A SimPy User's Guide: Garment Factory Tutorial

Team Members: Megan F Grannan, Vi Vu Thoai Yim Koi Nguyen
Project Group 157
Project 11 - Learn a high-level simulation language

Abstract

In this project, we provide a tutorial for SimPy, a high-level, Python-based simulation language. Like Arena, SimPy is a process-interaction language designed for discrete event simulation. We use SimPy to simulate a garment factory, and walk readers through this example to demonstrate how the language works along with its main features. Specifically, we focus on manufacturing shirts from the time an order of pre-cut fabric arrives at the factory through final inspection and pickup. We cover key features of SimPy such as creating an environment, setting up simulation parameters, generating processes and resources of different distributions and capacities, connecting steps in the process, running the simulation, and reporting key statistics. We believe SimPy has many strengths, including its flexibility, ease of implementation, and cost. SimPy does have some weaknesses in comparison to Arena in the areas of declaring attributes, reporting and visualization. Overall, we recommend SimPy as a viable alternative for less-complex simulation models, where cost is a primary concern, or when multiple people need to collaborate to build the simulation.

Background and Description of Problem

We demonstrate SimPy through an example of a garment factory. After an order is placed, materials are purchased, they go through the serging process (process of overcasting the edges to prevent fabric from fraying), then each of the 3 parts of the shirt is sent to the appropriate sewing process. The parts are assembled together in the assembling process. The assembled shirts then go through the binding and washing process. Finally, they go through an inspection process where items that don't meet quality requirements are disposed of. The rest are batched and shipped off to our client. A visualization of this example is included below. The process of stock control was not included in this visualization for the sake of simplicity, however, we will include it in our code.

