Simulation Group Project

SYSC 4005 - Discrete Simulation / Modeling
Final Report Submission

Authors
Connor Judd, 101109256
Marko Majkic, 101109409
Millan Wang, 101114457

Due : 23h00 April 12th 2022

1 - Problem Formulation

The problem to be addressed by this simulation describes a manufacturing company attempting to find performance metrics for a product manufacturing system with hopes to find data that will inspire methodologies that maximize production rates and minimize asset downtime. The manufacturing process creates 3 products which shall be referred to as P1, P2, and P3. The products are created on corresponding workstations referred to as W1, W2, W3 with components referred to as C1, C2, and C2.

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, that 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 choose to select either C2 or C3 for inspection without any consideration for 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, 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 with the freest space 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.

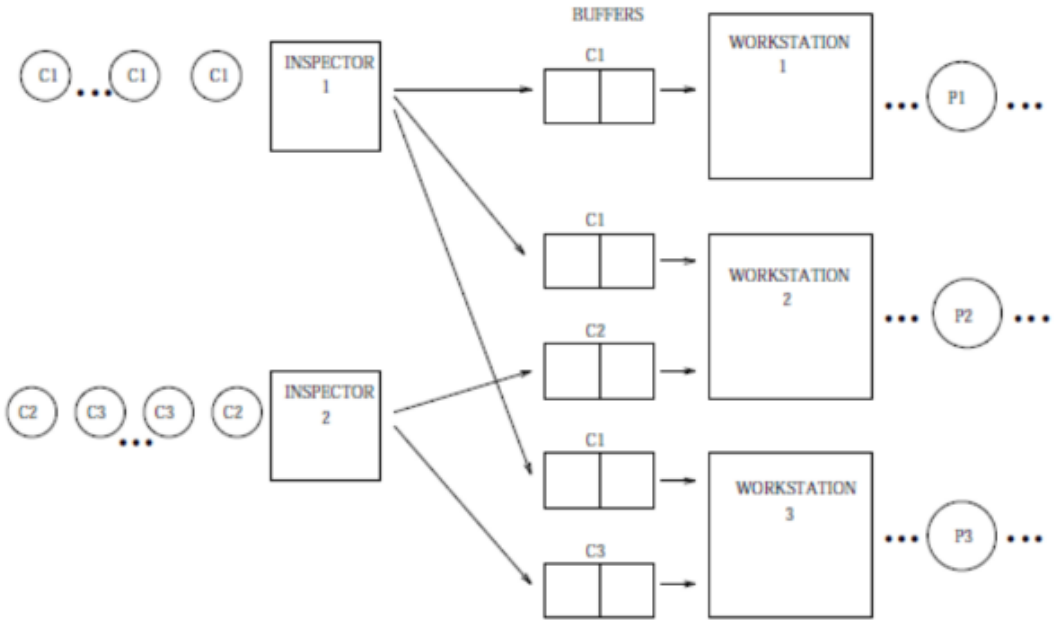


Figure 1: Schematic illustration of the manufacturing facility.

Inspectors start inspections without considering the buffers, and if an inspection completes when all buffers are full, the inspector will be blocked and wait until the first available buffer opening appears. The inspector is required to inspect components in the order that they appear in the queue, and cannot alter the order.

The company is interested in determining the throughput of products created over time, the proportion of time that each workstation is assembling compared to time spent waiting for buffers to populate, the average buffer occupancy for each buffer, and the time each inspector spends waiting for buffers to create space. The company would also like to evaluate the priority policy for inspector 1 when routing C1 to the three workstations, to maximize throughput, minimize the idle time of the inspector, and to evaluate incorporating a system for inspector I2 to determine which component to inspect next.

2 - Setting of Objectives

The main question that this simulation will address is to determine what would be the optimal way to arrange the inspectors to maximize the rate of product creation and minimize the rate that the inspectors are blocked. A simulation is the best way to address this question as it would be very difficult to find an analytical solution to this problem and it would be easier, and thus less resource-intensive, to approximate a solution with simulation.

The alternative systems to explore during the simulation would be to assess the impact of changing the methodology in which the inspector I2 selects which component to inspect next as well as which workstation buffers their inspected products will be sent to.

The completion of this project will involve the collaboration of 3 group members Connor Judd, Marko Majkic, and Millan Wang, and is expected to require a time budget of around 2 hours per week per member. No financial resources are currently expected to be needed to complete this project. The first deliverable deadline is on February 10th, and it will be assumed that the subsequent three deliverable deadlines will be approximately biweekly across the remaining 6.5 weeks of the 2nd half of the term (February 28th - April 12th). These deliverable deadlines are subject to change at the professor's discretion. Below is the plan on the deliverables for each deliverable deadline, and more detailed plans for deliverables 2-4 will be updated when they become a work in progress.

Deliverable 1 - February 10th 2022
- -Problem formulation
- -Setting of objectives & overall project plan
- -Model conceptualization
- -Model translation

Deliverable 2 - March 11th 2022
- -Data Collection & Input modeling
- -Detailed description of generating inputs based on identified model

Deliverable 3 - March 28th 2022
- -Model verification & validation
- -Production runs and statistical analysis

Deliverable 4 - April 12th 2022
- -Discussion and implementation of alternative operating policies
- -Conclusion
- -Final report
- -Complete simulation source code

3 - Model Conceptualization

This product manufacturing process will be modeled using simulation software. Inspectors, components, buffers, workstations, and products will not be real and will instead be represented in a python simulation. The simulation will use object-oriented programming to model these different entities.

To simulate the process of component inspection and product assembly, delay time generators will be used and the generated delay values will emerge from process-specific time distributions based on historical data. Each component and workstation has a corresponding .dat file containing historical timing data, and these files will be used to create the distributions needed to generate the delay values.

For inspector I1, only C1 components will be inspected, and upon completion, a process for selecting the workstation buffer to send the component to will be run. This process will try to select the buffer with the most empty space. If multiple buffers are tied for having the most empty space, then the workstation with the lowest workstation number will be selected. If all buffers are full, I1 will wait and then send the component to the first buffer that opens a buffer spot. After sending an inspected component to a buffer, I1 will start to inspect another C1 component.

For inspector I2, the random selection of components C2 & C3 for inspection will be modeled using random number generation. Since I2 does not currently have any choice on which component to inspect, I2 will also have no choice on which workstation to attempt to send the inspected component to. If the buffer corresponding to the component that I2 just inspected is full, I2 will wait until that buffer opens a spot before refilling the buffer with the held component and then randomly selecting a C2||C3 to inspect again.

Workstations will begin assembly when all of the workstation buffers have at least one component. Starting assembly will begin by removing components from the buffers and then generating an assembly time for completion. Once the completion time is reached, the assembly process will wait until all the buffers are populated again before starting another assembly. This wait time may be zero if the buffers are already populated.

All of 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 - Data Collection

Data collection has already been done. The historical data records are in the given .dat files and represent a historical record of the time taken to inspect components, and to build final products.

5 - Model Translation

Python was chosen as a language of simulation as all members of the group are familiar with it and it was determined that the time investment required to become fluent with dedicated simulation languages would not justify the potential increases in performance given the current time budgets. It was also determined that increasing the group members' fluency with Python's data analysis libraries would be more professionally marketable than learning dedicated simulation languages.

A UML class diagram representing the below entities and new additions in later iterations can be found alongside this report document in the source code ZIP

SimulationLogger.py

The **`SimulationLogger`** class will serve to allow progress messages to be logged at different stages of the simulation. The simulation logger includes three Boolean flag variables that are set to either true or false and allow the object to know if logging is enabled for each respective aspect of the simulation. The first flag is the **`__enable_inspector_logging`** flag that indicates if logging for the inspector actions is enabled. The second flag is the **`__enable_workstation_`**logging flag that indicates if logging for the workstation actions is enabled. Finally, there is the **`__create_log_files`** flag that indicates whether the simulation updates should be logged to a file (the use for this will be further explained below).

The **`SimulationLogger`** includes 4 functions and a constructor. The constructor takes as parameters the Boolean values for **`enable_inspector_logging`**, **`enable_workstation_logging`**, and **`create_log_files`** and initializes the class's flags with those values. The first function **`log_inspector_component_selection`** takes a component and time as parameters, and logs at which time this component has started selection. The second function, **`log_inspector_buffered_component`**, takes a component, workstation, and time as parameters and logs at which time the component has been put in the workstation's buffer. The third function, **`log_workstation_unbuffer`**, takes a workstation,and time as parameters, and logs at which time the workstation starts to build its assigned product. Finally, the **`log_product_created`** function takes a product and time as parameters and logs at which time the product has been produced.

Each function first checks that its respective enable flag is not set to false, and then constructs the string and prints it. If the flag is set to false, it will do nothing. In the future, the logging will be implemented using the python Logging library which will allow messages to be easily logged to a file in an organized manner, instead of to the console. The **`__create_log_files`** flag, while currently unused, will serve as a feature flag that will toggle the logging to file feature on and off.

SimulationEnums.py

The **`SimulationEnums`** file contains three classes that implement enumerations. The first class is the **`Product`** enumeration which contains the three possible products P1, P2, P3, and their respective ID's. The second class is the **`Component`** enumeration which contains the three possible components C1, C2, and

C3, and their respective ID's. The third class is the **Event_Types** enumeration which includes all the four possible events that occur in the simulation which are **Inspection_Complete** which is when an inspector completes their inspection, **Add_to_Buffer** which is when the inspector adds a component to a buffer, **Unbufffer_Start_Assembly** which is when a component is removed from a buffer and placed in the workstation for assembly, and **Assembly_Complete** which is when a product has been fully assembled.

Buffers.py

The **Buffer.py** file contains two classes – the **Component_Buffer** class and the **Component_Buffer_Manager** class. The **Component_Buffer** implements the buffer object itself where components are stored and waiting to be pushed into the workstation. The **Component_Buffer** has an **attempt_to_add_buffer** function which attempts to add a component to the buffer object, and returns true if successful, and false otherwise. It implements this by simply incrementing the **component_count** variable of the buffer object. The **Component_Buffer** also includes a function called **remove_from_buffer** which removes components from the buffer and throws an error if the buffer is empty. It also implements this by decrementing the **component_count** variable.

The **Component_Buffer_Manager** class acts as a controller for all the buffers. It routes components to the appropriate buffer and pushes components out of the buffer into the workstations. The **Component_Buffer_Manager** has two functions and a constructor. The constructor initializes all the **Component_Buffer** objects needed with their associated component product pairs. 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. It implements this by checking the type of component that was passed to it and then checking if it can add that component to one of the buffers that is associated with that component. The **attempt_to_assemble_product** method takes a product as a parameter and takes the needed components off the workstation buffer to assemble the product. It returns true if the needed components were provided from the buffers, false otherwise. It implements this by checking the product type and then checks if a buffer associated to that product can provide one of the components.

Inspectors.py

The **Inspector** File contains two classes, one for each inspector. Each inspector class will be responsible for calculating the delay time based on the imputed data file for the corresponding component. In **Inspector1** there is an **__init__** method to initialize the class with the mean of component 1. After this is the **__build_distribution_from_dat** method that will find the mean from the imputed data set.The code will locate the data files then work through it sumating the contents. It uses the sum to calculate and return the mean. 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** is very similar to **Inspector1.** **Inspector2** also has the **__init__** file to initialize but in **Inspector2** it sets the mean for C2 and C3. The following method **__generate_comp2or3** is used to randomly select between C2 and C3 for the next part to inspect. This uses random.getbits to create the random input. **__build_distribution_from_dat** is the next method and will find the mean from the imputed data set and component. The final method is **generate_inspect_time** and this will output the time delay depending on what component is sent in.

MainSim.py

The **MainSim.py** file is the main file that contains the **Simulation** class and the main script that facilitates running the simulation. The simulation is run using a single future event list and will continue to address and schedule events until the product creation limit is reached. The constructor method **__init__** for the **Simulation** class takes a logger object and sets up the needed properties to run the simulation. The **get_next_chronological_event** method identifies the closest chronological event in the future event list and returns & removes it to continue the simulation. This is implemented by iterating through the future event list and then comparing each event start time with the previously tracked minimum. The **schedule_add_to_buffer** method is given an inspector number, and adds an event to the future event list for when the inspection of the current component will complete. The calculation of this inspection time is done with the corresponding inspectors inspection time generation mechanism. The **process_add_to_buffer** method processes an attempt to add an inspected component to a buffer. If no buffers are available to receive this component nothing happens. 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. The **process_workstation_unbuffer** method deals with the previously mentioned workstation construction initiation events and 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 **process_product_made** method deals with the arrival of the completed assembly events and increases the simulation's product counters accordingly. This then adds an event to the future event list to make the workstation indicating that it is to start constructing a new product.

Workstation.py

The **Workstation.py** file contains a single class **Workstation** whose purpose is to receive components and calculate 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 as well as the product. The Next method is **__build_distrubution_from_dat** which will see what product is being worked on and select the corresponding datafile and calculate the mean to be used for random number generation. The final method is **get_unbuffer_time** and this method is used to generate a product assembly delay time from the mean.

Input Data Modeling

To proceed with this project, a random number and variate generation process for the individual component inspection and workstation build times must be determined. This will be done by analyzing the input data and then creating a distribution model for generation of new data points. This will be done with the following steps:

1. Generate a histogram for the simulation entity from the associated dat file
2. Compare the histogram to default distribution graphs.
   See which is the most similar for next step choice
3. Choose a distribution to test if the input data fits to the default graphs.
   QQ and Chi-squared testing
   If the tests conclude that the input data does not match the
   hypothesized distribution, try a different distribution
4. Select a method for random variate generation based on the distribution

The above process will be repeated for all 6 .dat files representing the different service times for the component inspection and workstation build processes.

Stat Manager

The `Stat_Manager` class provides functionality to select the data files to then make comparisons to different distributions. The method reads data from the selected files, makes calculations, and sends the required information to method calls in the `Grapher` class. When a `Stat_Manager` object is instantiated, it calls the `compute_stats` method and will save the data for the selected data set. The `compute_stats` method takes in an input to select which data file is going to be analyzed. The data file is then located, read line by line, and then converted into a Python list of floats. The input data has a precision of 3 decimals, and to prevent issues with float numbers, all values in the list are converted to integers by multiplying by 1000 to represent milliminutes instead of minutes. Using Python's built in list operations and statistics library, the sum, count, sample mean, and sample variance are determined, and a sorted copy of the list is also stored.

