Aaron Fox
CECS 622
Dr. Elmaghraby

Homework 2

Problem 1:

1. Generate N uniform numbers.
N uniform numbers were generated using Python's `random.uniform` function, which included numbers in the range [a, b] (a and b inclusive.) Per the Python documentation, random.uniform selects a "random floating point number *N* such that `a <= N <= b` for `a <= b` and `b <= N <= a` for `b < a`" [1]. Efforts were made to ensure Python's `random.random` was *not* used, for it only includes numbers in the range [a, b), which is only semi-open, although still uniform per the documentation.

The numbers themselves were generated using the method `generate_uniform_numbers`, which read:

```
# Generates N uniform random numbers
def generate_uniform_numbers(self, N):
    uniform_numbers = []
    for i in range(0, N):
        uniform_numbers.append(
            [random.uniform(0, 1), random.uniform(0, 1)])
    return uniform_numbers
```

This function used `random.uniform` to generate a list of `N` length of points as specified by the user in the GUI.

2. Calculate the value of pi for different values of N
The value of pi was calculated inside the `generate` function within the `Windows` GUI class of the program. Once an input `N` was given by the user and the `generate` button was clicked, Calculate the value of pi for different values of `N` number of uniform random points were generated using the `generate_uniform_numbers` method above. A for loop then runs through every point and calculates the number of points that are within the circle. This is done using the formula

$$distance = \sqrt{(x-0.5)^2 + (y-0.5)^2}$$

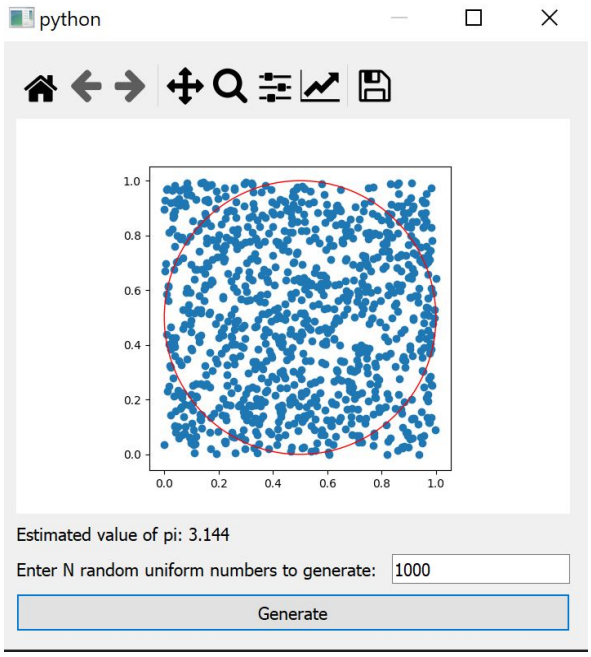
If the distance is less than or equal to 0.5 (the radius of the circle), then the point is deemed to be within the circle. The numbers 0.5 were used in the distance formula because the center of the graphical zero is at (0.5, 0.5) on the coordinate scale as seen in the image below.

The estimated value of pi is then calculated using

```
estimated_value = 4 * (number_points_inside / len(uniform_numbers))
```

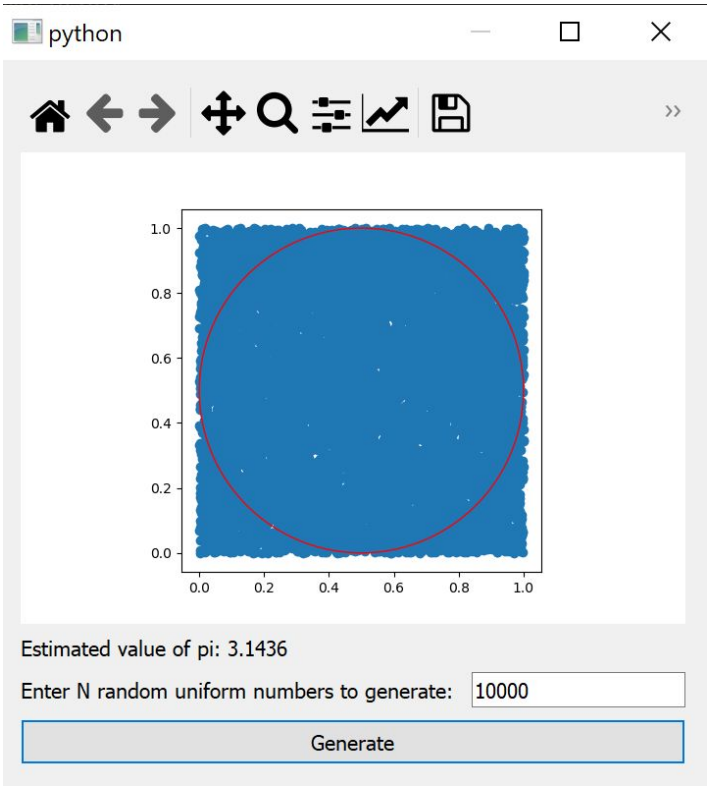
This formula stemmed from the fact that the bounding square had a length of 1 for a total area of 1*1=1. The circle inside the square had a radius of 0.5 for an area of π*(0.5)^2 = π/4. Dividing the area of the circle by the area of the square thus yielded $\frac{\pi}{4} / 1 = \frac{\pi}{4}$. The estimated value of pi is thus approximately equal to four times the number of points lying within the circle divided by the total number of points, which is found within the length of the generated length of uniform numbers.

3. Display visually the effect of N on the results.
The PyQt GUI was used with an embedded Matplotlib graph for graphing the points inside the bounding square and circle. The layout of the GUI can be seen below:



The simulation with 1000 random uniform points

The GUI allowed the user to input any valid number of uniform numbers to generate. The results of generating those points could be triggered by pressing the Generate button, and the estimated value of pi would dynamically update every time the button was pushed. Another example with 10000 generated points is seen below.



The simulation with 10,000 random uniform points

The GUI allowed for a toolbar that could zoom in on the dots, filter out the settings, save images, and go back and forth between generated iterations of the points.

4. Comment on what you learned from his exercise.
This exercise helped me realize how more data points can often yield better results. For instance, only 10 generated data points often will yield a number further away from the exact number of pi, such as 4.0 or 2.8 and only ever occasionally estimate a closer value of 3.2. I noticed that there is a general tend toward a more accurate estimation of pi with an increasing amount of data points, with 100 data points having an average percent error closer to 8 percent, 1000 data points having a percent error closer to 0.8%, and 10000 data points having an average percent error closer to 0.015%.