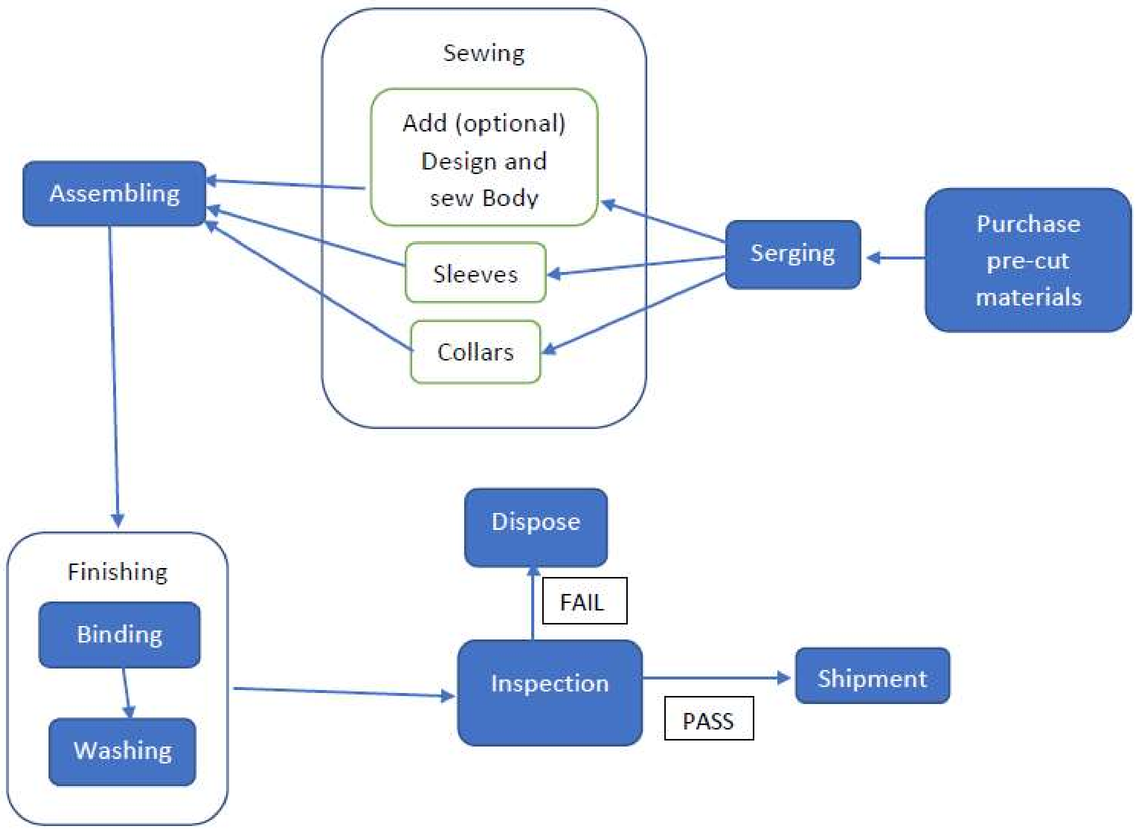


Figure 1: Garment factory simulation flowchart

We intentionally use an example of a shirt manufacturing system because we are familiar with the production process, but especially because in a real life example, the process differs based on the attributes of the entities. While learning about SimPy, one thing that stood out to us was the lack of methods to declare attributes for entities compared to Arena.

In the remainder of this report, we will take you through an overview of SimPy, walk through a basic simulation example, move to a full garment factory simulation example, and end with our evaluation of SimPy.

Main findings

SimPy Overview

SimPy is a free, open-source simulation package built in Python. SimPy has a process-interaction worldview and is best suited for discrete event simulation. SimPy uses basic principles of object-oriented programming in Python to implement simulations, including classes and generator functions.

Three building blocks are essential to understanding how SimPy works: *environment*, *processes*, and *events*. All SimPy simulations start by creating an *environment* - this is the sandbox in which all processes and events occur. Next, one can use generator functions to create *processes* within this environment. These processes have two purposes: to create *events* and to model what happens as entities pass through the system, experience events, and compete for resources. SimPy processes are defined, generated, then yield and wait to be triggered when an event occurs. Once an event is triggered, the process resumes; after the event, the process yields to the next event.

SimPy documentation is available at [1], which includes an overview of the package, instructions for downloading, and a number of helpful tutorials. We would also recommend the article “How to Use Generators and yield in Python” [2] to better understand concepts that are foundational for SimPy.

SimPy Tutorial Part 1: Simulating a Simple Garment Factory

Let’s start by illustrating a very simple garment factory with only one process to understand how SimPy works. Then, we’ll dive deeper into our full model.

```
# Step 1: Import libraries
import simpy
import numpy as np
import random

# Step 2: Define simulation parameters
materials_capacity = 100
SIM_TIME = 100

# Step 3: Create a Factory class
class Factory:
  def __init__(self, env):
    # Define containers to hold raw materials and finished shirts
    self.materials = simpy.Container(env, capacity = materials_capacity,
init = materials_capacity)
    self.finished_shirts = simpy.Container(env, capacity =
materials_capacity, init = 0)

# Step 4: Use functions to define the processes in the simulation
def shirt_mkr(env, garment_factory):
  while True:
    yield garment_factory.materials.get(1)
    shirt_making_time = np.random.triangular(0, 3, 10)
    yield env.timeout(shirt_making_time)
    yield garment_factory.finished_shirts.put(1)
```

```
# Step 5: Set up the SimPy environment
print('Garment Factory')
env = simpy.Environment()
garment_factory = Factory(env)

# Step 6: Generate the processes for the simulation
shirt_maker_gen = env.process(shirt_mkr(env, garment_factory))

# Step 7: Run the simulation and check its output
env.run(until=SIM_TIME)

print('Our factory made {}
shirts'.format(garment_factory.finished_shirts.level))
print('We still have materials to make {} more
shirts'.format(garment_factory.materials.level))
print('End of simulation')
---------------------------------------------------------------------------
Garment Factory
Our factory made 27 shirts
We still have materials to make 72 more shirts
End of simulation
```

In step 1, we import SimPy, along with random and numpy packages, which we will use to generate random variables with different distributions. In step 2, we define simulation parameters, including the total simulation time and also the capacity of any containers. Containers are useful for storing quantities of homogenous items - in our example, the raw materials to make shirts and then the finished shirts. When we move to our more advanced example, this is also where we will define any global variables and resource capacity. While one could define these parameters within the simulation itself, organizing them in a central place makes it easy to update the code and improves readability.

In step 3, we create the class Factory, where we can define any properties and methods of our factory. In this example, we use containers to describe the factory's capacity, currently defined as materials to make 100 shirts and capacity to store 100 finished shirts. Our factory starts off filled to its capacity of materials for 100 shirts, and with 0 completed shirts.