The next set of methods are used for doing operations on the data for visualization and hypothesis testing. The first is **generate_histogram**, which interfaces with the grapher class to generate histograms. The next method is **qq_plot** which also interfaces with the grapher for QQ Plot generation. The next methods are in the **chi_squared** family and conduct chi squared tests on the sample data to test if the sample data is consistent with a hypothesized distribution. The subsequent methods are used as helper methods to the above procedures to generate quantiles and cdf values. The above functionality inside `Stat_Manager` will be used for distribution fitting for the different simulation entities.

Graphing Process

The **Grapher** class serves as a helper for the **stat_manager** class. The **Grapher** class provides the ability to create a multitude of graphs by providing the functions with the proper parameters. There are 7 different graphs that can be graphed using the grapher: Histogram, Q-Q plot, Poisson Distribution Plot, Normal Distribution Plot, Exponential Distribution Plot, Weibull Distribution Plot, and a Log Normal Distribution Plot.
The functions and their specifications are as follow:

The **build_histogram** function allows the user to provide an array of data points, and a title. It will then calculate the appropriate number of bins using the data and the Freedman–Diaconis formula, create the histogram using Matplotlib's histogram function, then save it to a file with the title and a date/time stamp.

The **build_qq_plot** function allows the user to provide two lists of data points and a title. It will then create the q-q plot by plotting the lists on the x and y-axis respectively, then save it to a file with the title and a date/time stamp.

The **build_poisson** function allows the user to provide an array of lambdas (1 or more) and a title. It will then use Scipy's poisson library to create the poisson PMF and plot the function using Matplotlib. It will do this for each lambda in the array and plot them on the same figure, then save it to a file with the title and a date/time stamp.

The **build_normal** function allows the user to provide a mean, a standard deviation, and a title. It will then calculate the distribution's range, and using Scipy's Norm library, calculate the PDF and plot the function using Matplotlib, then save it to a file with the title and a date/time stamp.

The **build_exponential** function allows the user to provide an array of betas (1 or more), and a title. It will then use the Seaborn library's kdeplot function to plot an exponential relationship using the provided beta. It will do this for each beta in the array and plot them on the same figure, then save it to a file with the title and a date/time stamp.

The **build_weibull_constant_beta** function allows the user to provide an array of alphas (1 or more), and a title. It will then use Reliability statistics library's **Weibull_distribtuion** function to create a distribution and a corresponding PDF and plot it using Matplotlib. It will do this for each alpha in the array and plot them on the same figure, then save it to a file with the title and a date/time stamp.

The build **build_weibull_double_param** function, similarly to the **build_weibull_constant_beta** will create a Weibull plot in the same way, however, -**build_weibull_double_param** allows the user to provide an alpha and beta value, and it only can create one plot.

The **`build_log_normal`** function allows the user to provide a mean, standard deviation, and title. It will then create a log-normal distribution using Numpy, and plot the histogram, with the overlaid PDF, then save it to a file with the title and a date/time stamp.

Input Data Modeling - Distribution Fit Hypothesis Testing

To test if a collection of input data fits a hypothesized distribution, QQ tests and Chi-Squared tests will be conducted. For the simulation entities in this project, the distribution fit hypothesis testing will occur in the hypothesis.

QQ Testing

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 set of data. In the code we currently have implemented QQ testing for Exponential and Weibull distributions because of the initial observations of the histograms. For a Weibull distribution the following equation is used to find the inverse.

$$Q(p;k,\lambda) = \lambda(-\ln(1-p))^{1/k}$$

for $0 \leq p < 1$.

In this function, p is varied in the range (0,1) and uses variables k and lambda to control the fit. For the exponential distribution the following equation is used.

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

In this function p is varied in the range (0,1) using a variable value for lambda to control the fit. By changing the constants in the equation, the best possible fit for the distributions can be found. The fit of the distribution is evaluated by how close data points are to a straight line on the graph. The closer that this 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 before the verification can be found in the appendix, however the post-verification updates plots can be found in section 6

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, then the next step is to tabulate the observed frequencies in each bin in the same way that it is done for making histograms. Then for each bin, an expected frequency number will need to be determined. This expected frequency number will be calculated by

determining the difference between the cdf's 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 result of…

((observedFrequency-expectedFrequency)^2)/expectedFrequency

…will be calculated and then summed into a final variable called Chi-Squared. This Chi-Squared value will be compared to a reference Chi-Squared value from a table whose value will be dependent 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 is rejected and the distribution is not the same as the sample data, and if the table value is larger than the calculated value, the null hypothesis will not be rejected and we can proceed with the assumption that the selected distribution is the same as the input data. A more detailed run through of this process with sample data can be found in section 6.

Random Number Generation Process

To generate randomized variates to use for the simulation's component inspection times and workstation build times, a random number generation system was manually implemented and tested. After determining what distribution best models an input dataset, random numbers will be generated following the below steps

1. Seeded pseudo random generation with multiplicative congruential model
2. Use 2 multiplicative congruential models with L'Ecuyer's algorithm to get a better pseudo random number with a much longer generator period
3. Use the L'Ecuyer's generated number to generate a value in the specified distribution with inverse-transformation or the acceptance-rejection technique

Multiplicative Congruential Model

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

$$X_{current} = (aX_{previous}) \% m$$

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 previous generated number, the user must define an initial seed value to set as the first Xprevious so that subsequent numbers can be generated. Once a number is generated, the value for Xprevious will be overwritten by the current Xcurrent value.

L'Ecuyer's Algorithm

To increase the period of the random number generation to be sufficiently large to avoid a period cycle, two multiplicative congruential model instances will be combined to make a generator with a period of approximately 2x10^18. This will be done using a slightly modified version of L'Ecuyer's algorithm as shown below.

$SubGenerator\ 1: m = 2147483563, a = 40014, seed = first\ generated\ int\ from\ SubGenerator2$
$SubGenerator\ 2: m = 2147483399, a = 40692, seed = user\ defined\ in\ range[1, 2147483398]$

$$L'Ecuyer's\ Generator: X_{Current} = (X_{SubGen1,\ current} - X_{SubGen2,\ current})\ \%\ 2147483562$$

Xcurrent represents the current random integer generated from the L'Ecuyer process and the sub-generator specific Xcurrent values will come from their individual multiplicative congruential model generators. To convert the random integer to a decimal number in the range [0,1], denoted as R, the below translation will occur.

$$R_{current} = if\ (X_{current} > 0)\ \{\frac{X_{Current}}{2147483563}\}\ else\ \{\frac{2147483562}{2147483563}\}$$

Random Number Generation Testing

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

Kolmogorov-Smirnov Testing

To determine if the random number generation process produces uniform values, a Kolmogorov-Smirnov test will be conducted comparing a set of randomly generated numbers to a uniform distribution. The null hypothesis of this test states that there is no difference between the distribution of the random numbers and a uniform distribution, and the alternate hypothesis states that they are different.

This test will compare the continuous cumulative distribution function of a uniform distribution to the empirical cumulative distribution function observed from a set of random samples.

The first step is to generate a set of random numbers. 100 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 100 numbers will be created to represent the expected cumulative distribution function values, also ordered from smallest to largest. Given that there are 100 random samples to match, this list will also have 100 values, starting at 0.01 with each element being 0.01 larger than the previous as shown below

```
[0.01, 0.02, 0.03, . . . 0.98, 0.99, 1.0]
```

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 of the list will be the calculation result of the (currentExpectedValue) - (currentSortedRandomNumber). For example the first element of the D_plus list will be the result of

(0.01)-(smallestRandomNumber)

The second list will be called D_minus and each element of the list will be the calculation result of (currentRandomNumber)-(previousExpectedValue). The first element of D-minus will use previousExpectedValue=0, and the second element in D_minus will be

(2ndSmallestRandomNumber)-(0.01=previousExpectedValue)

Once the D_plus and D_minus lists are made, the largest 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(100)=0.136

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.

This test is done in the code in **`Lecuyer_Generator.py`** in the **`run_kolmogorov_smirnov_test()`** function and after multiple repeated automated tests, the null hypothesis was not rejected.

Autocorrelation Testing

To determine if the random number generation process produces independent numbers, an autocorrelation test will be conducted. The null hypothesis of this test states that the numbers are independent, and the alternate hypothesis is that they are not.

The test assesses if subsequently selected random numbers tend to follow a pattern, i.e if low numbers tend to be followed by higher numbers, or if high numbers tend to be followed by high numbers, etc.. This first involves setting & calculating the below variables, and the values used for testing this project are shown below

$N = 1000 = Total\ number\ of\ random\ numbers\ to\ generate$
$\alpha = 0.05 = confidence.\ This\ value\ halved\ will\ get\ ZTable\ value\ to\ beat$
$i = 1 = starting\ index\ in\ the\ set\ of\ random\ numbers$
$m = 1 = lag,\ how\ many\ numbers\ to\ skip\ between\ each\ choice$
$M = 998 = Largest\ integer\ that\ makes\ i + (M + 1)m \leq N\ true$

Given the above variables calculate the following values

$$\rho = \frac{1}{M+1}\left(\sum_{k=0}^{M} R_{i+km} R_{1+(k+1)m}\right) - 0.25$$

$$\sigma = \frac{\sqrt{13M+7}}{12(M+1)}$$

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

If the above condition is false, then the null hypothesis is rejected and we conclude that the numbers are dependent. If the above condition is true, then we cannot reject the null hypothesis, and can proceed with the assumption that the numbers are independent with 95% confidence. This test is done in the code in **`Lecuyer_Generator.py`** in the **`run_autocorrelation_test()`** function and after many repeated automated tests, the null hypothesis was not rejected

Random Variate Generation

Using the above uniform random number generator, random variates along a specified distribution will be generated with the following methods

Inverse-Transform Technique

For generating random variates, the preferred method is the inverse-transform technique. The inverse-transform technique requires an inversible cumulative distribution function for the distribution to generate from. Once the inverse of a distribution's cumulative distribution function is determined, then a randomly selected number from a uniform distribution will become the input parameter, and the output of the calculation 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 or Weibull, this technique will be used throughout.

Acceptance-Rejection Technique

When the inverse of a distribution's cumulative distribution function is impossible or prohibitively difficult to determine, the Acceptance-Rejection technique for generating random variates will be done instead of the Inverse-Transform technique.

Before generating a random variate with the acceptance-rejection technique, one must first determine a condition in which random variates will be accepted. Every time a number is generated it will be checked against this condition and if the condition is false, the number will be discarded and re-generated. If a generated number makes the condition true, then it will be forwarded to be used in the simulation.

In this project, since all random variates come from distributions that are easily modeled with the inverse-transform technique, the acceptance-rejection technique will not be used, however, the technique will be ready in the event that there are new simulation random variates that are needed and follow a distribution best represented by the acceptance-rejection technique.

6 - Verification of Parameter Estimation

To determine which parameters were best suited for each of the dataset's distributions, a systematic process was employed. First, a distribution of each dataset was hypothesized based on the generated histograms via visual inspection. Then, functions of the hypothesized distributions were implemented with variable input parameters to calculate the x-axis and y-axis values for the corresponding quantile-quantile plot. The set of x-values were the sorted values from the entity specific .dat files, while the y-values were the result of the inverse cumulative distribution function with an input of…

(current index in sorted dataset - 0.5) / (dataset size)

The set of x and y values are generated and passed into a function that plots the qq plot, and a function calculates the coefficient of determination of the relation to assess if the current parameter for the hypothesized function leads to points that are consistent with the dataset.

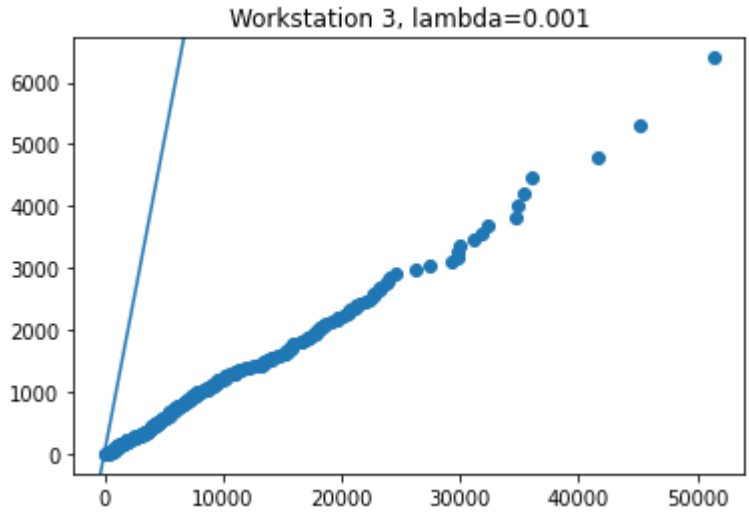
For the datasets in which a Weibull distribution is hypothesized, a system with nested for loops was implemented in order to iterate through a large number of parameter values in a short time, and quickly determine the most appropriate ones. The first loop iterates through a set of k values, while the nested for loop iterates through a set of various lambda values. For each k-value, the set of lambda values was tested. A report file for each dataset containing the parameters used and the resulting coefficient of determination values is created, as seen below in the image displaying a portion of a report file for one of the weibull distributed datasets. The report file also shows how the parameter values are systematically iterated through on an alternating basis using nested for loops.

```
k = 0.25, lambda = 9000, R^2 = 0.49031151836870335
k = 0.25, lambda = 9500, R^2 = 0.49031151836870335
k = 0.25, lambda = 10000, R^2 = 0.4903115183687035
k = 0.25, lambda = 10500, R^2 = 0.4903115183687035
k = 0.25, lambda = 11000, R^2 = 0.49031151836870335
k = 0.25, lambda = 11500, R^2 = 0.4903115183687036
k = 0.25, lambda = 12000, R^2 = 0.4903115183687035
k = 0.25, lambda = 12500, R^2 = 0.4903115183687038
k = 0.25, lambda = 13000, R^2 = 0.49031151836870335
k = 0.25, lambda = 13500, R^2 = 0.4903115183687036
k = 0.25, lambda = 14000, R^2 = 0.4903115183687035
k = 0.25, lambda = 14500, R^2 = 0.49031151836870335
k = 0.25, lambda = 15000, R^2 = 0.4903115183687036
k = 0.25, lambda = 15500, R^2 = 0.4903115183687035
k = 0.5, lambda = 9000, R^2 = 0.8647164109201921
k = 0.5, lambda = 9500, R^2 = 0.8647164109201924
k = 0.5, lambda = 10000, R^2 = 0.8647164109201924
k = 0.5, lambda = 10500, R^2 = 0.8647164109201921
k = 0.5, lambda = 11000, R^2 = 0.8647164109201918
k = 0.5, lambda = 11500, R^2 = 0.8647164109201926
k = 0.5, lambda = 12000, R^2 = 0.8647164109201928
k = 0.5, lambda = 12500, R^2 = 0.8647164109201921
k = 0.5, lambda = 13000, R^2 = 0.8647164109201919
k = 0.5, lambda = 13500, R^2 = 0.8647164109201921
k = 0.5, lambda = 14000, R^2 = 0.8647164109201919
k = 0.5, lambda = 14500, R^2 = 0.8647164109201924
k = 0.5, lambda = 15000, R^2 = 0.8647164109201919
k = 0.5, lambda = 15500, R^2 = 0.8647164109201921
k = 0.75, lambda = 9000, R^2 = 0.9761914830544044
k = 0.75, lambda = 9500, R^2 = 0.9761914830544033
k = 0.75, lambda = 10000, R^2 = 0.9761914830544042
k = 0.75, lambda = 10500, R^2 = 0.9761914830544035
k = 0.75, lambda = 11000, R^2 = 0.976191483054404
k = 0.75, lambda = 11500, R^2 = 0.9761914830544037
k = 0.75, lambda = 12000, R^2 = 0.9761914830544044
k = 0.75, lambda = 12500, R^2 = 0.9761914830544044
k = 0.75, lambda = 13000, R^2 = 0.976191483054404
k = 0.75, lambda = 13500, R^2 = 0.9761914830544044
k = 0.75, lambda = 14000, R^2 = 0.9761914830544042
k = 0.75, lambda = 14500, R^2 = 0.9761914830544042
k = 0.75, lambda = 15000, R^2 = 0.9761914830544037
k = 0.75, lambda = 15500, R^2 = 0.9761914830544044
```