After continually trying more and more data points, the estimation of pi would only get increasingly accurate, so I did not notice any concept of overfitting using the Monte Carlo method like I have experienced with machine learning before. I thus learned the power of having more data, even if it's just randomly generated uniform data.

This exercise also taught me some precautions about any biases in code that can yield the results I was looking for in the first place. Because of some of the equations I used, such as (4 * the number of points inside the circle / the total number of points) proved that it can be easily possible to manipulate even random uniform numbers to produce whatever number I desire with

a simple equation and some basic mathematics. This illustrated to me the power of computer programs' ability to manipulate and determine the output of data.

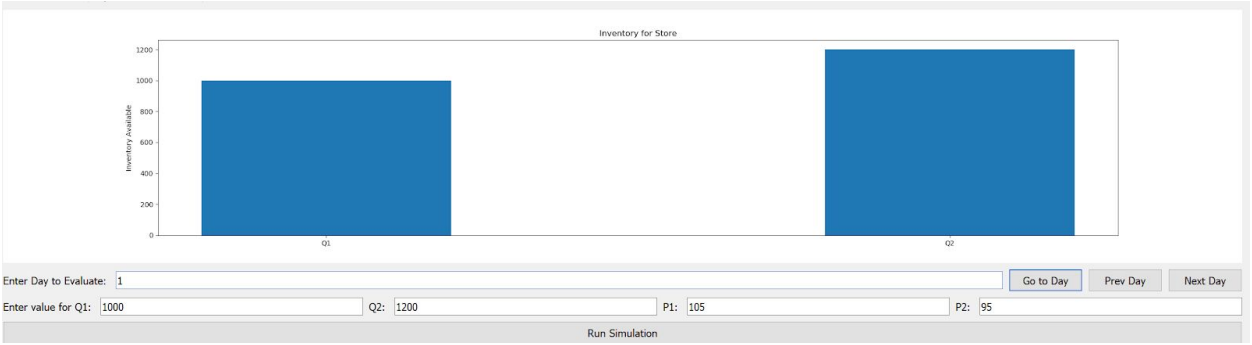
I also learned the importance of picking the specific kind of probability distributions in this experiment. If, for instance, a normal or pareto distribution were to be used rather than the uniform distribution used in this experiment, the estimated value of pi would likely be far less accurate, for the numbers wouldn't be evenly distributed inside and outside the circle. This proved an important lesson and picking the correct input and data points for all the experiments I run in the future.

Practicing with this Monte Carlo simulation helped to illustrate its potential uses in calculating potential risk and in forecasting models. It's obvious that, with the right data distributions, this simulation could be useful in the fields of finance (stocks simulations), business, science (epidemic simulations), engineering (telecoms networking simulations), and much more.

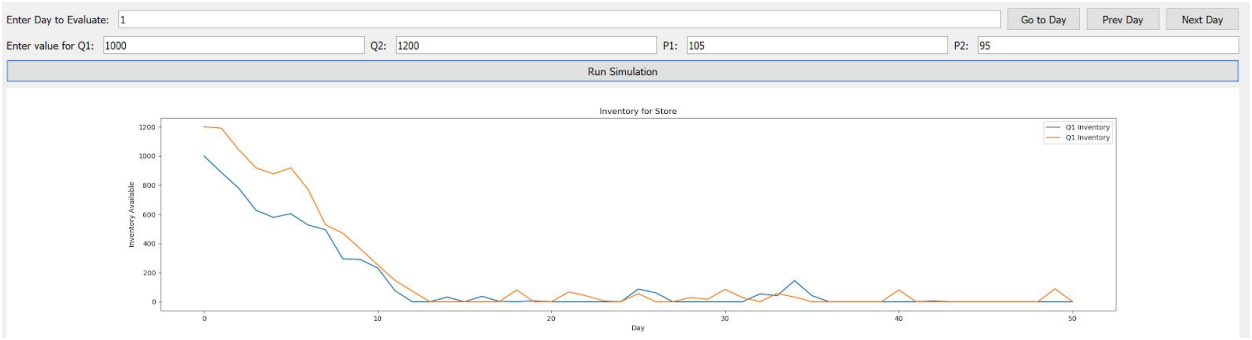
Problem 2:

1. Simulate the problem as described with a computer program.
PyQt, a cross-platform Python version of the GUI toolkit Qt, was used along with the graphing functions of Matplotlib and the mathematics of NumPy to graph, simulate, and visualize the problem as described in the prompt.

To begin, each input variable is stored in Qt's QLineEdit widget, which allows the user to dynamically simulate different variables and results on the fly without changing any code. There are two visual components to the GUI: One at the top which allows the user to visualize the inventories Q1 and Q1 for any particular day in a bar graph format, like so:



And a bottom visual component at the bottom which allows the user to visualize the results over time, like so:



This allowed the user to simultaneously visualize the results of the input inventory selections overall and at once.

The main portion of the simulation code excluding the GUI sections is as follows:

```
# START Run all of simulation in this block
# Create array of days up to 50 and then display results
# Initial inventory
Q1_inventory_remaining=self.Q1_inventory
Q2_inventory_remaining=self.Q2_inventory
number_of_days_to_run_simulation=50

# Create list of inventory for every day
inventories=[[Q1_inventory_remaining, Q2_inventory_remaining]]
# Run simulation for the input number of dats to run the simulation
for i in range(number_of_days_to_run_simulation):
    if self.P1_purchase_request_wait_time == 0:
        Q1_inventory_remaining=Q1_inventory_remaining + self.P1_request
        self.P1_purchase_request_wait_time=random.randint(1, 4)
    else:
        self.P1_purchase_request_wait_time=self.P1_purchase_request_wait_time - 1

    if self.P2_purchase_request_wait_time == 0:
        Q2_inventory_remaining=Q2_inventory_remaining + self.P2_request
        self.P2_purchase_request_wait_time=random.randint(1, 4)
    else:
        self.P2_purchase_request_wait_time=self.P2_purchase_request_wait_time - 1

    # Every day, have customers purchase a certain amount of inventory from business
    # Assume customers can purchase up
    # Mean value of days of inventory bought of the original stock value
    mean_days_of_stock_purchased_by_customers=90
    # mean and standard deviation, respectively
    mu, sigma=self.Q1_inventory/mean_days_of_stock_purchased_by_customers, 100.5
    distribution_set=np.random.normal(mu, sigma, 1000)

    # Ensure randomly chosen number is not negative (very unlikely given the
    # Gaussian distribution, but still possible)
```

```
            i=0
            normal_random_q1=distribution_set[i]
            while normal_random_q1 < 1:
                i=i + 1
                normal_random_q1=distribution_set[i]

            # Ensure independency of values of Q1 and Q2 by making sure they use two separate distributions

            # Mean value of days of inventory bought of the original stock value
            mean_days_of_stock_purchased_by_customers_q2=90
            # mean and standard deviation, respectively
            mu, sigma=self.Q2_inventory/mean_days_of_stock_purchased_by_customers_q2, 100.5
            distribution_set_q2=np.random.normal(mu, sigma, 1000)

            i=0
            normal_random_q2=distribution_set_q2[i]
            while normal_random_q2 < 1:
                i=i + 1
                normal_random_q2=distribution_set_q2[i]

            # Decrement respective amounts of stock according to these two independent Gaussian distribution
selections
            Q1_inventory_remaining=Q1_inventory_remaining - \
                math.ceil(normal_random_q1)
            Q2_inventory_remaining=Q2_inventory_remaining - \
                math.ceil(normal_random_q2)

            # Ensure quantities do not go into the negatives
            if Q1_inventory_remaining < 0:
                Q1_inventory_remaining=0

            if Q2_inventory_remaining < 0:
                Q2_inventory_remaining=0

            inventories.append(
                [Q1_inventory_remaining, Q2_inventory_remaining])
        # END Run of all simulation
```