In step 4, we use functions to define more of our factory's processes. These processes are ones that require additional resources. In this simple example, we define one function - `shirt_mkr` - which accepts the environment and instance of the Factory class and makes shirts. We start with `while True` to indicate that this process is available for the full duration of the simulation. We then get materials to make one shirt from the `materials` container, define the shirt making time as a random variable, and use `env.timeout(shirt_making_time)` to simulate this process's time. Finally, we put the finished shirt in the `finished_shirts` container.

Notice that we use `.get()` and `.put()` to remove items from or add them to a container, and `yield` to yield each event until it is triggered.

By step 5, we now have all the processes and properties of the simulation defined and are almost ready to start the simulation. We set up the SimPy environment by creating an environment, `env`, and then creating an instance of Factory, `garment_factory`, within that environment. In step 6, we call the generator function `shirt_mkr` as a process within the environment. This step is what actually adds the process to SimPy's event queue so we can start making shirts. Finally, in step 7, we use `env.run()` to tell SimPy to run the simulation until the end time we have defined, and print a summary of the simulation.

SimPy Tutorial Part 2: Full Garment Factory Simulation

From the basic process model, we make edits to scale up our example and include all 7 processes. The full Python code can be found in the attached code file.

In step 1, for better visualization of the simulation's statistical output, we import pandas, scipy, and tabulate. In step 2, we increase the simulation time to 7 days to give the system state enough time to warm up for the continuous simulation. In addition, we define global variables and lists, where we will store the output data from the simulation including the time we received/completed an order, a record of each batch that went through quality checks, etc. It's a good idea to have the production capacity and factory resources as global variables so that we can easily change these values when we simulate the production process as the factory 'grows or scales down'. Resource capacity needs to be defined for different types of resources, and minimum values of resources are required to simulate the process of stock control. This code snippet demonstrates the above mentioned tasks.

```
# REPORTING VARIABLES -
# define variables to store simulation results
num_orders_completed = 0
order_in_lst = []
order_out_lst = []
# Change capacity as the factory 'grows'
min_order = 100
max_order = 1000
# RESOURCES
num_body_mkrs = 6
num_sleeve_mkrs = 3

min_stock = 70
```

In step 3, we update the class Factory. To model process interaction of the materials as they queue up to use the shared resources we use `simpy.Resource()` to control the use of shared machines and labor. To model the use of input material to make new products, we use `simpy.Container()` to control the flow of material and its output. To schedule an event that will execute when triggered we use `env.process()`, events like these allow the system to interact with processes 'outside' the model. Such as when stock level is low, an event is triggered to restock our `simpy.Container()`, as in the example below:

```
# serging process
self.serger = simpy.Resource(env, num_serging_mc)
self.post_serge_bodies = simpy.Container(env, capacity = bodies_capacity, init
= 0)

# stock control and dispatch
self.stock_ctrl = env.process(self.stock_ctrl(env))
self.dispatch_ctrl = env.process(self.dispatch_ctrl(env))
```

In step 4, we write functions to define the processes for our model. It's important to note that `env.process()` requires an input of a function that tells the system exactly what to do, said function is often written inside the class function. For example, the `stock_ctrl` function, inside class Factory, tells the system to trigger an event when the resource level of any input material has reached a certain level. The event will then generate a new order, calculate the amount of materials needed, and add new materials to the existing resource containers.

```
def stock_ctrl(self, env):
    global qty_making
    global order_in_lst
    global body_level
    global sleeves_level
    global collar_level
    yield env.timeout(0)
    while True:
      if self.bodies.level <= min_stock or self.collars.level <= min_stock
or self.sleeves.level <= min_stock:
        print("Stock below critical level at time {0}. Processing next
order.".format(env.now))
        yield env.timeout(random.randint(60, 120)) #  It takes X time to
process the new order
        order_qty = random.randint(min_order, max_order)
        order_sizes.append(order_qty)
        if self.current_order_qty == 0:
          self.current_order_qty = order_qty
        matl_qty = int(order_qty * 1.15) # material quantity will account
for items that fail quality control
        qty_making.append(matl_qty)
        #order_in = env.now
        order_in_lst.append(env.now)
        print("Accepting new order of {0} and sourcing material for {1}
shirts at time {2}.".format(order_qty, matl_qty, env.now))
        yield self.bodies.put(matl_qty *2)
        yield self.sleeves.put(matl_qty *2)
        yield self.collars.put(matl_qty *4)
        print("Piece stock levels: bodies - {0}, collars - {1}, sleeves -
{2}".format(self.bodies.level, self.collars.level, self.sleeves.level))
        body_level.append(self.bodies.level)
        sleeves_level.append(self.sleeves.level)
        collar_level.append(self.collars.level)
        yield env.timeout(60 * 2)
      else:
        yield env.timeout(60)
```