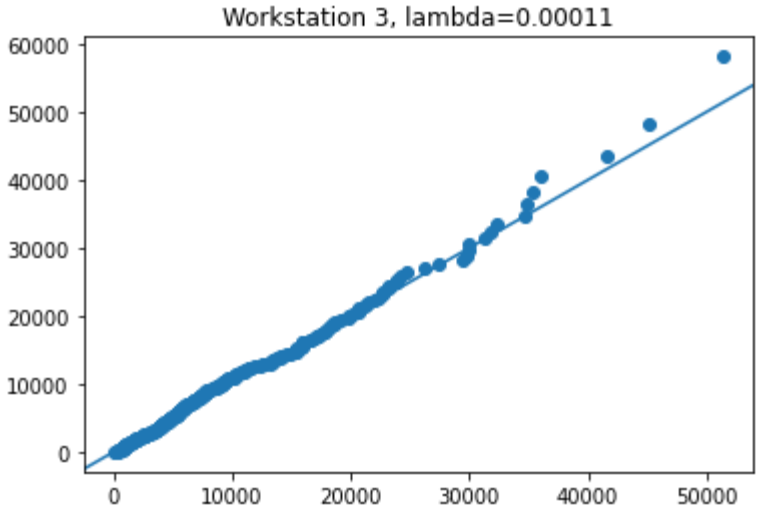
Upon inspection of this report file, the parameters that yield the highest coefficient of determination are found, and the corresponding qq-plots are also inspected to ensure those parameters did indeed provide the most linear relation that most closely followed the 45-degree reference axis.

For datasets demonstrating an Exponential distribution, a more simple approach was taken as there was only one parameter to vary - the k-value. A single for-loop that iterates through a set of lambda values was used to generate the plots and coefficients of determination, and a similar observational approach as was taken for the Weibull datasets was followed.

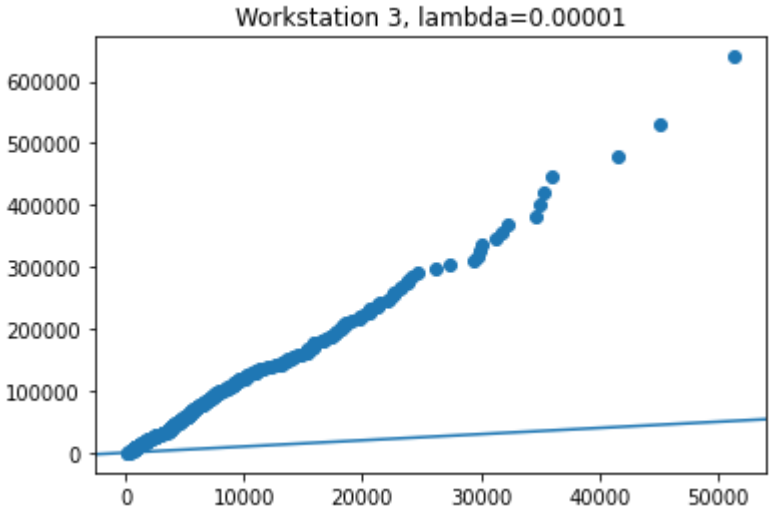
As can be seen by the example plots below, with a varying lambda value, the relation's alignment with the 45 degree reference axis changes. The process of iterating through parameter values with miniscule steps at a large scale allowed to narrow down an accurate parameter estimate.



In the case above, a lambda value that's slightly too large led the relation to be found below the reference axis.



In the case above, it is evident that an ideal lambda value was found due to the relation following the reference axis very closely.

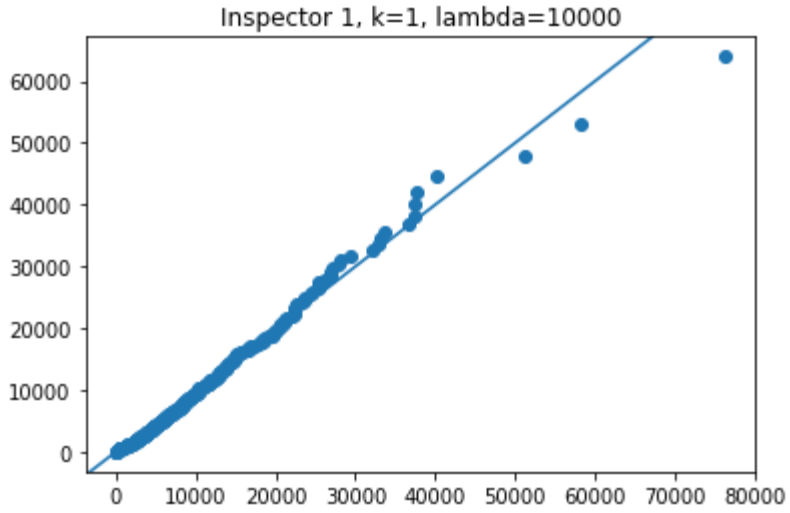


In the case above, a lambda value that's slightly too small led the relation to be found above the reference axis.

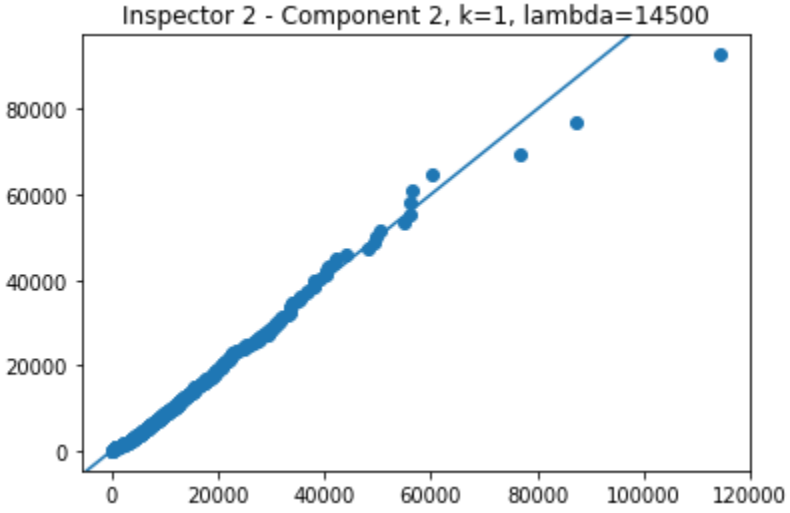
Parameter Estimation Results

After having ran through the parameter estimation algorithm outlined above for each of the six entities in the system the following parameters were found to be ideal:

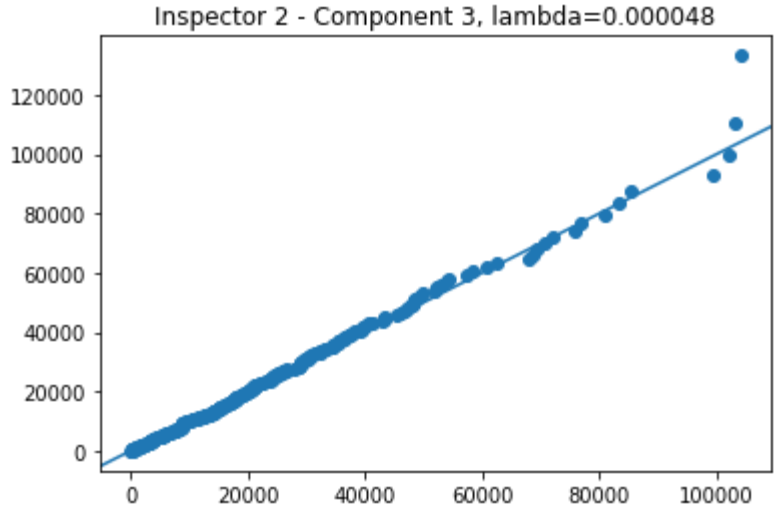
For inspector 1 which demonstrated a Weibull distribution, a k-value of 1 and a lambda of 10000 yielded a relation that was most linear and conforming to the 45-degree reference axis.



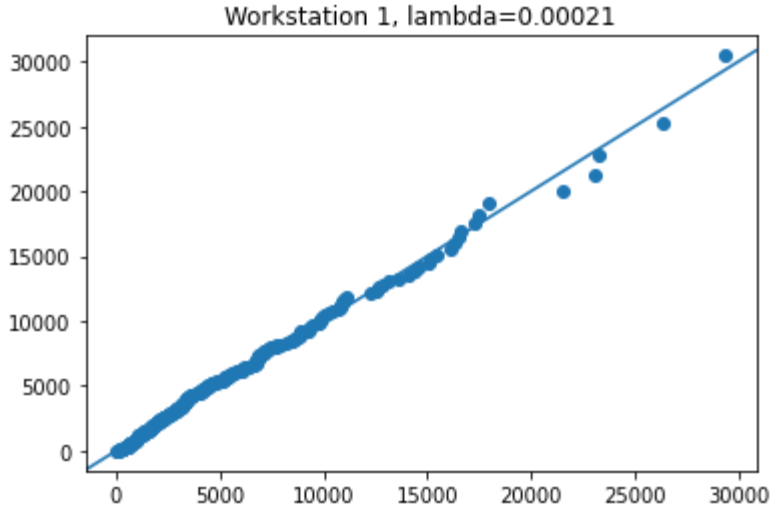
For inspector 2 - component 2 which demonstrated a Weibull distribution, a k-value of 1 and a lambda of 14500 yielded a relation that was most linear and conforming to the 45-degree reference axis



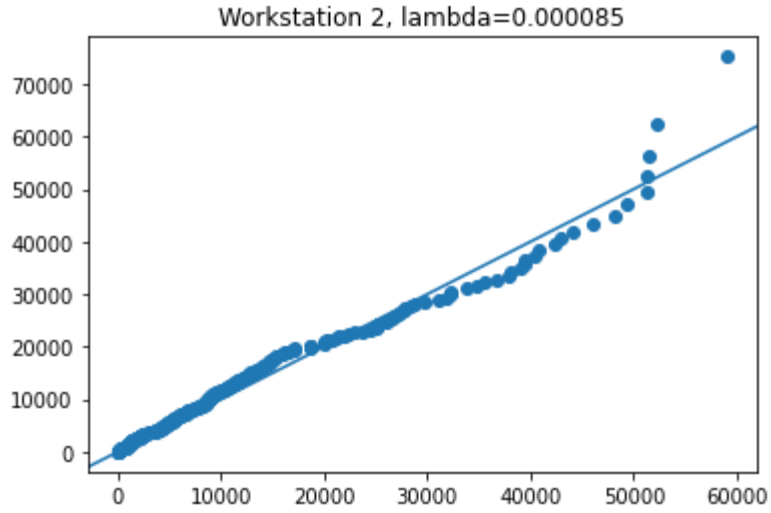
For inspector 2 - component 3 which demonstrated an Exponential distribution, a lambda of 0.00011 yielded a relation that was most linear and conforming to the 45-degree reference axis.



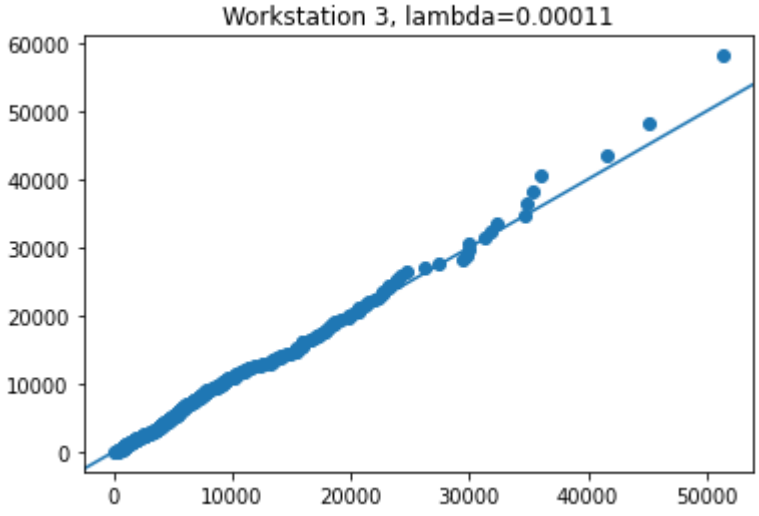
For workstation 1 which demonstrated an Exponential distribution, a lambda of 0.00021 yielded a relation that was most linear and conforming to the 45-degree reference axis.



For workstation 1 which demonstrated an Exponential distribution, a lambda of 0.000085 yielded a relation that was most linear and conforming to the 45-degree reference axis.



For workstation 3 which demonstrated an Exponential distribution, a lambda of 0.00011 yielded a relation that was most linear and conforming to the 45-degree reference axis.



To further validate if parameters hypothesized from the QQ plotting process are consistent with the dataset, a Chi-Squared test will be conducted on the dataset with reference to the particular hypothesized distribution and parameters.

Using the previously determined parameters Chi-Squared tests were used to test these parameters and see how well this data matched the distribution. This was accomplished by creating two functions for the Chi-Squared testing, one for the exponential and one for the Weibull.