It can be seen here that the simulation begins by first assigning an initial value to the Q_1 inventory and Q_2 inventories as input by the GUI and assuming that the simulation should run for approximately 50 days. The actual inventories of every day are stored in the `inventories` list, which stores a list of tuples containing the values for Q_1 and Q_2.

A for loop then runs through for every day as specified by `number_of_days_to_run_simulation`. Inside this loop, it checks the randomly assigned (between 1 and 4 per a uniform distribution) values of `P1_purchase_request_wait_time` and `P2_purchase_request_wait_time`. If either are at 0, then the values of those purchase requests are considered to be completed and their values are added to the total stock found in Q_1 and Q_2, respectively; their times are then uniformly and

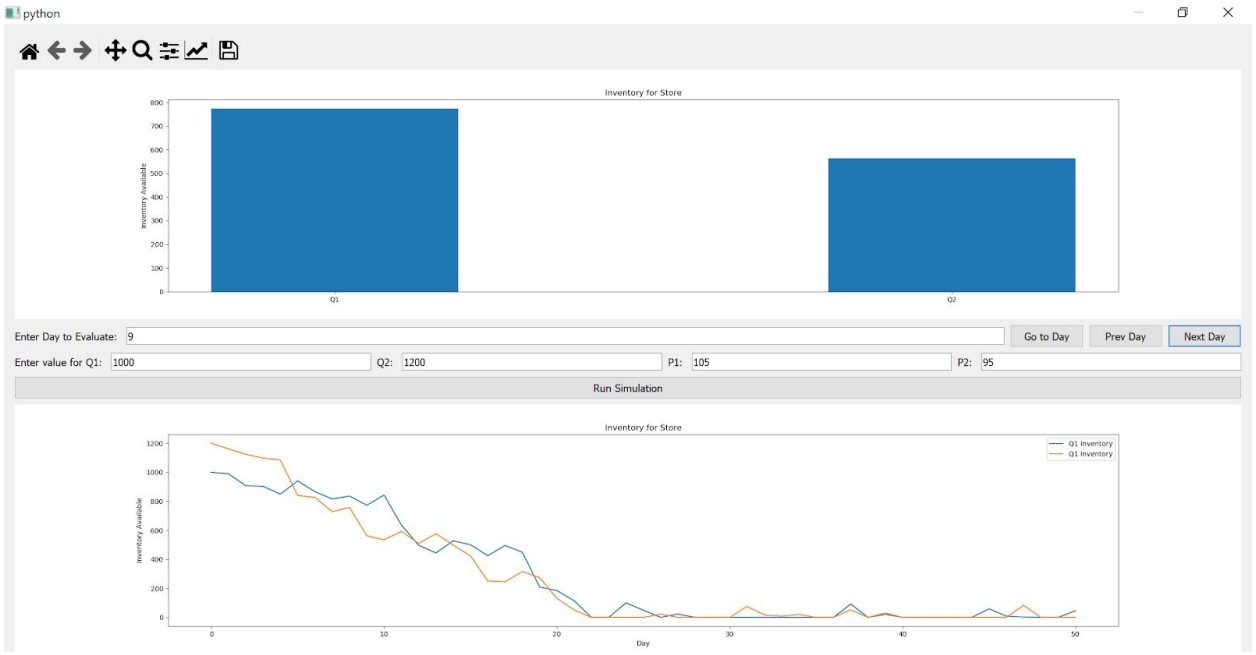
randomly assigned to another value between the integers found in 1 and 4, inclusive. If, on the other hand, their values are not at 0, then their values for the day are decremented and the for loop continues.

The purchase of the inventory by customers is then simulated using a Gaussian distribution with a mean assigned to be the value of the inventory divided by a number chosen by the user based on how many days of stock the company expects to keep on hand for customers to purchase as specified by `mean_days_of_stock_purchased_by_customers`. A standard deviation for the normal distribution is then set to be 100.5 to allow for a wide variety of selection by customers to simulate how customers may be more likely to purchase goods on weekends or on holidays, for example. Two independent Gaussian distributions are created and selected from each time so that the values of the inventories in Q_1 and Q_2 can run independently and simultaneously. The values selected by the Gaussian distributions are then subtracted from their respective inventories of Q_1 and Q_2, ensuring that the quantities can never go below zero if that situation does happen to occur. The for loop continues to run the simulation until all the days are expired.

2. Make some assumptions for different values of Q1, Q2, P1, and P2 and run your simulation collecting meaningful results.

