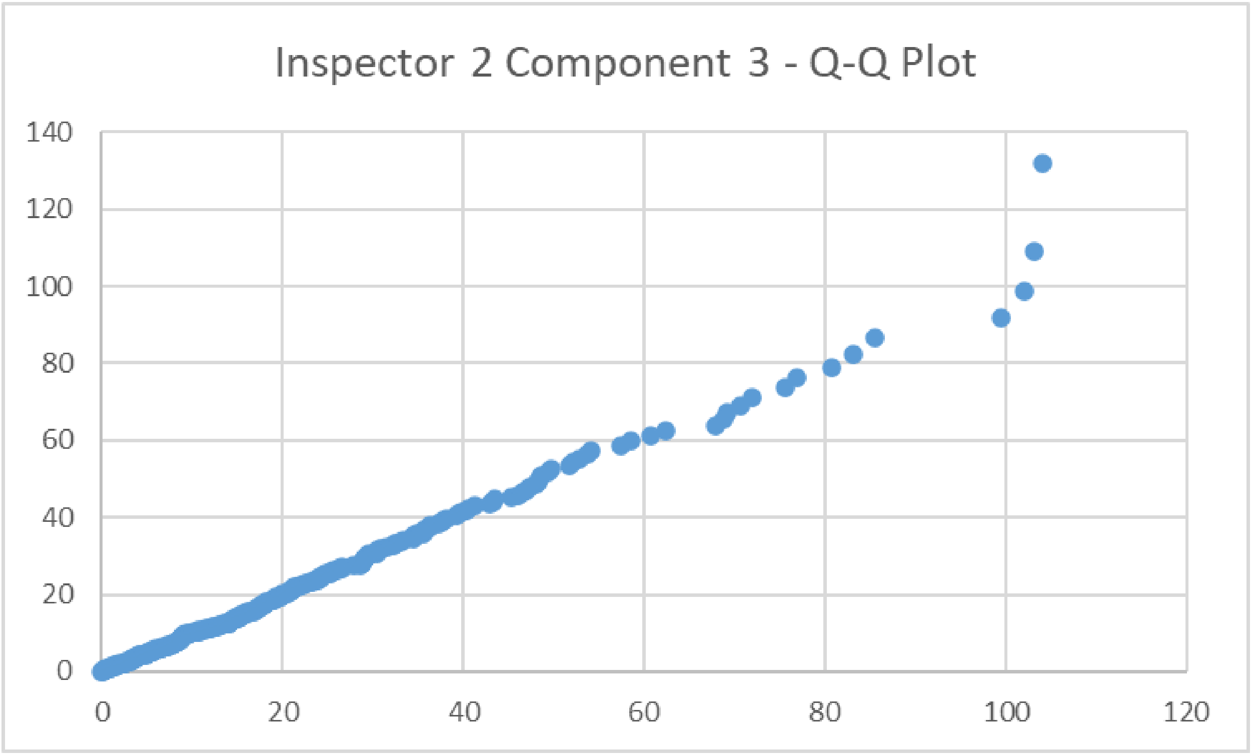


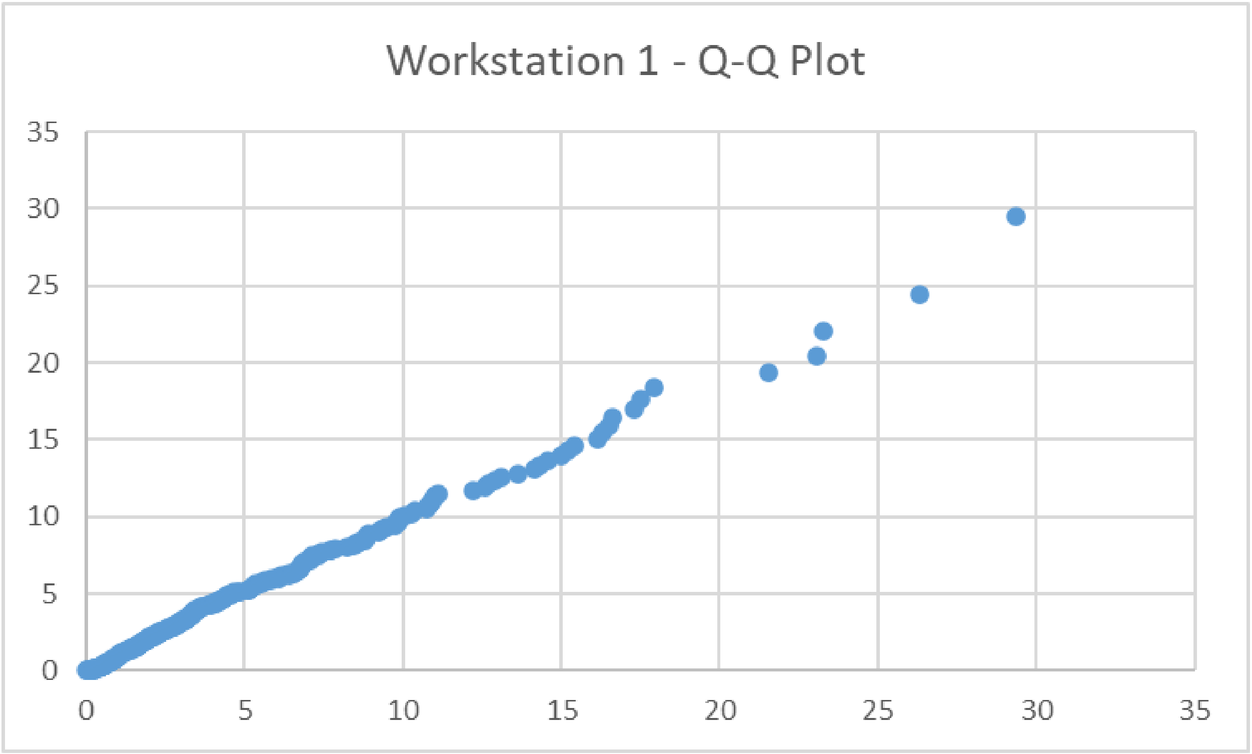
Exponential Q-Q Plots

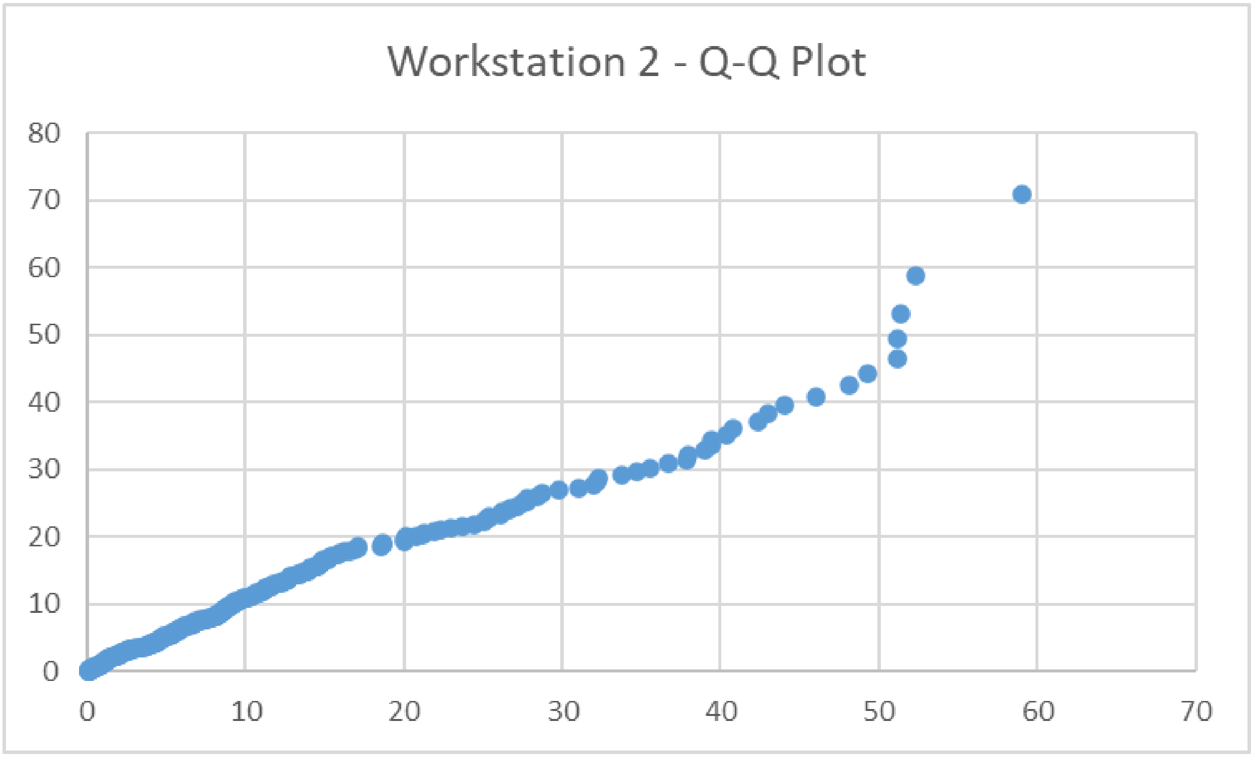
QQ and Chi-squared testing will be conducted using an exponential distribution to test the fit. If the exponential distribution is unsuccessful, a different distribution will be attempted. Then the inverse transform method of random variate generation will be conducted on the distribution. Below is a QQ plot of the sample data compared to an exponential distribution. This QQ plot is a straight line with a slight tail showing that the correct distribution was selected. The figures below represent the exponential Q-Q plot of the system entity's sample data.