For the **weibull_chi_squared** method it will take the input parameters of **self**, **bin_number** for the number of starting bins, **k_value** and **b_value**. The k and b values were determined by our parameter estimation for the two Weibull data sets. The first thing the method does is to get the sorted data list for the selected data set and saves this as **sorted_data**. The next step is to save the input **bin_number** to **k_intervals**. The bin width is then determined with the following function:

bin_width = round((max(sorted_data) - min(sorted_data)) / k_intervals)

This calculates the range of data points that are in each box by subtracting the lowest value in the list from the highest and dividing this by the **k_intervals**. This value is rounded and saved to **bin_width.** After this the **k_value** and **b_value** were set to **weibull_k** and **weibull_b**. The next step is to calculate the expected frequencies using the given parameters. To do this first an empty list is created called **expected_frequencies**. Then using a for loop for all the range of **k_intervals**, each interval has its expected frequency calculated. This is done using the following equation:

(self.weibull_cdf(i+1,weibull_k,weibull_b)-self.weibull_cdf(i,weibull_k,weibull_b))*300)

This uses a method called **weibull_cdf** that calculates the Cumulative distribution function. This method takes inputs of **self, x, k** and **b**. Using the math library, the following calculation is returned

1-e^(-1(x/b)^k)

This function is returned for the next interval (1+i) minus the current interval at i multiplied by 300, which is the sample data size. Each of these values is appended to the **expected_frequencies** list. This expected frequency list contains the expected frequencies for each of the intervals. Next the **observed_frequencies** list was initialized. Once again for all the **k_intervals** a for loop is iterated through that sorts the data list into the intervals. This checks if the value is within a specific interval and if it is it will increase the intervals counter. This list will now contain the data from the dat file sorted by frequency. Then there is a line that will check if the lists are the same length and will fix the observed list if they are different lengths.

The next part calls the **chi_squared_rebin** method to fix any low frequency bins. This method takes the **observed_frequencies**, the **expected_frequencies** and the threshold for the number of items in a bin and will reduce bins with frequencies lower than the threshold. The first part of the method is to verify the lists are the same length. This is because if the lists are different the program will error. The program then goes through the **expected_ frequencies** list forwards and backwards combining adjacent bins if they are not above the threshold quantity of 5. If bins for the expected frequency are combined the corresponding observed bins will be combined as well Once the rebinning is done the new observed and expected frequency lists are returned. Using the new observed and expected frequency lists

the Chi-Square calculation can be done. A for loop for the range of the length **observed_frequencies** the following calculation is done:

((observedFrequency-expectedFrequency)^2)/expectedFrequency

Each of these calculations is appended to a list called **chi_components**. These components will be totaled and compared with the **chi_square_table_value** of 339.2604 from the chi square table for 300 data points. If the sum of the **chi_components** is less then this value we will fail to reject the null hypothesis and we can assume the distributions match. If the value is higher then distributions do not match. The last thing the method does is print out the **chi_componets** sum for viewing.

For the **exponential_chi_squared** method it will take the input parameters of **self**, **bin_number** for the number of starting bins and **Lambda**. The **Lambda** value was determined by our parameter estimation for the four Exponential data sets. The first thing the method does is to get the sorted data list for the selected data set and saves this as **sorted_data**. The next step is to save the input **bin_number** to **k_intervals**. The bin width is then determined with the following function:

bin_width= round((max(sorted_data) - min(sorted_data)) / k_intervals)

This calculates the range of data points that are in each box by subtracting the lowest value in the list from the highest and dividing this by the **k_intervals**. This value is rounded and saved to **bin_width.**. After this **Lambda** was set to **exponential_lambda**. The next step is to calculate the expected frequencies using the given parameters. To do this first an empty list is created called **expected_frequencies**. Then using a for loop for all the range of **k_intervals**, each interval has its expected frequency calculated. This is done using the following equation (one expression shown across 2 lines):

((self.exponential_cdf(i+1,exponential_lambda)
-self.exponential_cdf(i,exponential_lambda))*300)

This uses a method called **exponential_cdf** that calculates the Cumulative distribution function. This method takes inputs of **self**, **x** and **ld**. Using the math library, the following calculation is returned

1-e^(-x*ld)

This function is returned for the next interval (1+i) minus the current interval at I multiplied by 300. Each of these values is appended to the **expected_frequencies** list. This expected frequency list contains the expected frequencies for each of the intervals. Next the **observed_frequencies** list was initialized. Once again for all the **k_intervals** a for loop is iterated through that sorts the data list into the intervals. This checks if the value is within a specific interval and if it is it will increase the intervals counter.This list will now contain the data from the dat file sorted by

frequency. Then there is a line that will check if the lists are the same length and will fix the observed list if they are different lengths.

The next part calls the **chi_squared_rebin** method to fix any low frequency bins. This method takes the **observed_frequency**, the **expected_frequency** and the threshold for the number items in a bin and will reduce bins with frequencies lower than the threshold. The first part of the method is to verify the lists are the same length. This is because if the lists are different the program will error. Then the program then goes through the **expected_frequencies** list forwards and backwards combining adjacent bins if they are not above the threshold quantity of 5. Once the rebinning is done the new observed and expected frequency lists are returned. Using the new observed and expected frequency lists the Chi-Square calculation can be done. A for loop for the range of the length **observed_frequencies** the following calculation is done:

((observedFrequency-expectedFrequency)^2)/expectedFrequency

Each of these calculations is appended to a list called **chi_components**. These components will be totaled and compared with the **chi_square_table_value** of 339.2604 from the chi square table for 300 data points. If the sum of the chi square components is less then this value we will fail to reject the null hypothesis and we can assume the distributions match. If the value is higher the distributions do not match. The last thing the method does is print out the **chi_componets** sum for viewing.

Using this system to test each of the distributions first required the parameters found to be scaled back down to the current units for the chi squared:

Inspector 1 Com 1 k = 1, lamda = 10000 → k = 1, lamda = 10
Inspector 2 Com 2 k = 1, lamda = 14500 → k = 1, lamda = 14.5
inspector 2 Com 3 lamda = 0.000 48 → lamda = 0.048
Workstation 1 lamda = 0.00021 → lamda = 0.21
Workstation 2 lamda = 0.000 85 → lamda = 0.085
Workstation 3 lamda = 0.00011 → lamda = 0.11

Using these values bin numbers between 17 and 60 were tested to see what would output the lowest Chi Squared Sum. The results are shown in the following table:

	Chi Total	Bin Number (Original)	Bin Number (Rebinned)	Fail to reject null hypothesis?	Distribution Type
Inspector 1 C1	32.7700	59	40	Yes	Weibull
Inspector 2 C2	84.8121	60	50	Yes	Weibull
Inspector 2 C3	79.7825	56	55	Yes	Exponential
WS 1	9.7014	30	19	Yes	Exponential
WS 2	41.3009	56	55	Yes	Exponential
WS 3	29.4318	53	36	Yes	Exponential

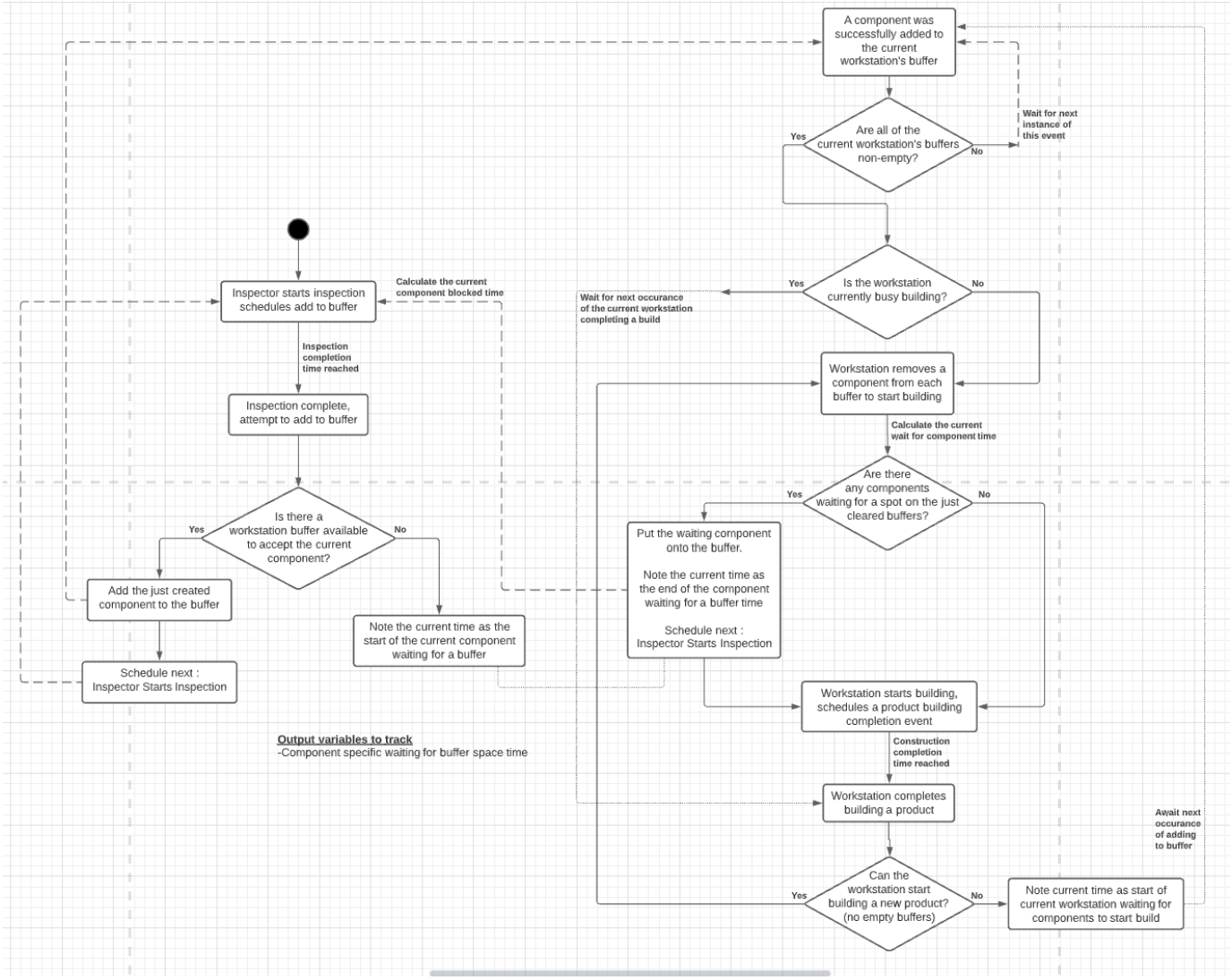
Testing all six of the data sets results in the Null hypothesis not being rejected and we can assume that the distribution does match. Each of the tests were conducted altering the bin number until the lowest **chi_total** was found. The number of bins one more and less then the starting numbers were tested to ensure that the lowest total was achieved. The functionality of our Chi-Squared process was tested further by using a collection of 300 randomly generated variables using the parameters. 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.

Simulation Verification

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 document. To verify the conceptual model, a flow chart has been created to show how future simulation events shall be generated based on the initial events, and this flowchart can be found below. This flow chart also details the measurement of the key output variables, which is the component specific inspector blocked times for when an inspector is waiting for a buffer to become available.

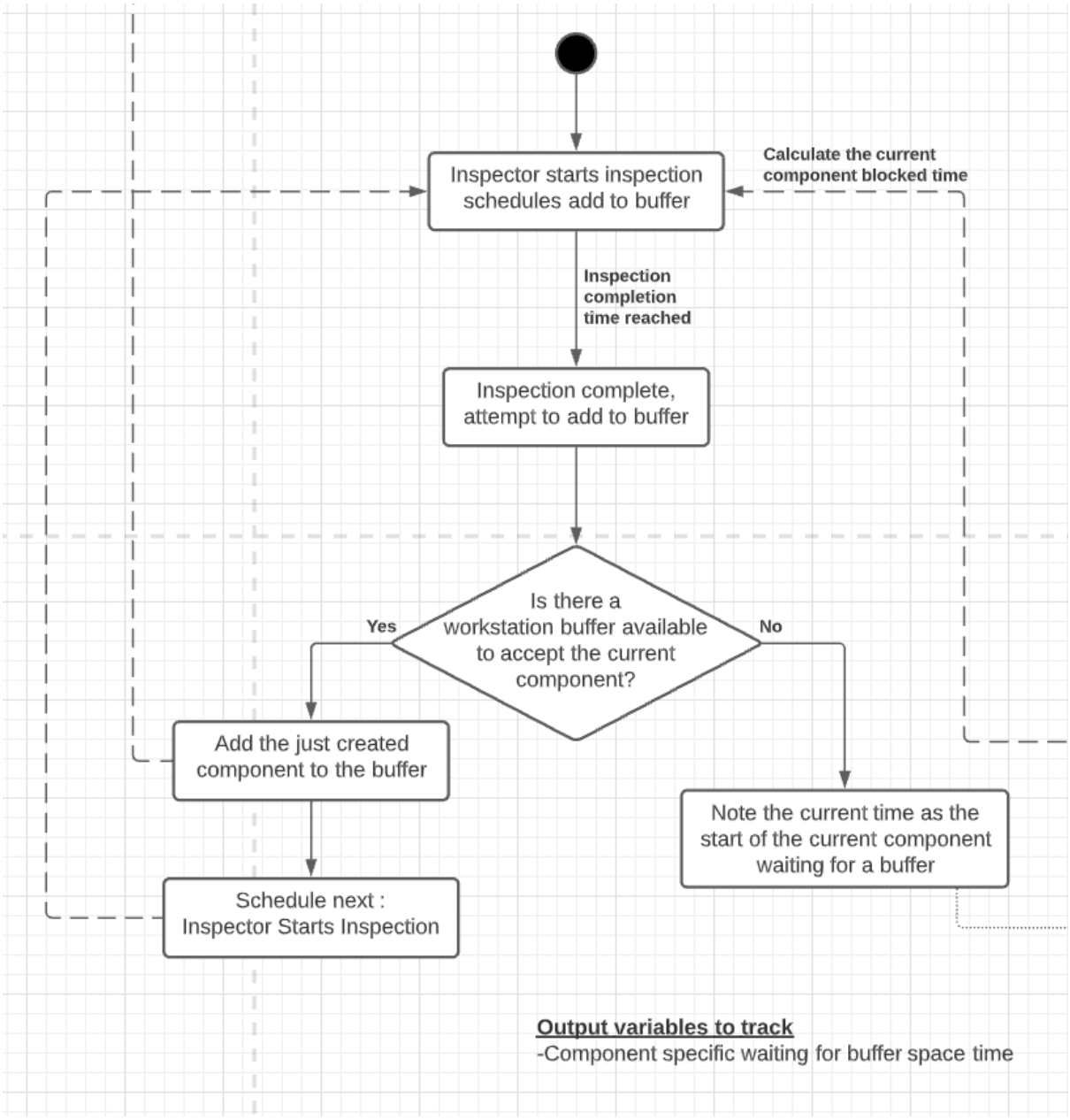
These output variables will then be generated across many different simulation runs to determine the confidence intervals.