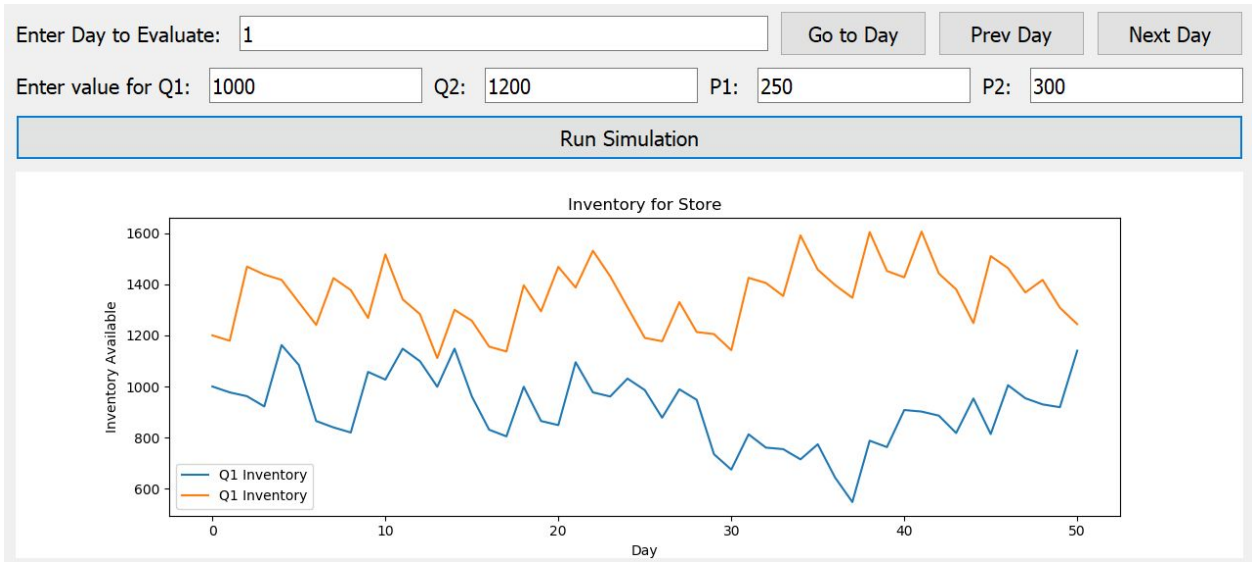
Many different iterations and summary results were uncovered and are displayed in part 3 below.

3. Display your summary of results visually or in a tabular fashion.
The results of running the simulation were visualized using a GUI for each simulation run, for example:



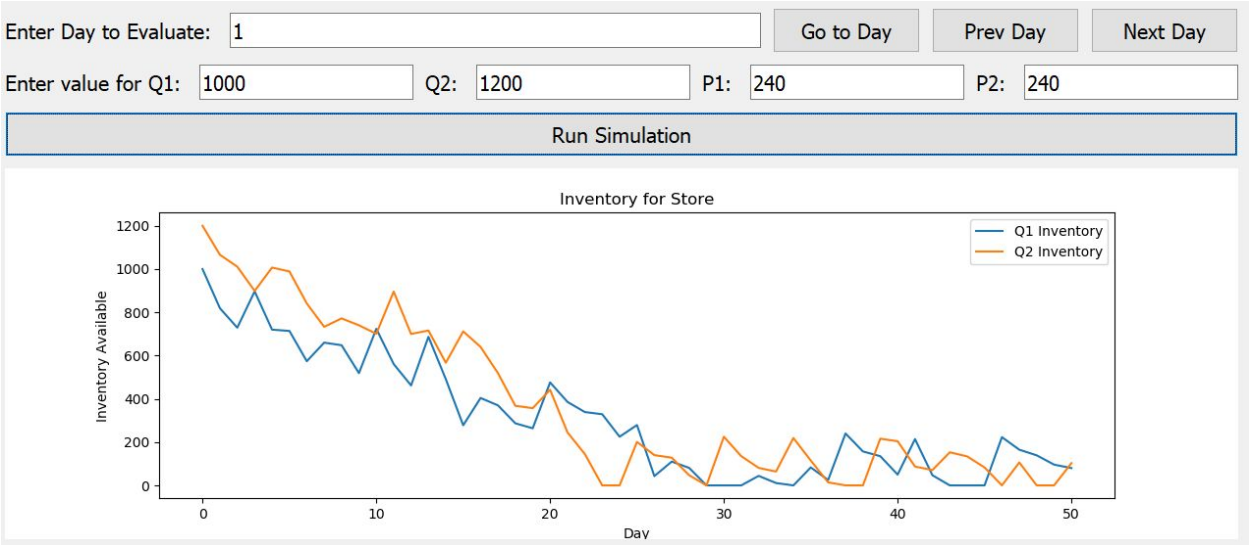
The above GUI illustrates the results of running the simulation each time and how the general downward trend of inventory is found if the purchase requests for the company are not high enough when compared to the customers' purchases of the inventory.

An important find by running this simulation is that, although the problem states that $Q_1 >> P_1$, a value of $P_1 = 0.25 * Q_1$ provided a more optimal result of keeping inventory in stock without running out or gathering too much inventory at once. The results can be illustrated by the general flat trend over 50 days as seen here:

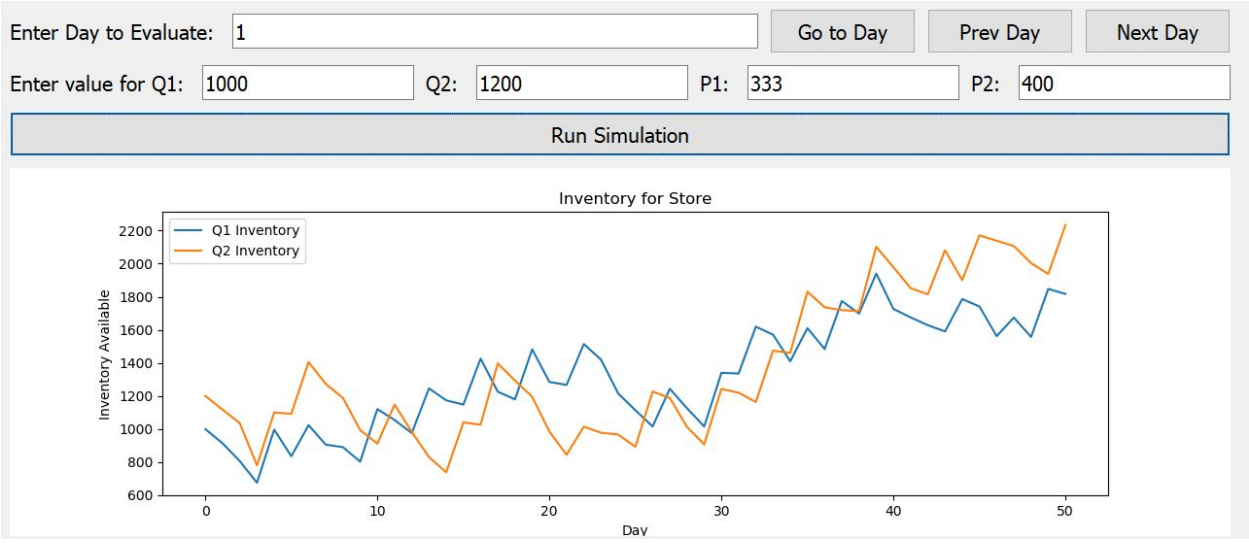


To be sure, this function relies on the fact that the customers are constantly trying to purchase a normal distribution amount of goods with a mean centered around the total original value of Q_1 and Q_2 divided by 90, which could vary greatly depending on the size of the store.