For other processes that interact within the model, the process function tends to be simpler. Let's step through the assembling function together as an example. The function first tells the system to get 1 collar piece, 1 body piece, and 2 sleeve pieces from the appropriate containers. We record the time after input materials have arrived at the process as the starting time of the process. Then it will generate a random processing time from the distribution that we have defined for it (normal distribution in this case). After the process is finished we place the output in the appropriate container so that it can be used as input for the next process. We record this time as the ending time of the process. Simple calculation will allow us to figure out the total process-time, which will be recorded into the global lists we previously defined.

```
def assembling(env, garment_factory):
 global assembling_wait
 while True:

   yield garment_factory.post_collar_mkr.get(1)
   yield garment_factory.post_body_mkr.get(1)
   yield garment_factory.post_sleeve_mkr.get(2)
   assem_in = env.now
   assembling_time = max(np.random.normal(6, 2,1)[0], 0) # Control so we
don't get negative values
   yield env.timeout(assembling_time)
   yield garment_factory.post_assembling.put(1)
   assem_out = env.now
```

Unless the input materials and their associated output have a single attribute, then the function will get very complicated very quickly. We have tested 3 different methods to solve this problem. For example, if a piece of material has two attributes {part: body, size: s}, it is a body piece for a size s shirt. Then from the material sourcing step we need to set up 6 different containers (3 parts x 2 attributes) to keep the different materials separated. It is also more difficult to utilize shared resources such as sewing machines as the different materials can get mixed up with each other, so we would need to create additional internal processes to keep different materials separated from each other. The second method is to create sorting processes that would assign output materials into attribute-associated containers. This method allows different materials to utilize the same processes and shared resources. This method is closer to real life factory set ups and is the simplest method. The last method is to hold the attributes in a Python list or dictionary, however, this method is my least favorite as it requires re-assignment of resource attributes after each process, which can slow down the simulation, and creates a less realistic model. Ultimately, we decided to simplify the simulation model by having only single attribute materials.

Between steps 4 and 5 we create additional generator functions consisting of for loops to generate multiples of a resource. These generator functions are lazy iterators that do not exit after the function is run but the state function is remembered throughout the simulation which also adds stochasticity between the resources.

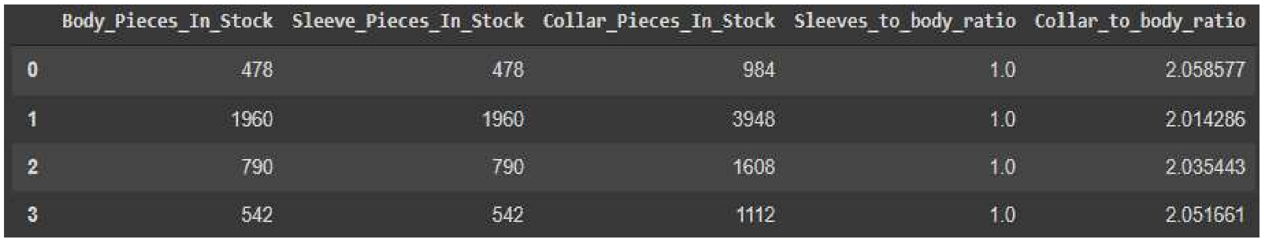
```
def serger_gen(env, garment_factory):
 for i in range(num_serging_mc):
   env.process(serger(env, garment_factory))
   yield env.timeout(0)
```

In step 5, we input the random seed and set up the environment. In step 6, we call on the generators to create events and run the processes. In step 7, we run the simulation. The control and dispatch processes create a loop that replenishes materials and disposes of finished products. This allows the simulation to keep running without any disruptions caused by lack of supply or storage capacity. In fact, the model simulation will run in production loops, each loop is a new order until we tell it to stop. Therefore, we set the simulation end time to be 7 days, during which the factory system receives 12 orders and makes 6284 shirts, 87.1% of which pass inspection.