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 in debug mode, 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.

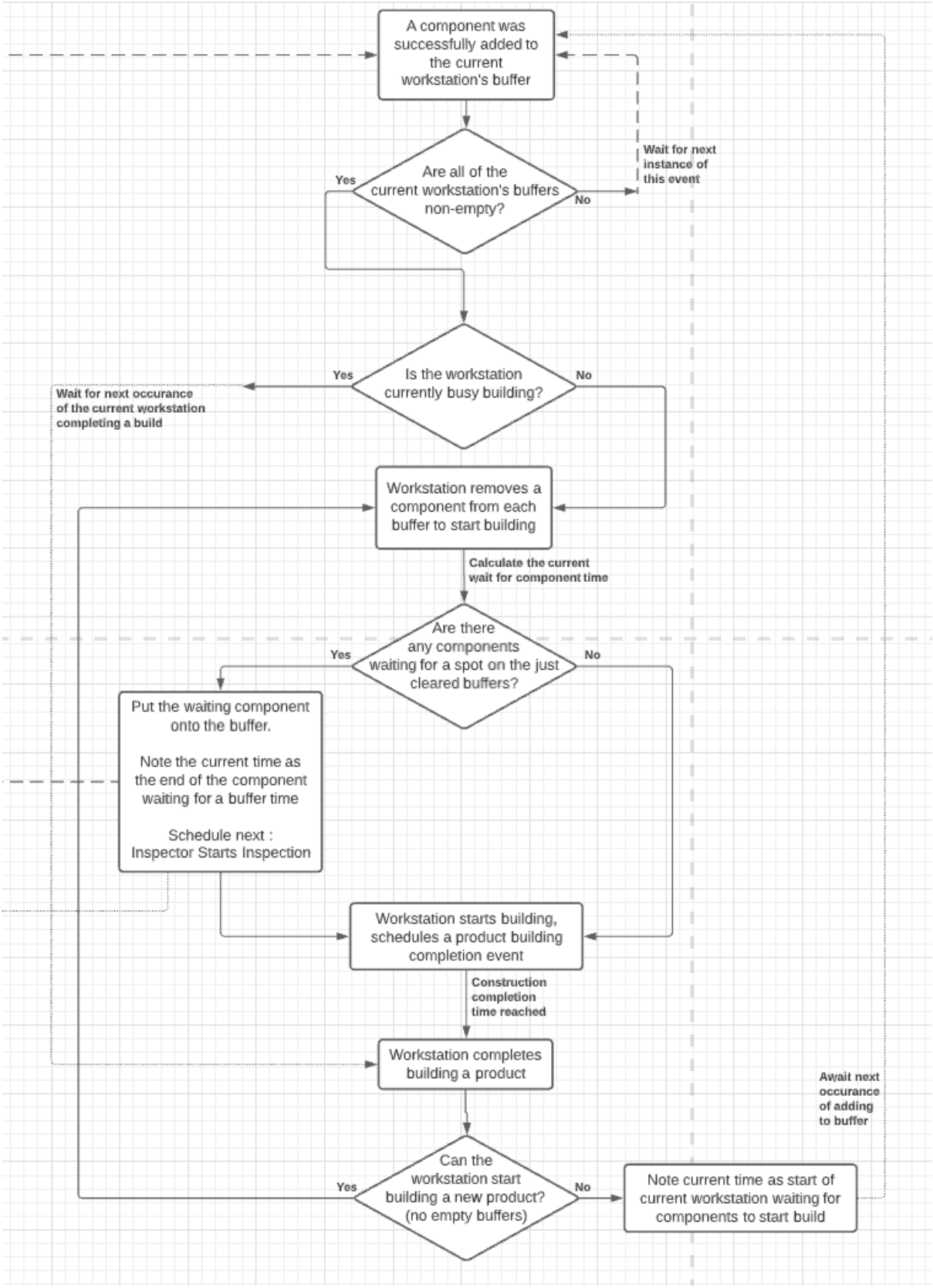


Overall Flowchart Overview

Scroll down to see Inspector and Workstation areas in more detail



Inspector side of flow chart



Workstation side of flowchart

Given these flowcharts and traversal through the program with a debugger to verify that all cases were covered by all group members, it was concluded that the model accurately represents the flow chart modeling of the manufacturing facility.

7 - Simulation Validation

Typically, the process for validating a simulation model involves comparing the model to real life observations to determine if the model is an accurate and consistent representation of the real life phenomena. In this project, the purpose is to model a manufacturing facility, and given that we were given collections of sets of historical data for component inspection and product build times, it can be assumed that the manufacturing facility exists in real life, and that it would be feasible to make a request to obtain additional observations to obtain a more detailed historical dataset that corresponds with the output variables of interest in the simulation model. Knowing the real life observed true values for the output variables of interest, a collection of simulation generated output variables can be used to determine a simulation mean & standard deviation to determine a confidence interval for the output variables to compare to the observed true values.

If the simulation's calculated confidence interval contains the observed true value, then the model can be accepted if the worst case error is less than or equal to the epsilon error threshold. If the worst-case error is greater than the epsilon error threshold value with the confidence interval containing the true value, then more simulation replications will be required to reach a conclusion.

If the simulation's calculated confidence interval does not contain the empirically observed true value, then the best-case and worst-case errors shall be compared to the epsilon error threshold to determine if the model can be accepted, requires more replications to come to a conclusion, or if the model needs to be revised & improved. First check if the worst-case error is less than the epsilon error threshold. If so, then the model can be accepted, otherwise compare the best-case error with the epsilon error threshold. If the best-case error is less than the epsilon error threshold, then more replications are needed to come to a conclusion. If the best-case error is greater than the epsilon error threshold, then the model cannot be accepted, and should be revised and adjusted before restarting the validation process.

In light of the fact that the real life version manufacturing process detailed in this project is hypothetical, there is currently no way to know the true 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. Given this, simulation output variable confidence intervals will still be generated, and validation will then be done by ensuring that the output variables are reasonably close to each other between different simulation runs. This is to ensure that the output variables are reasonably precise, and accessing the accuracy could be deferred to a later task if/when the observed true output variables are determined. Given a confidence interval of the simulation output parameters, the width can provide insight into what epsilon error thresholds 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.

Output Variables & Confidence Intervals

Under the assumption that the current simulation implementation is verified and validated, we can then proceed to run multiple simulation replications so that confidence intervals for the output

Number of Replications

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

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

Where R is the number of replications, z(alpha) is the z-value at alpha, S0 is the standard deviation of data, and epsilon is the error threshold of the data. With this, the ideal replication count for each output variable from both the standard and alternate operating procedures can be calculated. An error value of 10% of mean is used for each calculation as the target is to be within 10% of the mean.

$$R \geq \left(\frac{z_{0.025} S_0}{\varepsilon}\right)^2 = \left(\frac{1.96 \cdot 1689.903}{0.1 \cdot 2763.885}\right)^2 = 143.614$$

$$R \geq \left(\frac{z_{0.025} S_0}{\varepsilon}\right)^2 = \left(\frac{1.96 \cdot 119368.229}{0.1 \cdot 93002.935}\right)^2 = 632.844$$

$$R \geq \left(\frac{z_{0.025} S_0}{\varepsilon}\right)^2 = \left(\frac{1.96 \cdot 11357.371}{0.1 \cdot 12253.346}\right)^2 = 330.034$$

$$R \geq \left(\frac{z_{0.025} S_0}{\varepsilon}\right)^2 = \left(\frac{1.96 \cdot 1181.337}{0.1 \cdot 1715.363}\right)^2 = 182.200$$

$$R \geq \left(\frac{z_{0.025} S_0}{\varepsilon}\right)^2 = \left(\frac{1.96 \cdot 125473.042}{0.1 \cdot 130548.693}\right)^2 = 354.869$$

$$R \geq \left(\frac{z_{0.025} S_0}{\varepsilon}\right)^2 = \left(\frac{1.96 \cdot 7683.117}{0.1 \cdot 19000.898}\right)^2 = 62.811$$

The maximum replication value is 633 which is what is used for the simulation. Using this number of replications takes approximately 40 minutes for both the standard and alternate operating procedure.

Initialization Phase

Initialization bias is a phenomenon in which the initial conditions of the simulation may be unrealistic and non-representative of the model's actual conditions. In order to mitigate this bias, the simulation is divided into an initialization phase and data collection phase. The initialization phase is a period of time in which the simulation can ramp up to normal operation where data is not collected. Once the operation has ramped up to steady-state conditions, data collection can reliably begin.

To implement this initialization phase, an event could be handled the moment that the initialization phase ends and the production phase starts to reset all of the output variables so that all future adjustments will only consider the events occurring during the production phase, and will disqualify all events that occurred in the initialization period from consideration.

For this project, it was determined that the effort and time budget required to implement such a system would have limited benefit, as the impact of initialization bias on the output variables becomes less significant as the length of each simulation replication increases. Currently, a single simulation replication stops after 100,000 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 group decided to proceed with gathering results without using an initialization period. This design choice may manifest itself as a greater level of variance for the output variables, thus leading to wider confidence intervals, however the width of the confidence interval can be reduced by increasing the replication count.

Alternate Operating Policy

The alternate operating policy used in this project will change the behavior of inspector 2 to check the buffer capacities to determine which component to inspect next instead of randomly selecting the next component to select. The buffers will first be checked to determine which workstation has the least elements on their inspector 2 buffer, and then that component will be inspected next by inspector 2. The behavior of inspector 1 will remain unchanged as its standard operating policy used in previous iterations is already operating optimally based on what inspector 1 is capable of and is able to know. However, it is hypothesized that the average wait time for inspector 1 will be decreased as more efficient inspections by inspector 2 will improve the production rates of the workstation which will then increase the rate at which the buffers clear for inspector 1.

This alternate operating policy was implemented by adding a new method to **`buffers.py`** called **`choose_next_inspector2_component`** which returns component C2 if the workstation 2 C2 buffer is less full, and component C3 if the workstation 3 C3 buffer is less full. In the event that both buffers have the same number of contained components at the time of inspection, then the system will default to its standard operating policy and randomly select the next component to inspect. It was considered to add a system that determines the next component to inspect based on which buffer would clear first chronologically based on the future event list, however, accessing the future event list to determine when workstations

would complete their assemblies may not properly represent reality as it is often impossible to determine exactly how long a process will take as shown by the impossibility to implement 100% accurate progress bars in software, and Alan Turing's 1937 proof to address the Halting problem. Given this, a simple snapshot analysis of buffer occupancy shall suffice for this alternative operating policy. Inside **`Inspectors.py`** , both the Inspector1 and Inspector2 classes got updates to their **`generate_inspect_time`** methods to expect a new component parameter. Given that Python does not have strict type rules for the lists that are used to hold the inspectors, the execution in **`MainSim.py`** calls the same **`generate_inspect_time`** method on both inspectors meaning that both inspector classes need to have methods with matching method signatures. Inspector1 does nothing with its added component parameter, whereas Inspector2 will return an inspection time corresponding to the given component. If the component is None/null, then Inspector2 will randomly choose a component to inspect along its standard operating policy. Inside **`MainSim.py`**, there is a boolean feature flag to decide if the standard or alternate operating policy is to be used. This feature flag will determine if None/null is always passed as the component parameter to the inspector's **`generate_inspect_time`** methods, or if the component to inspect will be chosen based on buffer contents. Implementing this feature flag makes it easy to switch between the standard and alternate operating policies with a simple one line change. Using the same simulation stopping point mentioned above and the same number of replications, the same output variables between the different operating policies can be compared to determine if the performance of the alternate operating policy will increase manufacturing efficiency.

9 - Operating Policy Output Variable Comparison

NOTE : Times from the dat files were given in minutes with 3 decimal point precision, however, to avoid floating point number errors, they were all multiplied by 1000 before use in the simulation. Time units are in milli-minutes which translate to 0.06 seconds/milli-minute or 60 milliseconds/milli-minute
NOTE : All confidence intervals are calculated at 95%

Operating policy 1 - Standard

Component 1 average buffer blocked time : 2763 ± 148 milliminutes
Component 2 average buffer blocked time : 130548 ± 11000 milliminutes
Component 3 average buffer blocked time : 19231 ± 673 milliminutes

```
Operating Policy: Standard, Inspector 2 randomly inspects
Number of products to make before stopping : 100000
Number of simulation replications: 500

Component #1 Average waiting time
Mean: 2763.884730200943
Standard deviation: 1689.9032355168279
n: 633

Component #2 Average waiting time
Mean: 130548.69259886343
Standard deviation: 125473.04213370643
n: 633

Component #3 Average waiting time
Mean: 19230.897505014313
Standard deviation: 7683.117090348424
n: 633
```

Operating policy 2 - Alternate

Component 1 average buffer blocked time :1715 ± 104 milliminutes
Component 2 average buffer blocked time : 93002 ± 10500 milliminutes
Component 3 average buffer blocked time : 12253 ± 995 milliminutes

```
Operating policy : Alternate, Inspector 2 inspects based on buffers
Number of products to make before stopping : 100000
Number of simulation replications: 500

Component #1 Average waiting time
Mean: 1715.362556702877
Standard deviation: 1181.3366935087688
n: 633

Component #2 Average waiting time
Mean: 93002.9348326631
Standard deviation: 119368.22851102626
n: 633

Component #3 Average waiting time
Mean: 12253.346221178977
Standard deviation: 11357.371276354526
n: 633
```

To compare the output variables between the two operating policies, the following equation will be used to determine the confidence interval of their differences.