To contrast, as soon as the store begins to change it's uniformly distributed purchase requests to become as low as one-fifth that of the original inventory stocks, the downward trend of all the inventories tends toward zero after a mere 50 days, as seen below:



If, on the other hand, the values of the purchase requests are always set to be one third that of the original Q_1 and Q_2 inventories every day, then the inventory of the stock will soon skyrocket even over a mere 25 days, often doubling its values on an average of simply 39 days, like so:



This shows how a P_1/P_2 value that is relatively too high or too low can quickly mean that a store can either run out of stock or can run out of room for places to put stock in their store.

The results of this simulation thus illustrate that, given the assumptions in the program, the "Goldilocks zone" of purchase requests should be approximately one-fourth that of the original inventories if the store owner would like to constantly have around that stock in the store and not run out of stock nor have too much unwanted stock at any particular time.

4. Comment on what you observed and if you would have any recommendation for the store owner.

Based on the assumptions that a store estimates that its customers would purchase roughly one-thirtieth to one-fiftieth of its overall stock every day with a standard deviation of 100.5 in a Gaussian distribution and that purchase requests are sent every 1 to 4 days uniformly, I would recommend the store owner to have its purchase request be roughly ¼ of the original Q_1/Q_2 stock value.

When the purchase requests go over ¼ the original value (e.g. ⅓ or ½, the store begins to gain inventory far too quickly and can unintentionally double its stock over the period of roughly 39 days on average. This could lead to too much inventory in the store and there would likely not be enough space in the store for inventory to be placed.

When the purchase requests go under ¼ of the original value, on the other hand (e.g. ⅕ or ⅙), the stores quickly begin to run out of stock after an average of only 25 days. This would prevent the store from being able to sell inventory that had been selling well previously.

Thus, after running this simulation with the given assumed variables, I would recommend the store owner to write its purchase requests to be roughly one-fourth of the original inventory quantities in order to maintain a constant inventory for sales and to not take up too much store space.

If more time were to be given to this project, research and exploration would have been given to the manipulation of more variables that were assumed in this project, such as the average purchased stock by customers every day, the standard deviation of the Gaussian distribution modeling the customer purchases, the amount of stock the stores wish to have in inventory relative to the customer's desires, and the range of purchase request dates other than in the range of 1 to 4.

Manipulating these numbers could have provided an even more accurate recommendation for the store owner that could be variable-based on several factors that are unique to the store owner's case and changes. This would have been more useful for the store owner's future decision-making in improving his inventory purchases and stock holdings. Monetary value could

also have been assigned to the purchase requests, purchases of the customers, and unique values could have been assigned to each Q_1 and Q_2. This could have led to different recommendations based on money rather than simply amount of inventory.

Problem 3:

Write a paragraph as self-reflection on Assignment 1 programming problem on how you would improve the simulation if you had much more time.

If more time were to be invested into the 3D space simulation program from Assignment 1, more variables would have been considered. Most notably, some obstacles and a limited arena may be useful and influential in the results of this situation, for those are practical obstacles found in the real world rather than the infinite universe assumed in this model. Variable speed may also prove to provide more swift and skilled maneuvers for the ships and may prove to illustrate that avoiding the opposite ship should be a more utilized option than the assumed constant speed that was used in this model. In addition, more in-depth strategies and strategic models could have been tested rather than the random-based strategies used in the model (although the models were random, the input parameters still heavily influenced the ship behaviors, which allowed for the mocking of specific strategies, however limited.) To be sure, the simulated behavior here has its limits due to its dearth of possible variables, but it still remains a useful model for more basic primitive strategies (which, ironically, probably wouldn't be used by futuristic species of animals that are intelligent enough to operate these advanced spacecraft.)

Appendix

Source code for estimating pi using the Monte Carlo method:

calculating_pi.py:

```
# Aaron Fox
# CECS 622-01
# Dr. Adel Elmaghraby
# Assignment 2 Problem 1

import sys # For exiting program properly
from PyQt5.QtWidgets import QDialog, QApplication, QPushButton, QVBoxLayout, QLineEdit, QLabel,
QHBoxLayout # for GUI
from PyQt5.QtCore import Qt  # To enable maximizing of the GUI

# Use Matplotlib for the embedded graphing display of the dots inside the circle/square
from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as FigureCanvas
from matplotlib.backends.backend_qt5agg import NavigationToolbar2QT as NavigationToolbar
import matplotlib.pyplot as plt
import matplotlib.patches as patches  # For bounding square
```

```
import random # For its random.uniform function
import math # For using square root to find distance points are from circle center

# All of the gui is based out of this Window class using Python PyQt5 GUI framework
class Window(QDialog):
    def __init__(self, parent=None):
        super(Window, self).__init__(parent)
        # Allow for easy maximizing of the GUI
        self.setWindowFlag(Qt.WindowMaximizeButtonHint, True)

        # For keeping track of points that fall inside circle
        self.number_of_points_within_cirle = 0
        # For keeping track of poitns that fall outside the circle
        self.number_of_points_outside_cirle = 0

        # Figure instance for plotting
        self.figure = plt.figure()

        # self.canvas is the widget that displays the figure,
        # taking the figure instance as a parameter to __init__
        self.canvas = FigureCanvas(self.figure)

        # Adds toolbar to top of GUI for saving pictures, zooming in graph, etc.
        self.toolbar = NavigationToolbar(self.canvas, self)

        # Add input box for user to request N uniform numbers
        self.random_number_input = QLineEdit()
        self.random_number_input_label = QLabel()
        self.random_number_input_label.setText(
            "Enter N random uniform numbers to generate: ")

        # Button connected to generation of points on figure to visually
        # display the uniform random distribution of points inside/outside the circle
        self.button = QPushButton('Generate')
        self.button.clicked.connect(self.generate)

        # set the layout (vertical)
        layout = QVBoxLayout()
        layout.addWidget(self.toolbar)
        layout.addWidget(self.canvas)

        # Estimate of pi value
        self.pi_estimate_text = QLabel()
        self.pi_estimate_text.setText("Estimated value of pi: ")
        layout.addWidget(self.pi_estimate_text)

        # Number of N numbers to generate input
        horizontal_layout = QHBoxLayout()
        horizontal_layout.addWidget(self.random_number_input_label)
```

```
        horizontal_layout.addWidget(self.random_number_input)
        layout.addLayout(horizontal_layout)

        # Add Generate button to layout
        layout.addWidget(self.button)

        self.setLayout(layout)

        # Create initial empty plot for aesthetic purposes
        self.initial_plot()

    # Initial plot to make the GUI look initially pretty for aesthetic purposes
    def initial_plot(self):
        self.figure.clear()

        # create an axis
        ax = self.figure.add_subplot(1, 1, 1)
        plt.gca().set_aspect('equal', adjustable='box')

        # Create a circle patch
        circle = patches.Circle(xy=(.5, .5), radius=.5,
                                linewidth=1, edgecolor='r', facecolor='none')

        # Add the patch to the Axes
        ax.add_patch(circle)

        # refresh canvas
        self.canvas.draw()

    # Generates and displays the circle, graph, and the uniform random points on the graph
    def generate(self):
        # Clear the figure in case stuff is on it already
        self.figure.clear()

        # create an axis
        ax = self.figure.add_subplot(1, 1, 1)

        # Make the aspect ratio equal and adjustable so that the circle looks like a proper circle
        plt.gca().set_aspect('equal', adjustable='box')

        # Create a circle
        circle = patches.Circle(xy=(.5, .5), radius=.5,
                                linewidth=1, edgecolor='r', facecolor='none')

        # Add the patch to the Axes
        ax.add_patch(circle)

        # Get input N (amount of numbers to generate) from input
        # Make sure the input is a number and is not 0 (to prevent divide by 0 error)
        if self.random_number_input.text().isnumeric() and int(self.random_number_input.text()) != 0:
```

```
        N = int(self.random_number_input.text())

        # Generate N random numbers
        uniform_numbers = self.generate_uniform_numbers(N)

        ### Plot all uniform number pairs ###
        # Create lists of each x and y pair to plot them on a scatter plot
        uniform_number_x = []
        uniform_number_y = []

        # Keep track of the number of points inside circle
        number_points_inside = 0
        for i in range(len(uniform_numbers)):
            uniform_number_x.append(uniform_numbers[i][0])
            uniform_number_y.append(uniform_numbers[i][1])
            # If the distance of the point from the circle is less than or equal to 0.5,
            # then that point falls within the circle
            # Accoount for the fact that the center of the circle is at (0.5, 0.5), so distance
            # formula for distance from center is sqrt((x - 0.5)^2 + (y - 0.5)^2)
            if math.sqrt((uniform_numbers[i][0] - 0.5)** 2 + (uniform_numbers[i][1] - 0.5) ** 2) <= 0.5:
                number_points_inside = number_points_inside + 1

        print("number_points_inside == " + str(number_points_inside))
        print("total_points == " + str(len(uniform_numbers)))

        # Plot all points found in uniform numbers
        plt.scatter(uniform_number_x, uniform_number_y)

        # Estimated value
        # Multiply by four based on dividing area of circle (pi/4) by area of square (1), which
        # means the estimated value should be around pi/4 / 1 = pi/4. Thus, multiplying result by
        # 4 means that it should yield an estimated value of pi
        estimated_value = 4 * (number_points_inside / len(uniform_numbers))
        # Update estimated pi value label
        self.pi_estimate_text.setText("Estimated value of pi: " + str(estimated_value))

        # refresh canvas
        self.canvas.draw()
    else:
        print("Input does not contain a valid number")

# Generates N uniform random numbers
def generate_uniform_numbers(self, N):
    uniform_numbers = []
    for i in range(0, N):
        uniform_numbers.append(
            [random.uniform(0, 1), random.uniform(0, 1)])
    return uniform_numbers
```

```
if __name__ == '__main__':
    app = QApplication(sys.argv)

    main = Window()
    main.show()

    sys.exit(app.exec_())
```