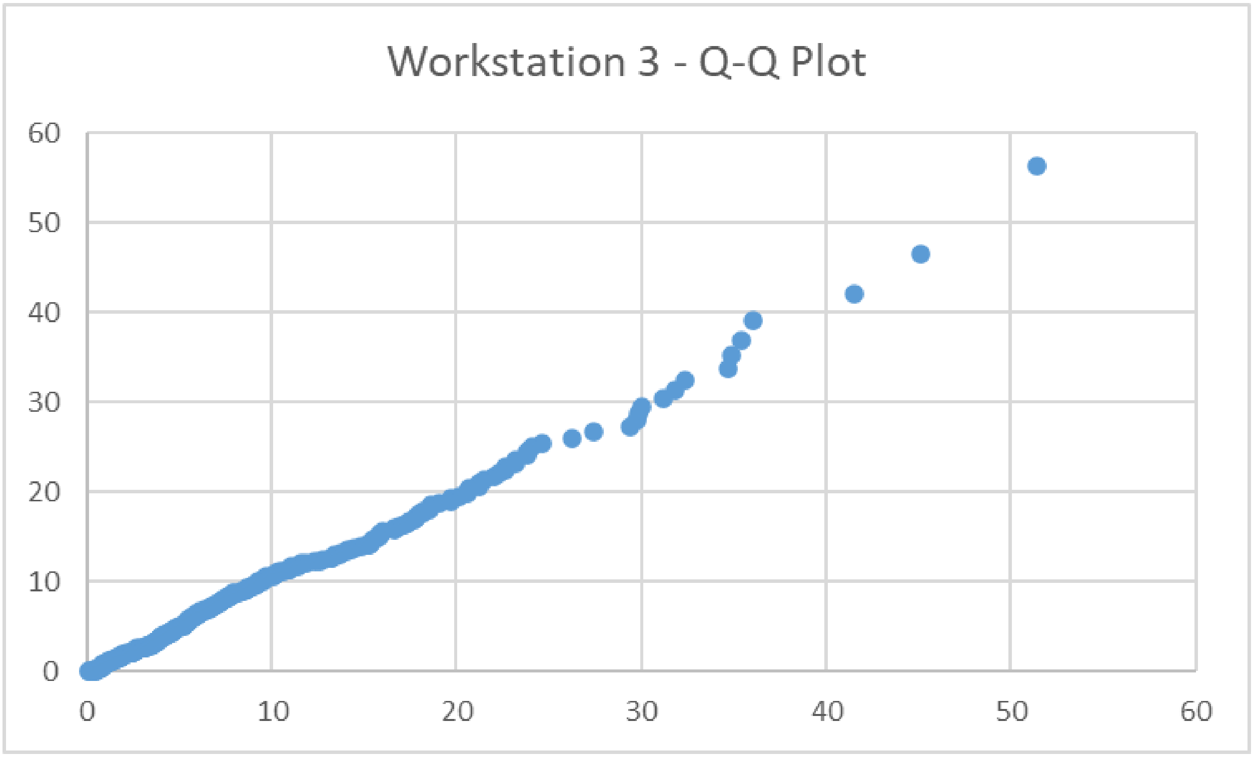

Inspector 1 - Q-Q plot
70
60
50
40
30
20
10
0
0
10
20
30
40
50
60
70
80
90
● Q-Q plot


Inspector 2 Component 2 - Q-Q plot
120
100
80
60
40
20
0
0
20
40
60
80
100
120
140


Inspector 2 Component 3 - Q-Q Plot
140
120
100
80
60
40
20
0
0
20
40
60
80
100
120


Workstation 1 - Q-Q Plot
35
30
25
20
15
10
5
0
0
5
10
15
20
25
30
35


Workstation 2 - Q-Q Plot
80
70
60
50
40
30
20
10
0
0
10
20
30
40
50
60
70


Workstation 3 - Q-Q Plot
60
50
40
30
20
10
0
0
10
20
30
40
50
60