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

Due to the fact that the variances of the two operating policies are not similar enough for a reasonable assumption of equivalence, the following equations will be used to determine the degrees of freedom and the s.e.(Ybar.1-Ybar.2) values in the above equation.

$$s.e.\left(\bar{Y}_{.1} - \bar{Y}_{.2}\right) = \sqrt{\frac{S_1^2}{R_1} + \frac{S_2^2}{R_2}}$$

$$\upsilon = \frac{\left(S_1^2/R_1 + S_2^2/R_2\right)^2}{\left[\left(S_1^2/R_1\right)^2/(R_1-1)\right] + \left[\left(S_2^2/R_2\right)^2/(R_2-1)\right]}, \quad \text{round to an interger}$$

Comparison Confidence Intervals

NOTE: Alpha = 0.05 for all

Component 1

s.e.(Ybar.1-Ybar.2) = 81.95218164977152

Degrees of Freedom = 5860

t value = 1.9604 *(Essentially z value with such a high DoF)*

Confidence interval = 1049 ± 161 milliminutes

C.I. Lower Bound as percent of standard operating policy mean = 32%

Component 2

s.e.(Ybar.1-Ybar.2) = 6883.395580944891

Degrees of Freedom = 2800

t value = 1.9608 *(Essentially z value with such a high DoF)*

Confidence interval = 37546 ± 13497 milliminutes

C.I. Lower Bound as percent of standard operating policy mean = 18%

Component 3

s.e.(Ybar.1-Ybar.2) = 545.004848787751

Degrees of Freedom = 1343

t value = 1.9617 *(Essentially z value with such a high DoF)*

Confidence interval = 6748 ± 1069 milliminutes

C.I. Lower Bound as percent of standard operating policy mean = 30%

Given the above confidence intervals, it can be observed that the ranges are entirely positive, with the lower bound of each range still being a significant percentage of the mean for the standard operating policy. This indicates that the mean component block times with the alternate operating policy are noticeably shorter than the mean component block times of the standard operating policy, which indicates higher inspector utilization rates and thus better performance.

Conclusion

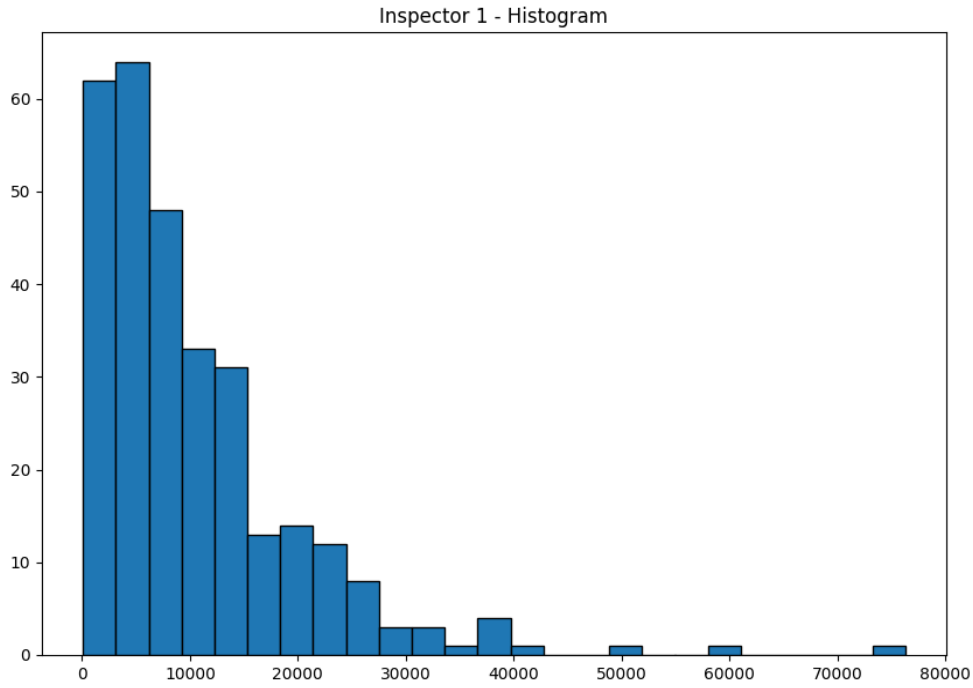
As shown above, we can expect to see an increase in performance when using the alternate operating policy over the standard operating policy. The team believes that the increase in performance is due to the fact that inspector 2 will no longer risk making a “wrong choice” for the next component to inspect. An example of a “wrong choice” is if the C3 buffer is full, C2 buffer is not full, and Inspector 2 still selects C3 to inspect next, or vice versa. The alternate operating policy also prevents the possibility of inspector 2 choosing the same component repeatedly, which would starve one of the workstations from starting builds, and increases the likelihood of getting blocked by the full buffer. The simple addition of checking the buffers before inspector 2 chooses the next component to inspect reduces the likelihood and frequencies in which the above conditions occur, thus increasing performance.

To relate the results of this simulation model to the “real life” manufacturing facility, there would most likely be costs associated with providing Inspector 2 with the necessary resources to change its operating policy to match the alternate operating policy implemented in the simulation model. Given that manufacturing facilities generally operate with very substantial capital investment & the need to operate on a large scale to be financially sustainable, it can be assumed that the manufacturing process will continue for a significant amount of time, produce a large number of completed products, and that the price of the completed products is sufficiently stable. As the manufacturing process continues for longer periods of time, the economic benefit gained through the increased efficiency of the alternate operating policy implementation will increase, and those economic benefits should eventually surpass the resource costs of implementing the alternate operating policy, thus justifying its implementation.

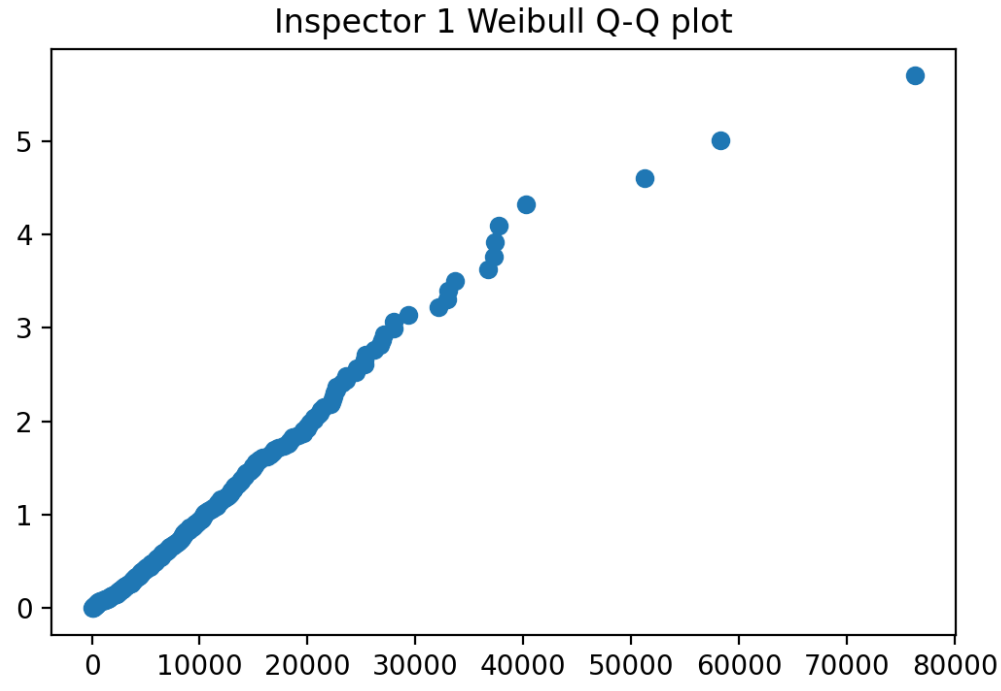
Given this, it is also important to consider that the above conclusion is based on a simplified simulation that models a description of a hypothetical “real life” manufacturing facility. Economically viable manufacturing at scale is an extremely complex process involving many interconnected variables with often chaotic interdependencies, and non-linear responses. The simulation results and conclusions should then only be used as a guide for how to proceed with the manufacturing process, however, empirically measured performance data should always take precedence over simulation conclusion when used to inspire informed business decisions, especially considering the larger economic consequences of decisions in this area.

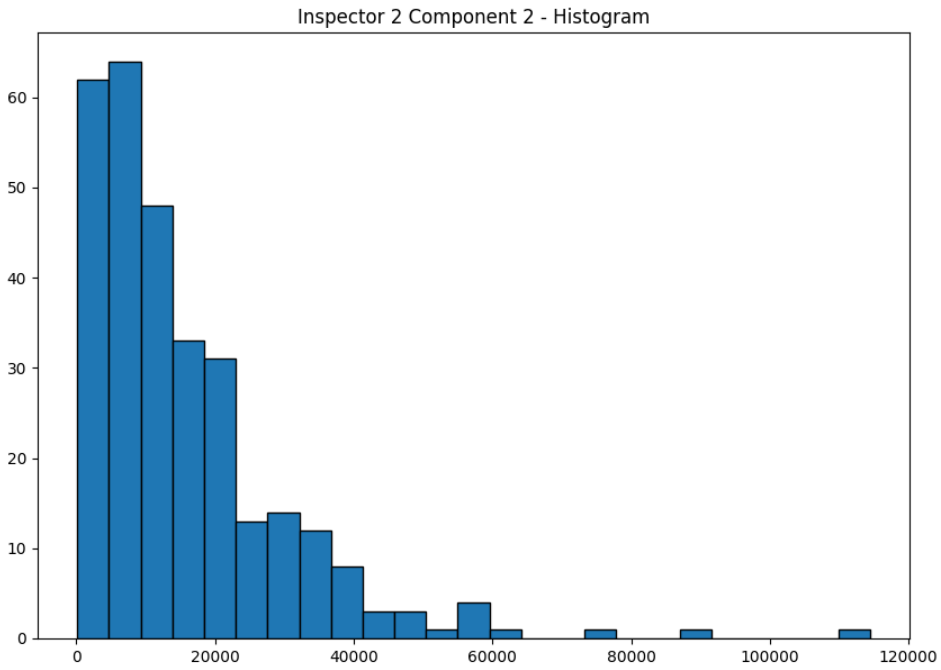
Appendices:

Appendix A - Analysis of Sample Data for Simulation Entities

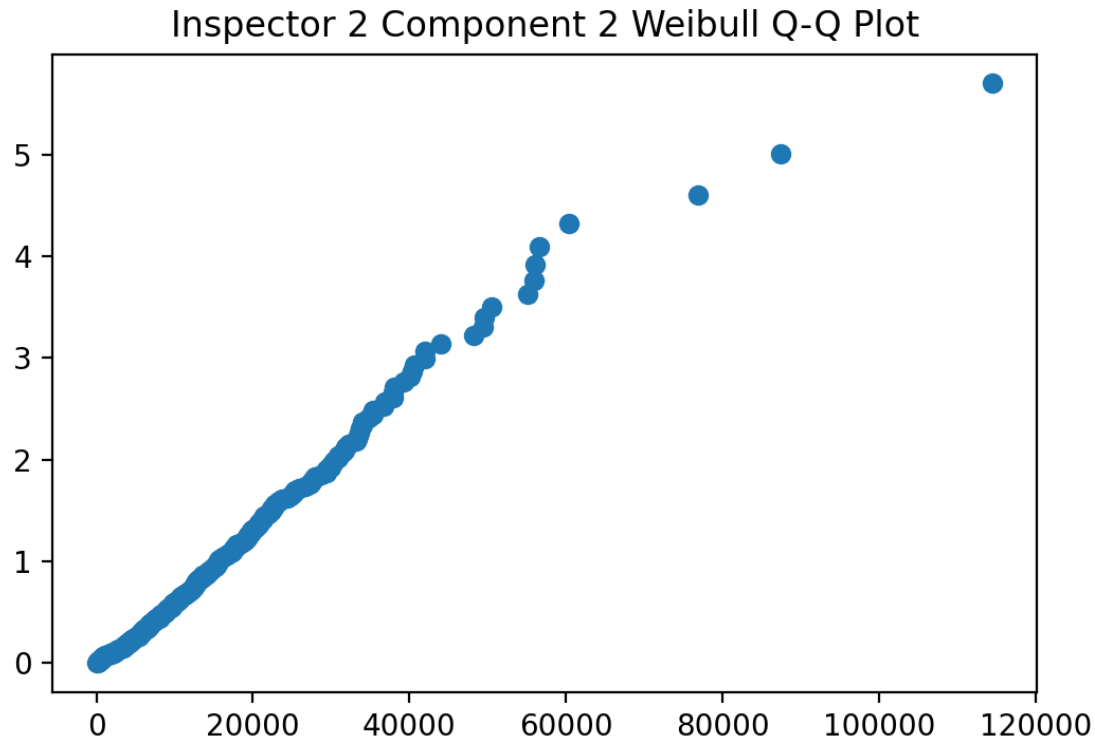


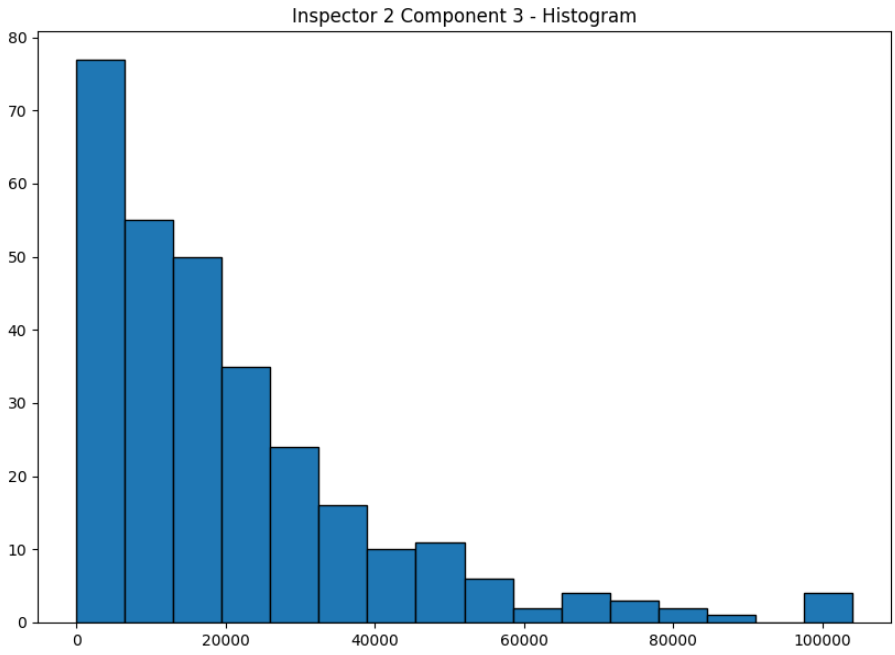
The distribution displayed above seems to follow a Weibull distribution with the first bin being smaller than the second, and then a downward trend following. Using a Weibull distribution, QQ and Chi-squared testing will be conducted to test the fit. If the Weibull distribution is unsuccessful a different distribution will be attempted. Then the inverse transform method of random variate generation will be conducted on the distribution. Below is a QQ plot for this sample data set. This QQ plot is a straight line showing that the correct distribution was selected.