Source Code for inventory_modeling.py:

```
# Aaron Fox
# CECS 622-01
# Dr. Adel Elmaghraby
# Assignment 2 - Problem 2

import sys  # For exiting program properly
from PyQt5.QtWidgets import QDialog, QApplication, QPushButton, QVBoxLayout, QLineEdit, QLabel,
QHBoxLayout  # for GUI
from PyQt5.QtCore import Qt  # To enable maximizing of the GUI

# Use Matplotlib for the embedded graphing display of the dots inside the circle/square
from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as FigureCanvas
from matplotlib.backends.backend_qt5agg import NavigationToolbar2QT as NavigationToolbar
import matplotlib.pyplot as plt
import matplotlib.patches as patches  # For bounding square

import random  # For its random.uniform function
import math  # For using square root to find distance points are from circle center
import numpy as np  # For the use of a Gaussian distribution of the customer purchases

# All of the gui is based out of this Window class using Python PyQt5 GUI framework
class Window(QDialog):
    def __init__(self, parent=None):
        super(Window, self).__init__(parent)
        # Allow for easy maximizing of the GUI
        self.setWindowFlag(Qt.WindowMaximizeButtonHint, True)

        # Current day of simulation
        self.current_day = 0

        # Figure instance for plotting
        self.figure = plt.figure()

        # self.canvas is the widget that displays the figure,
        # taking the figure instance as a parameter to __init__
        self.canvas = FigureCanvas(self.figure)

        # Adds toolbar to top of GUI for saving pictures, zooming in graph, etc.
        self.toolbar = NavigationToolbar(self.canvas, self)
```

```
# Add input boxes for user to input values
# Q1 and Q2 represent initial inventory quantities
self.Q1 = QLineEdit()
self.Q1.setText("1000")
self.Q1_label = QLabel()
self.Q1_label.setText("Enter value for Q1: ")

# Q2 Input and Label
self.Q2 = QLineEdit()
self.Q2.setText("1200")
self.Q2_label = QLabel()
self.Q2_label.setText("Q2: ")

# P1 and P2 are the amounts being deducted from Q1 and Q2 respectively
# every 1 to 4 days. NOTE: P1 << Q1 and P2 << Q2
# P1 Input and Label
self.P1 = QLineEdit()
self.P1.setText("105")
self.P1_label = QLabel()
self.P1_label.setText("P1: ")

# P2 Input and Label
self.P2 = QLineEdit()
self.P2.setText("95")
self.P2_label = QLabel()
self.P2_label.setText("P2: ")

# Button connected to generation of points on figure to visually
# display the uniform random distribution of points inside/outside the circle
self.button = QPushButton('Run Simulation')
self.button.clicked.connect(self.run_simulation)

# set the layout (vertical)
layout = QVBoxLayout()
layout.addWidget(self.toolbar)
layout.addWidget(self.canvas)

# For evaluating data in bar chart
self.day_to_evaluate_value = QLineEdit()
self.day_to_evaluate_value.setText("1")
self.day_to_evaluate_label = QLabel()
self.day_to_evaluate_label.setText("Enter Day to Evaluate: ")

self.go_to_day_button = QPushButton("Go to Day")
self.go_to_day_button.clicked.connect(self.go_to_day_button_clicked)

self.next_day_button = QPushButton("Next Day")
self.next_day_button.clicked.connect(self.next_day_clicked)

self.prev_day_button = QPushButton("Prev Day")
```

```
self.prev_day_button.clicked.connect(self.prev_day_clicked)

horizontal_layout1 = QHBoxLayout()
horizontal_layout1.addWidget(self.day_to_evaluate_label)
horizontal_layout1.addWidget(self.day_to_evaluate_value)
horizontal_layout1.addWidget(self.go_to_day_button)
horizontal_layout1.addWidget(self.prev_day_button)
horizontal_layout1.addWidget(self.next_day_button)
layout.addLayout(horizontal_layout1)

# END bar chart GUI manipulation buttons

horizontal_layout = QHBoxLayout()
horizontal_layout.addWidget(self.Q1_label)
horizontal_layout.addWidget(self.Q1)
horizontal_layout.addWidget(self.Q2_label)
horizontal_layout.addWidget(self.Q2)
horizontal_layout.addWidget(self.P1_label)
horizontal_layout.addWidget(self.P1)
horizontal_layout.addWidget(self.P2_label)
horizontal_layout.addWidget(self.P2)
layout.addLayout(horizontal_layout)

# Add Generate button to layout
layout.addWidget(self.button)

self.setLayout(layout)

# Keep track of values of Q1 and Q2 inventories here
self.Q1_inventory = int(self.Q1.text())
self.Q2_inventory = int(self.Q2.text())
self.P1_request = int(self.P1.text())
self.P2_request = int(self.P2.text())

# random.randint generate uniform distribution of integers between 1 and 4, inclusive of both
self.P1_purchase_request_wait_time = random.randint(1, 4)
self.P2_purchase_request_wait_time = random.randint(1, 4)

# Place all simulation results here
self.simulation_inventory_results = []

# Create initial empty bar plot for aesthetic purposes
self.initial_plot()

# Second Figure for line plot
# Figure instance for plotting
self.figure2 = plt.figure()

# self.canvas is the widget that displays the figure,
# taking the figure instance as a parameter to __init__
```

```
        self.canvas2 = FigureCanvas(self.figure2)
        layout.addWidget(self.canvas2)
        self.lineplot_init()
        # self.next_day_clicked()
        # self.next_day_clicked()

    # Go to day as specified by user
    def go_to_day_button_clicked(self):
        if not self.simulation_inventory_results:
            return
        self.figure.clear()

        # create an axis
        ax = self.figure.add_subplot(1, 1, 1)

        self.current_day = int(self.day_to_evaluate_value.text())
        # self.day_to_evaluate_value.setText(str(self.current_day))

        labels = ['Q1', 'Q2']
        label_locs = [1, 2]
        width = .40
        ax.bar(label_locs, [self.simulation_inventory_results[self.current_day][0],
                            self.simulation_inventory_results[self.current_day][1]], width, label='Q1')

        # print('test')
        # Set Graph labels, title, x-axis
        ax.set_ylabel('Inventory Available')
        ax.set_title('Inventory for Store')
        ax.set_xticks(label_locs)
        ax.set_xticklabels(labels)

        # refresh canvas
        self.canvas.draw()

    # Plot next day on bar graph
    def next_day_clicked(self):
        # Make sure that there are simulation results to graph
        if not self.simulation_inventory_results:
            return
        self.figure.clear()

        # create an axis
        ax = self.figure.add_subplot(1, 1, 1)

        self.current_day = self.current_day + 1
        self.day_to_evaluate_value.setText(str(self.current_day))

        labels = ['Q1', 'Q2']
        label_locs = [1, 2]
        width = .40
```

```
        ax.bar(label_locs, [self.simulation_inventory_results[self.current_day][0],
self.simulation_inventory_results[self.current_day][1]], width, label='Q1')

        # Set Graph labels, title, x-axis
        ax.set_ylabel('Inventory Available')
        ax.set_title('Inventory for Store')
        ax.set_xticks(label_locs)
        ax.set_xticklabels(labels)

        # refresh canvas
        self.canvas.draw()

    # Plot previous day on bar graph
    def prev_day_clicked(self):
        # Ensure there are results to graph
        if not self.simulation_inventory_results:
            return
        self.figure.clear()

        # create an axis
        ax = self.figure.add_subplot(1, 1, 1)

        self.current_day = self.current_day - 1
        self.day_to_evaluate_value.setText(str(self.current_day))

        labels = ['Q1', 'Q2']
        label_locs = [1, 2]
        width = .40
        ax.bar(label_locs, [self.simulation_inventory_results[self.current_day][0],
                            self.simulation_inventory_results[self.current_day][1]], width, label='Q1')