Our Evaluation of SimPy

In our opinion, SimPy has many strengths. SimPy is free, open-source, and easy to download. Since it's built in Python, a language we are familiar with, using SimPy does not require learning a new language. This makes it quick to learn, and facilitates pulling in outside data or connecting to inputs or outputs of another Python program. One can use their preferred IDE to code a SimPy simulation, and collaborate easily with others using tools like Google Colab. These points are in contrast to Arena, which requires purchasing a program to run simulations beyond a basic capacity and does not integrate easily with other data sources or programs. Had we done the same project in Arena, it would have been very difficult and time consuming to share code and build the model together. SimPy's website includes a number of helpful tutorials and examples which demonstrate key features of the package. We find SimPy's documentation to be easier to read and navigate than Arena's. However, it can be difficult to locate additional examples on how to code a specific process beyond what is found on SimPy's website. In addition, SimPy has drastically lower running simulation-time compared to Arena, which allows us to quickly make adjustments to our code in between simulation runs and verify the results.

SimPy does have some limitations, most notably that it is not visual. To create even a simple simulation in SimPy, the user would likely want to create a separate diagram of all the steps, including how materials and entities move between each process. This is in contrast to Arena, where the program itself is a diagram and it is easy to connect different processes. The fact that SimPy is not visual also makes it more difficult to test and debug the model as one builds it. While in Arena the model diagram allows for quick diagnosis of where a problem is occurring or where more resources may be needed (i.e. because there is a long queue forming), in SimPy the user would have to manually insert a number of print statements, or output a control chart post-simulation run to achieve the same goal. For example, take the stock control chart below: if we did not set up our serging process to serge each set of shirts together but instead had individual serging processes and time-distributions for each shirt part, the rate at which the materials get used up would be different (sleeve-to-body-ratio and collar-to-body-ratio). Monitoring these rates allows us to quickly figure out where the bottlenecks are, which in our case are the earlier processes in the system.

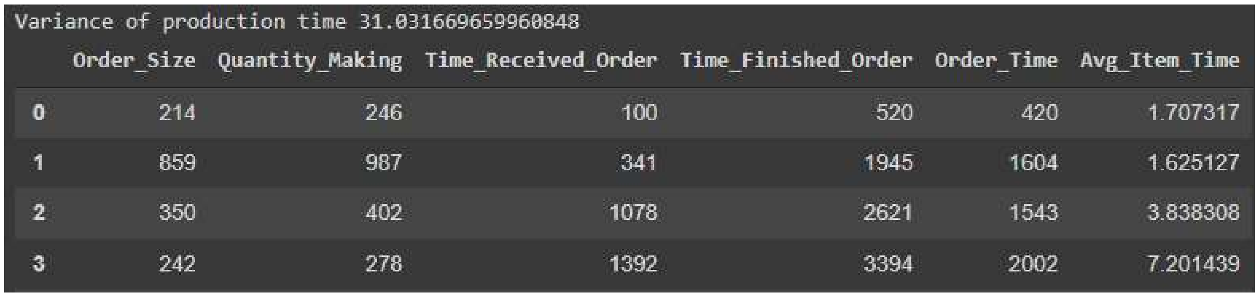


	Body_Pieces_In_Stock	Sleeve_Pieces_In_Stock	Collar_Pieces_In_Stock	Sleeves_to_body_ratio	Collar_to_body_ratio
0	478	478	984	1.0	2.058577
1	1960	1960	3948	1.0	2.014286
2	790	790	1608	1.0	2.035443
3	542	542	1112	1.0	2.051661

Table 1: Stock Control Chart

SimPy does not have any built-in methods for multiple attributes declaration, a functionality that can be achieved easily in Arena where multiple attributes can be assigned to an entity and reassigned as it moves through different processes in the system. This forces the users to make up for this inadequacy with some creative coding, which makes the simulation system significantly more complicated, and coupled with the lack of visualization, we are more prone to make mistakes.

The other main drawback we found with SimPy is in reporting. SimPy does not automatically generate any reporting or track any statistics. The user must manually decide what to report on, in what format, build those variables into the code, and then create their own report. In contrast, Arena offers several options for automatic reporting, which save significant time and effort. The lack of automated reporting and visuals would also make a SimPy model and simulation results harder to share with stakeholders, as it falls entirely on the user to track summary statistics and visualize the model. However, there are many available Python data formats that we can output the simulation results into. In addition, any statistical calculation, or data manipulation needed can be done in Python easily and quickly, even if we have a large amount of simulation data. For example, we have the option to output our results into any tabular format available in Python that we want. The figure below is a pandas dataframe of our simulation results, in which we calculate the average time it takes to make one shirt in each order. The SciPy package in Python also allows us to calculate the variance of the entire system in seconds. We manage to make improvements to the system by trying to improve the variance in average production time of a shirt, in multiple ways such as lowering the value of min_stock variable to increase time between orders.