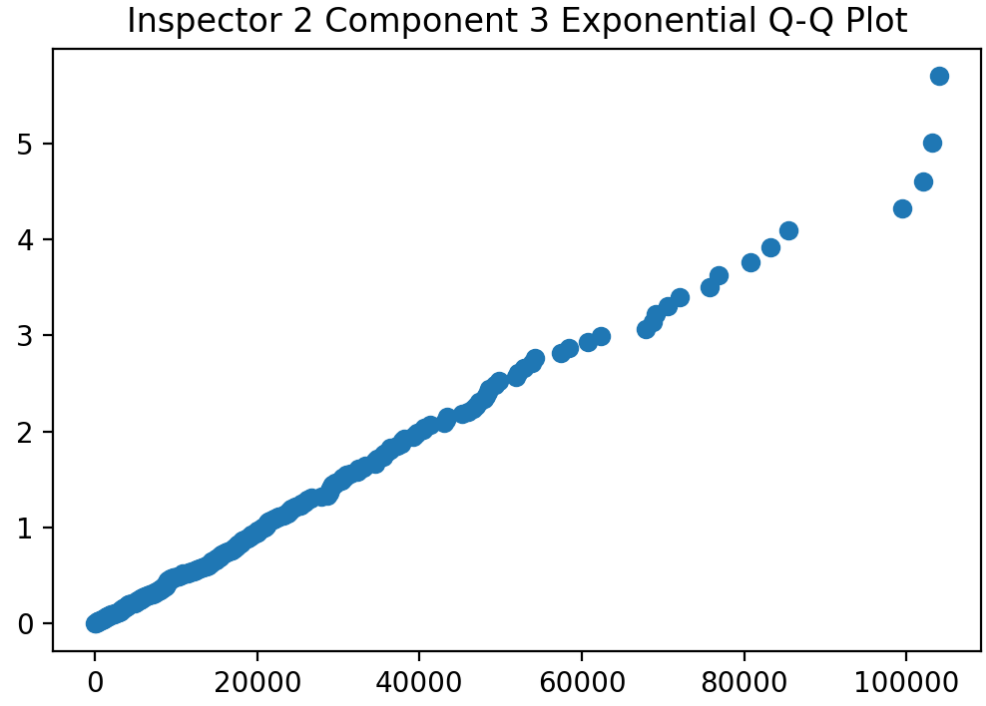


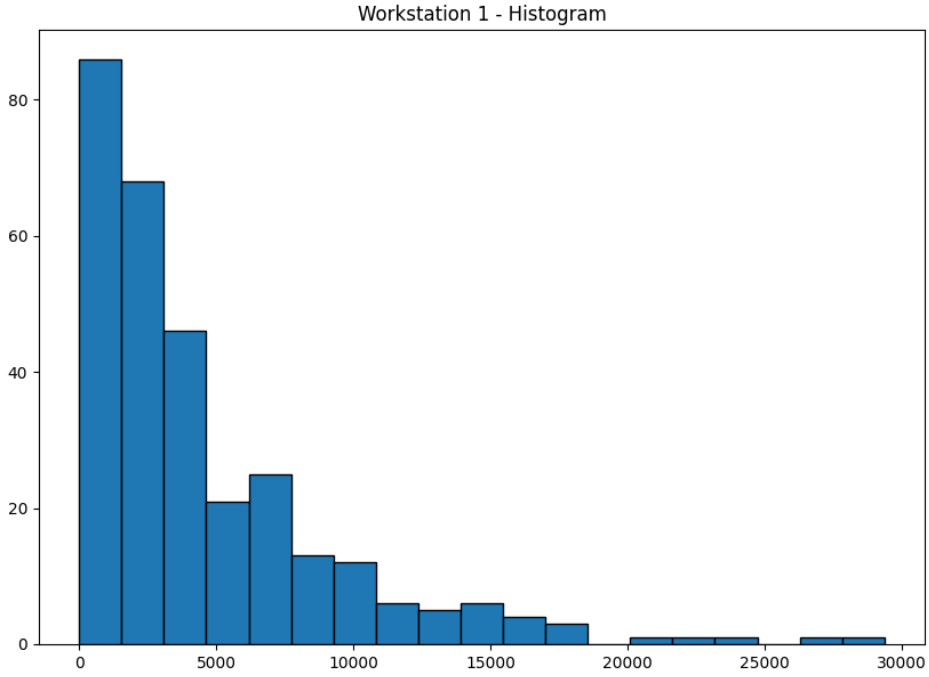
The distribution displayed above seems to follow a Weibull distribution with the first bin being smaller than the second, and then a downward trend following. Using a Weibull distribution, QQ and Chi-squared testing will be conducted to test the fit. If the Weibull distribution is unsuccessful a different 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 a Weibull distribution. This QQ plot is a straight line showing that the correct distribution was selected.



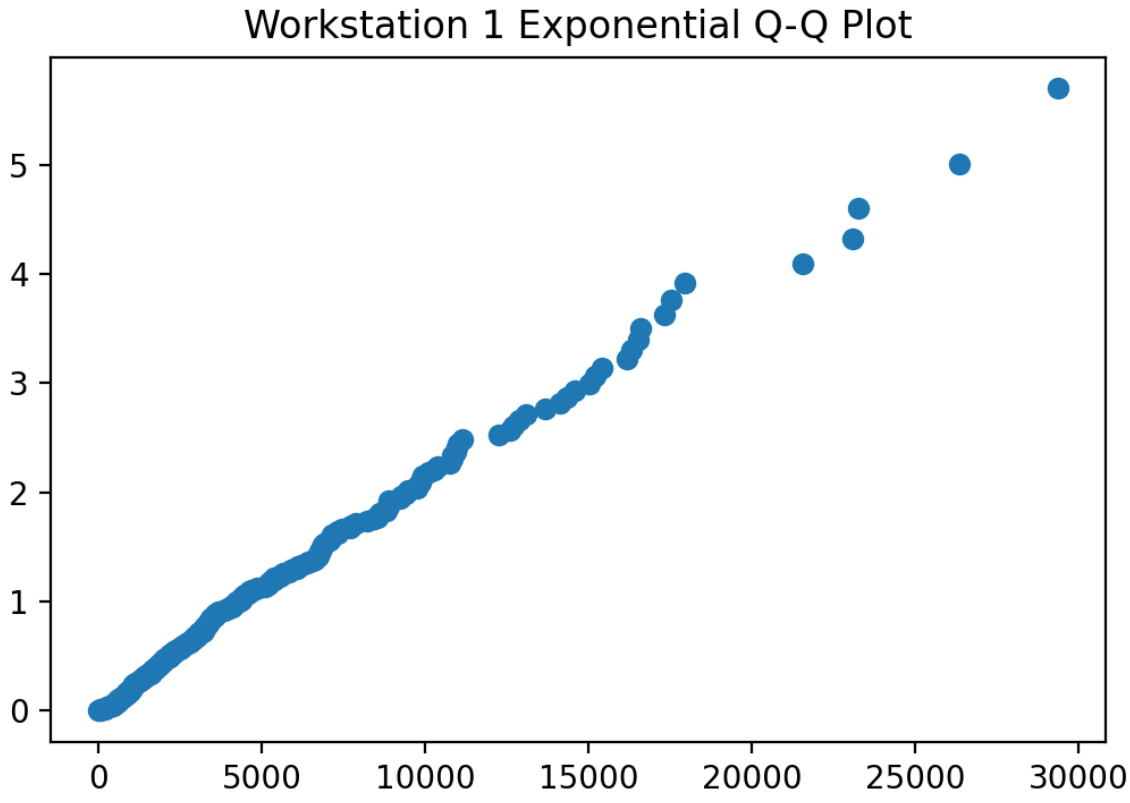


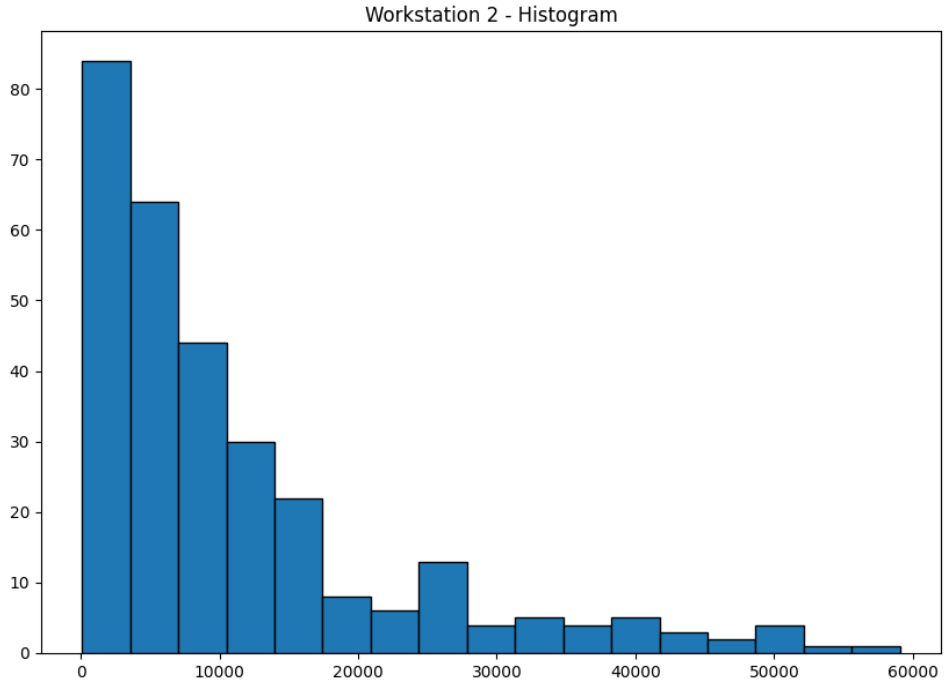
The distribution displayed above seems to follow an exponential distribution with a continuous downwards trend. Using an exponential distribution, QQ and Chi-squared testing will be conducted to test the fit. If the exponential distribution is unsuccessful a different distribution will be attempted. 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.



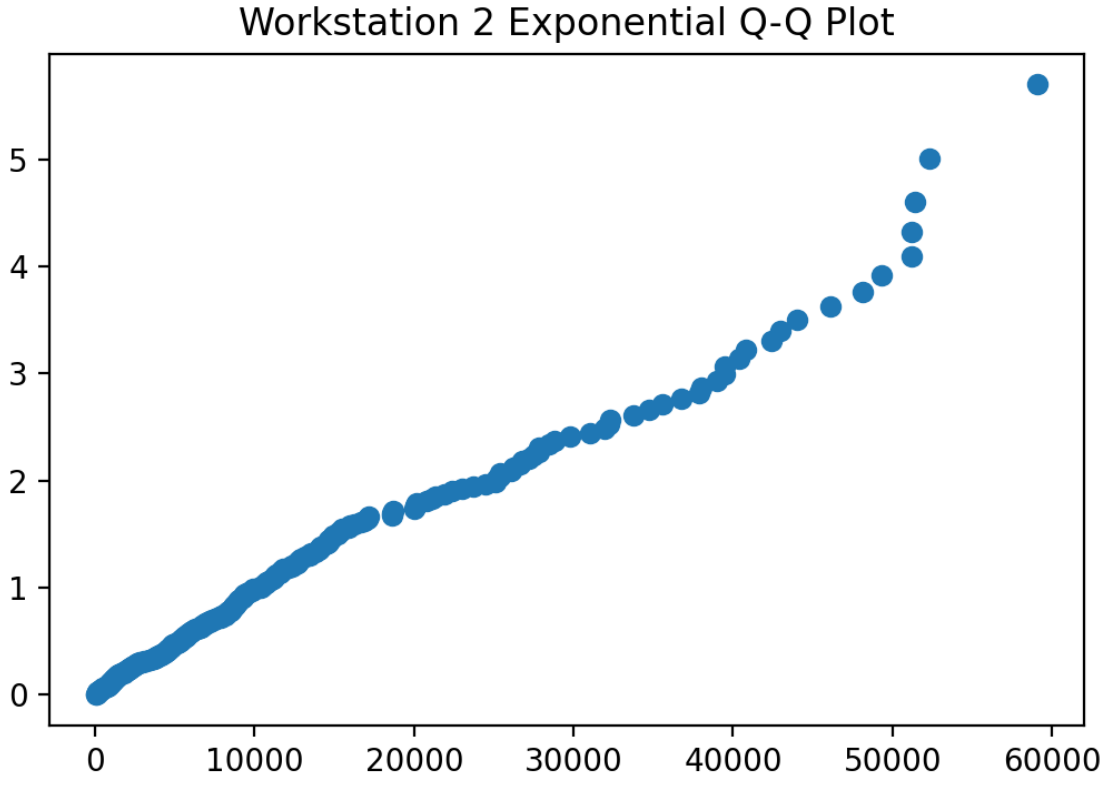


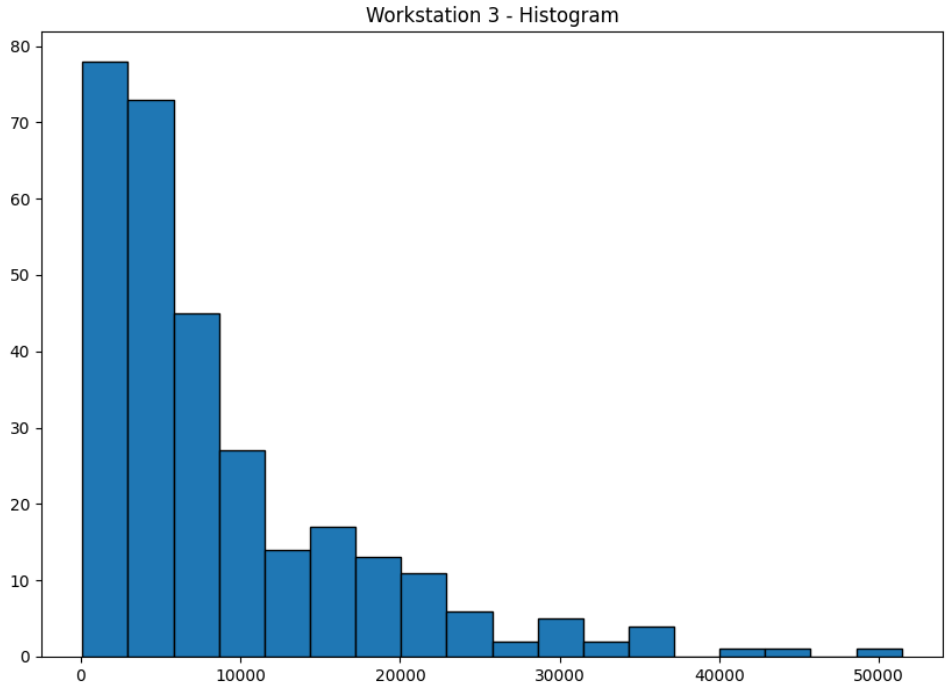
The distribution displayed above seems to follow an exponential distribution with a continuous downwards trend. Using an exponential distribution, QQ and Chi-squared testing will be conducted to test the fit. If the exponential distribution is unsuccessful a different distribution will be attempted. 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 showing that the correct distribution was selected.