        # Set Graph labels, title, x-axis
        ax.set_ylabel('Inventory Available')
        ax.set_title('Inventory for Store')
        ax.set_xticks(label_locs)
        ax.set_xticklabels(labels)

        # refresh canvas
        self.canvas.draw()

    # Initial Line plot to make GUI look initially pretty for aesthetic purpises
    def lineplot_init(self):
        self.figure2.clear()
        # create an axis
        ax=self.figure2.add_subplot(1, 1, 1)

        # Set Graph labels, title, x-axis
        ax.set_ylabel('Inventory Available')
        ax.set_xlabel('Day')
        ax.set_title('Inventory for Store')
```

```
    self.canvas2.draw()


# Plot the line graph illustrating changes in each inventory
def plot_line_graph(self):
    self.figure2.clear()
    # create an axis
    ax=self.figure2.add_subplot(1, 1, 1)

    # Set Graph labels, title, x-axis
    ax.set_ylabel('Inventory Available')
    ax.set_xlabel('Day')
    ax.set_title('Inventory for Store')

    Q1_nums=[]
    Q2_nums=[]
    for i in range(len(self.simulation_inventory_results)):
        Q1_nums.append(self.simulation_inventory_results[i][0])
        Q2_nums.append(self.simulation_inventory_results[i][1])

    ax.plot(range(len(Q1_nums)), Q1_nums, label='Q1 Inventory')
    ax.plot(range(len(Q2_nums)), Q2_nums, label='Q2 Inventory')
    ax.legend()

    # refresh canvas
    self.canvas2.draw()


# Initial plot to make the GUI look initially pretty for aesthetic purposes
def initial_plot(self):
    self.figure.clear()

    # create an axis
    ax=self.figure.add_subplot(1, 1, 1)
    # plt.gca().set_aspect('equal', adjustable='box')

    labels=['Q1', 'Q2']
    label_locs=[1, 2]
    width=.40
    ax.bar(label_locs, [self.Q1_inventory,
          self.Q2_inventory], width, label='Q1')

    # Set Graph labels, title, x-axis
    ax.set_ylabel('Inventory Available')
    ax.set_title('Inventory for Store')
    ax.set_xticks(label_locs)
    ax.set_xticklabels(labels)

    # refresh canvas
    self.canvas.draw()
```

```
# Runs simulation to on inventories based on input purchase request and inventory quantity amounts
def run_simulation(self):

    # START Run all of simulation in this block
    # Create array of days up to 50 and then display results
    # Initial inventory
    Q1_inventory_remaining=self.Q1_inventory
    Q2_inventory_remaining=self.Q2_inventory
    number_of_days_to_run_simulation=50

    # Create list of inventory for every day
    inventories=[[Q1_inventory_remaining, Q2_inventory_remaining]]
    # Run simulation for the input number of dats to run the simulation
    for i in range(number_of_days_to_run_simulation):
        if self.P1_purchase_request_wait_time == 0:
            Q1_inventory_remaining=Q1_inventory_remaining + self.P1_request
            self.P1_purchase_request_wait_time=random.randint(1, 4)
        else:
            self.P1_purchase_request_wait_time=self.P1_purchase_request_wait_time - 1

        if self.P2_purchase_request_wait_time == 0:
            Q2_inventory_remaining=Q2_inventory_remaining + self.P2_request
            self.P2_purchase_request_wait_time=random.randint(1, 4)
        else:
            self.P2_purchase_request_wait_time=self.P2_purchase_request_wait_time - 1

        # Every day, have customers purchase a certain amount of inventory from business
        # Assume customers can purchase up
        # Mean value of days of inventory bought of the original stock value
        mean_days_of_stock_purchased_by_customers=90
        # mean and standard deviation, respectively
        mu, sigma=self.Q1_inventory/mean_days_of_stock_purchased_by_customers, 100.5
        distribution_set=np.random.normal(mu, sigma, 1000)

        # Ensure randomly chosen number is not negative (very unlikely given the
        # Gaussian distribution, but still possible)
        i=0
        normal_random_q1=distribution_set[i]
        while normal_random_q1 < 1:
            i=i + 1
            normal_random_q1=distribution_set[i]

        # Ensure independency of values of Q1 and Q2 by making sure they use two separate distributions

        # Mean value of days of inventory bought of the original stock value
        mean_days_of_stock_purchased_by_customers_q2=90
        # mean and standard deviation, respectively
        mu, sigma=self.Q2_inventory/mean_days_of_stock_purchased_by_customers_q2, 100.5
        distribution_set_q2=np.random.normal(mu, sigma, 1000)
```

```
            i=0
            normal_random_q2=distribution_set_q2[i]
            while normal_random_q2 < 1:
                i=i + 1
                normal_random_q2=distribution_set_q2[i]

            # Decrement respective amounts of stock according to these two independent Gaussian distribution
selections
            Q1_inventory_remaining=Q1_inventory_remaining - \
                math.ceil(normal_random_q1)
            Q2_inventory_remaining=Q2_inventory_remaining - \
                math.ceil(normal_random_q2)

            # Ensure quantities do not go into the negatives
            if Q1_inventory_remaining < 0:
                Q1_inventory_remaining=0

            if Q2_inventory_remaining < 0:
                Q2_inventory_remaining=0

            inventories.append(
                [Q1_inventory_remaining, Q2_inventory_remaining])
        # END Run of all simulation
        self.simulation_inventory_results=inventories
        print(self.simulation_inventory_results)
        self.plot_line_graph()

        # Clear the figure in case stuff is on it already
        self.figure.clear()

        # create an axis
        ax=self.figure.add_subplot(1, 1, 1)

        # Get input N (amount of numbers to generate) from input
        # Make sure the input is a number and is not 0 (to prevent divide by 0 error)
        if (self.Q1.text().isnumeric() and int(self.Q1.text()) != 0
            and self.Q2.text().isnumeric() and int(self.Q2.text()) != 0
            and self.P1.text().isnumeric() and int(self.P1.text()) != 0
            and self.P2.text().isnumeric() and int(self.P2.text()) != 0):
            self.Q1_inventory=int(self.Q1.text())
            self.Q2_inventory=int(self.Q2.text())
            self.P1_request=int(self.P1.text())
            self.P2_request=int(self.P2.text())

            # Prep basics of bar graph
            labels=['Q1', 'Q2']
            label_locs=[1, 2]
            width=.40
            ax.bar(label_locs, [self.Q1_inventory,
```

```
                    self.Q2_inventory], width, label='Q1')

        # Set Graph labels, title, x-axis
        ax.set_ylabel('Inventory Available')
        ax.set_title('Inventory for Store')
        ax.set_xticks(label_locs)
        ax.set_xticklabels(labels)


        # refresh canvas
        self.canvas.draw()
      else:
        print("Inputs do not contain a valid number")

if __name__ == '__main__':
    app=QApplication(sys.argv)

    main=Window()
    main.show()

    sys.exit(app.exec_())
```

References

[1] “random - Generate pseudo-random numbers,” *random - Generate pseudo-random numbers - Python 3.8.1 documentation*. [Online]. Available: https://docs.python.org/3/library/random.html. [Accessed: 27-Jan-2020].

[2] “PyQt5 Reference Guide,” *PyQt5 Reference Guide - PyQt v5.14.0 Reference Guide*. [Online]. Available: https://www.riverbankcomputing.com/static/Docs/PyQt5/. [Accessed: 27-Jan-2020].

[3] “Grouped bar chart with labels,” *Grouped bar chart with labels - Matplotlib 3.1.2 documentation*. [Online]. Available: https://matplotlib.org/3.1.1/gallery/lines_bars_and_markers/barchart.html. [Accessed: 03-Feb-2020].