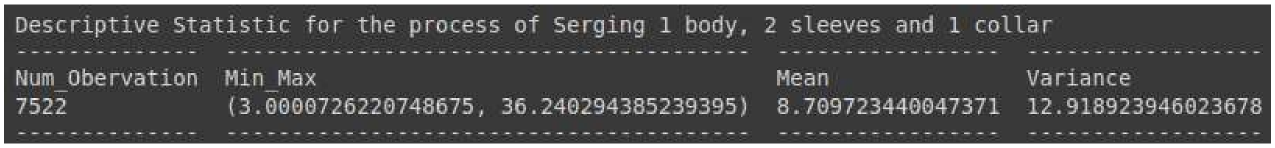


Variance of production time 31.031669659960848

	Order_Size	Quantity_Making	Time_Received_Order	Time_Finished_Order	Order_Time	Avg_Item_Time
0	214	246	100	520	420	1.707317
1	859	987	341	1945	1604	1.625127
2	350	402	1078	2621	1543	3.838308
3	242	278	1392	3394	2002	7.201439

Table 2: Statistical Overview of the System.

It is also very easy to calculate the statistics for an individual process and output into a tabular format. For example, for the serging process, we use the built in method `stats.describe()` to quickly return meaningful statistics such as mean and variance, which helps us make specific changes to individual processes.



```
Descriptive Statistic for the process of Serging 1 body, 2 sleeves and 1 collar
-------------  --------------------------------------  ----------------  ------------------
Num_Obervation  Min_Max                                 Mean              Variance
7522            (3.0000726220748675, 36.240294385239395)  8.709723440047371  12.918923946023678
-------------  --------------------------------------  ----------------  ------------------
```

Table 3: Serging process statistics

Overall, we think SimPy is a viable simulation program for certain use cases. If cost is a prohibitive factor, or if a simulation is needed as a one-time analysis and a company deems it is not worth investing in paid software, SimPy is one option to consider. SimPy is also good for situations where multiple individuals need to collaborate on coding the model, or when the user is familiar with Python and needs to learn a simulation program with limited time and instructional resources. In situations where simulations are frequently used and shared with stakeholders, or where the model is more complex, it is likely worth investing in a paid program that would automate reporting and visualization.

Conclusion

We find SimPy to be intuitive to implement once you understand how to use it. It is a good, free, Python-based alternative to Arena and can be used to build a range of discrete-event simulations. While the initial model planning stage took longer with SimPy than Arena, once we had the structure of our program set up and knew how to generate and connect processes, implementation was not difficult. We were able to apply many of the same concepts we learned in class with Arena in SimPy, including generating entities with different arrival rates, modeling processes with different distributions, reporting summary statistics, and diagnosing bottlenecks.

Ideas for future work on this project include building a full, end-to-end model of a more complex garment factory. This might include manufacturing multiple pieces of clothing, and would incorporate preprocessing orders and design work before the manufacturing process begins. We would also include multiple quality checkpoints, explicitly model what happens to items that do not pass quality control, and build a more detailed version of packing and shipping items. We would also learn how to simulate more advanced situations in SimPy, including assigning different attributes to entities (i.e. multiple sizes of shirts) and differentiating between resources (i.e. creating a resource schedule or assigning different resources different sets of skills). During our research, we also came across a book on the ModSimPy package in Python [3], which focuses on methods to simulate continuous events such as epidemic events, heat diffusions, etc. The topics were all very fascinating and are great ideas for future projects. It would also be interesting to explore using other Python libraries to visualize the simulation, or connecting to inputs and outputs of other programs.

References:

[1] Lünsdorf, O. & Scherfke, S.(2013). Documentation for SimPy (3.0) [Online]. Available: https://simpy.readthedocs.io/en/latest/contents.html#

[2] Stratis, K. (2016). How to Use Generators and yield in Python [Online]. Available: https://realpython.com/introduction-to-python-generators/

[3] Downey, A.(2020). Modeling and Simulation in Python [Online]. Available : https://greenteapress.com/modsimpy/ModSimPy3.pdf