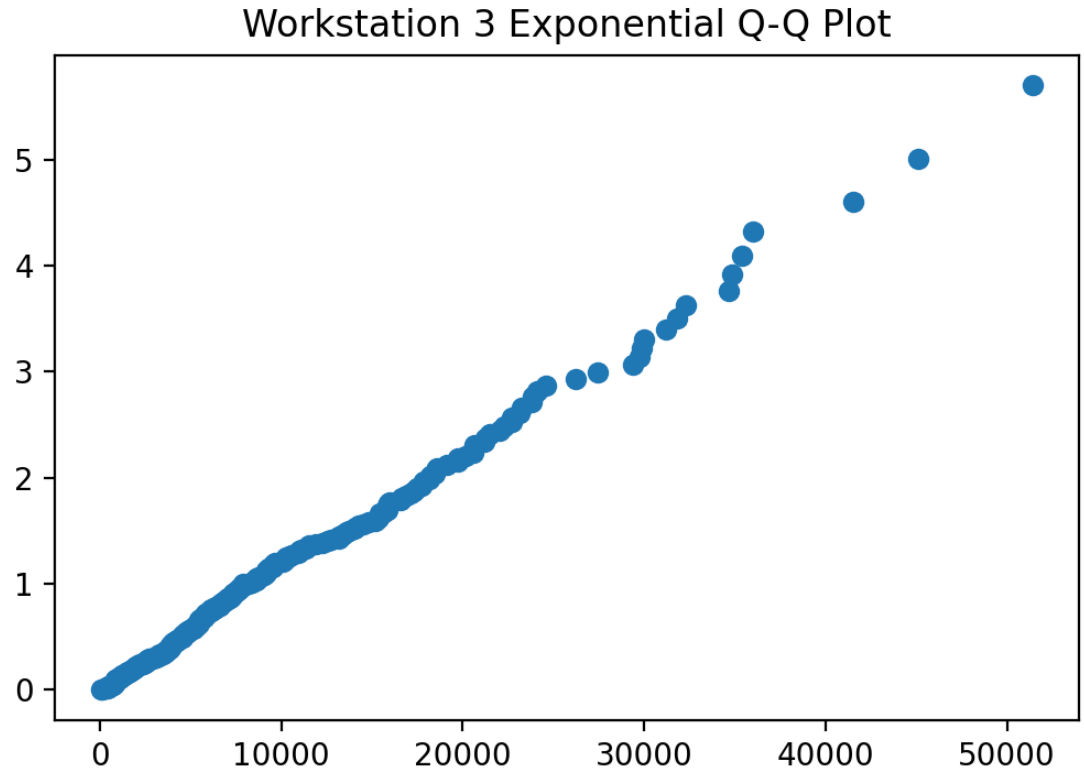


The distribution displayed above seems to follow an exponential distribution with a continuous downwards trend. Using an exponential distribution, QQ and Chi-squared testing will be conducted to test the fit. If the exponential distribution is unsuccessful a different distribution will be attempted. 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.

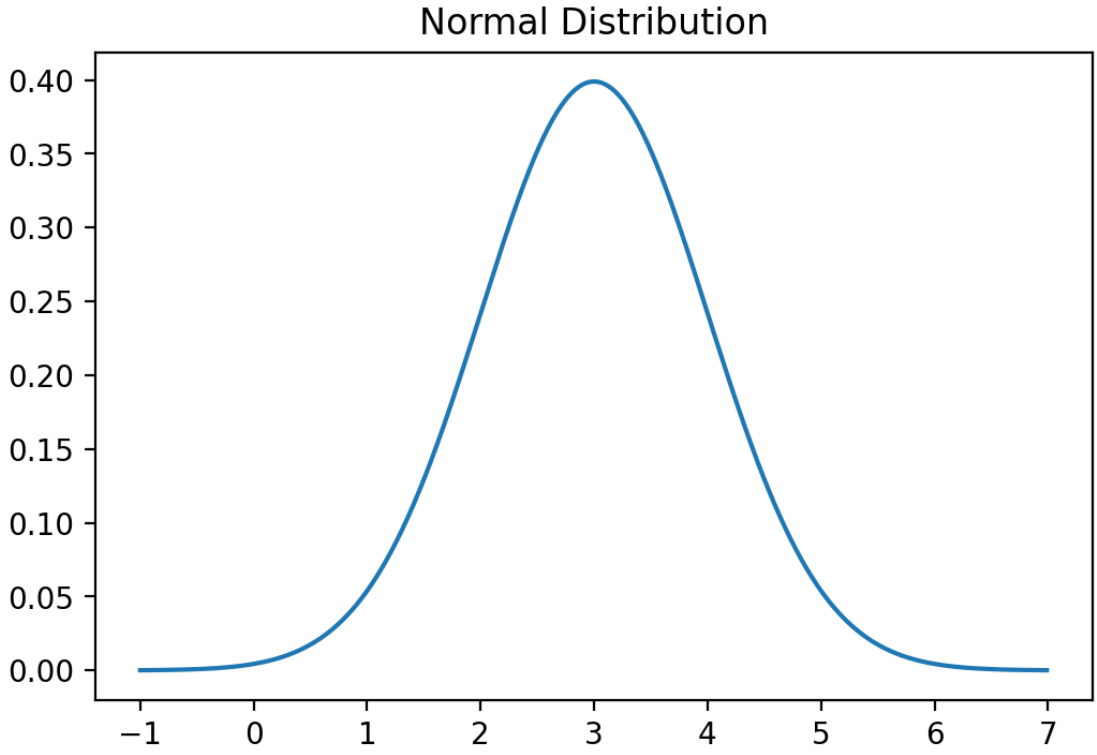


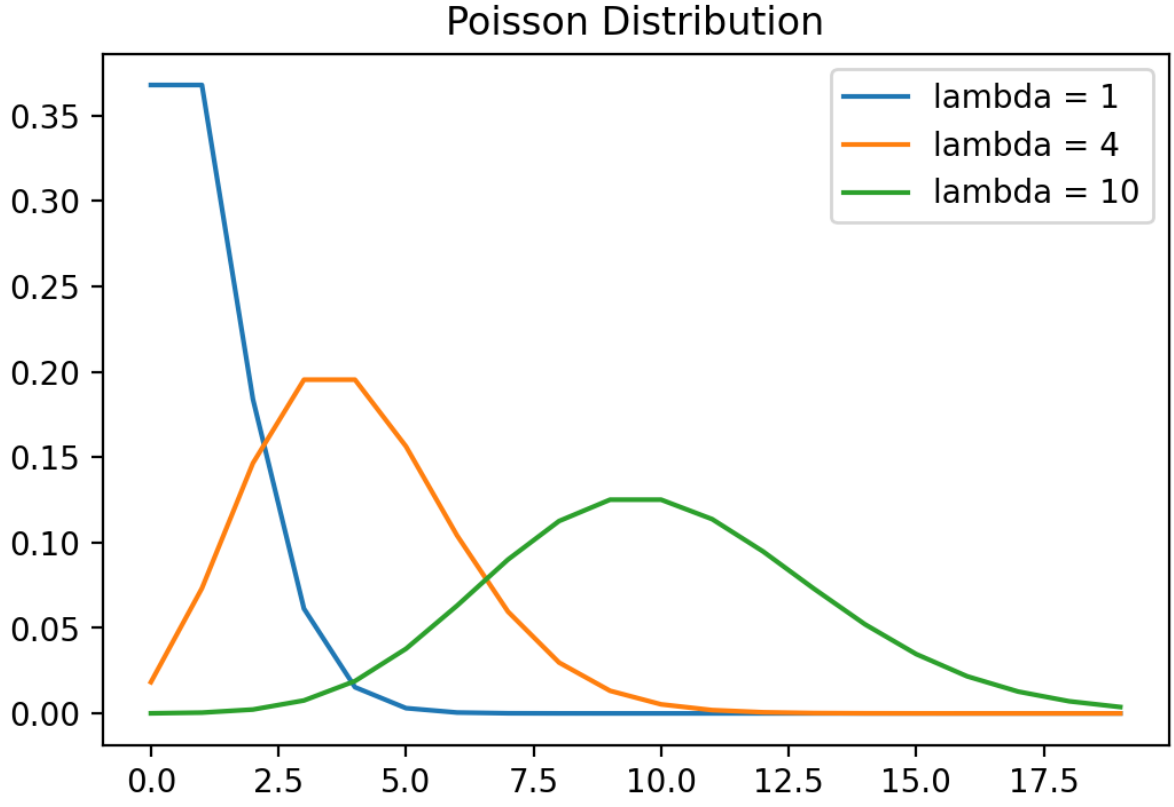


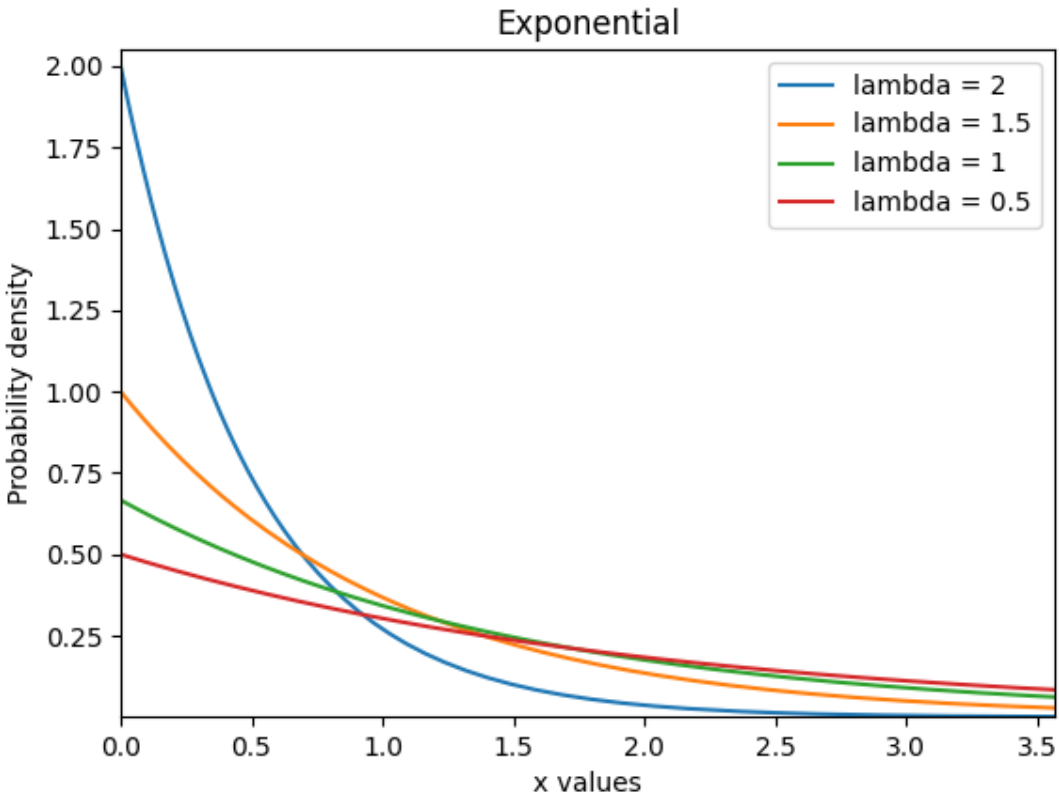
The distribution displayed above seems to follow an exponential distribution with a continuous downwards trend. Using an exponential distribution, QQ and Chi-squared testing will be conducted to test the fit. If the exponential distribution is unsuccessful a different distribution will be attempted. 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 showing that the correct distribution was selected.

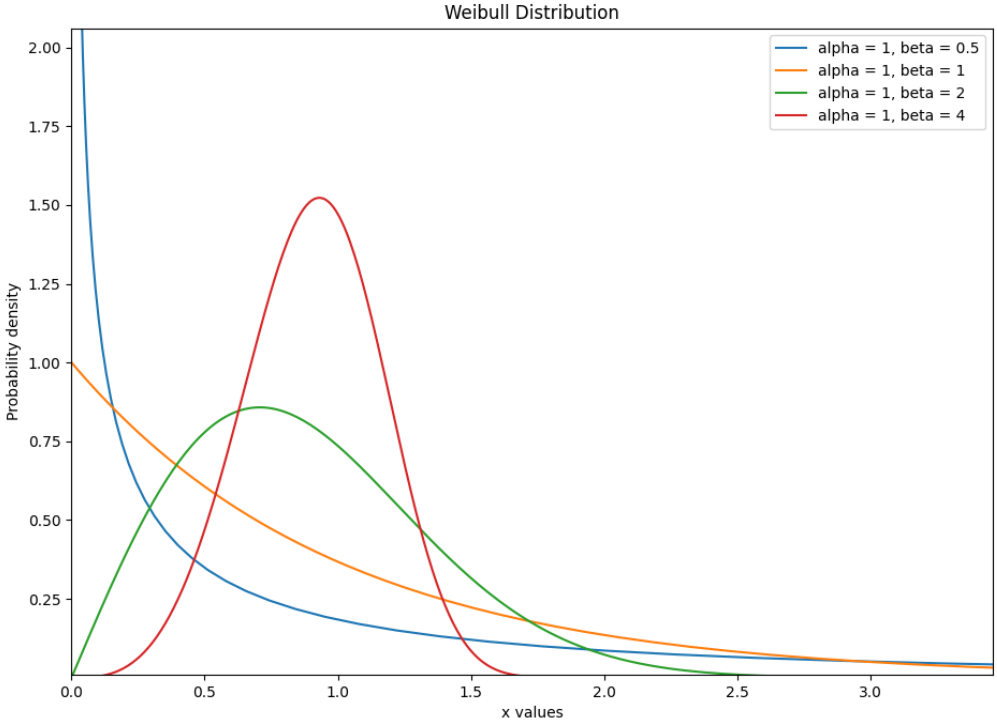


Appendix B - Shapes of Distributions for Reference






Exponential
lambda = 2
lambda = 1.5
lambda = 1
lambda = 0.5
Probability density
x values
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5


Weibull Distribution
alpha = 1, beta = 0.5
alpha = 1, beta = 1
alpha = 1, beta = 2
alpha = 1, beta = 4
Probability density
x values
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.0
0.5
1.0
1.5
2.0
2.5
3.0