Gabrien Clark & Siham Elmali

ISYE 6644 project

Fall2020

Table of Contents

1. **Abstract**
2. **Background and problem description**
   - 2.1. What is simulation?
   - 2.2. Discrete-Event simulation paradigms
3. **Introduction to the SimPy framework**
   - 3.1. SimPy installation
   - 3.2. SimPy overview
   - 3.3. Tutorial with few basic examples
     - 3.3.1. **Coffee_machine.py**: our first simulation in SimPy
     - 3.3.2. **Traffic_lights.py**: Another simple example
     - 3.3.3. **Clothing_store.py**: simulating a clothing store
     - 3.3.4. **Inventory.py:** simulating an (s,S) inventory in SimPy
   - 3.4. Advantages
   - 3.5. Disadvantages
4. **Future Work**
5. **Conclusion**
6. **References**
7. **Appendix**

1. Abstract:

This project report illustrates our modest attempt to give a brief, yet thorough overview of SimPy; a powerful discrete-event simulation framework written in the modern programming language Python. The report goes through how the framework works and its basic features, while providing an intuitive user guide for anyone who is interested in using Python to simulate real world problems.

2. Background and problem description

2.1. What is simulation

Simulation is a powerful tool that allows us to model and analyze nearly any type of real world or conceptual system over a given time period. Through simulation we are able to analyze how these systems function as a whole and how entities interact within each system, giving us the capability to create more effective plans for system designs as well as answer questions that are not easily solved through analytical and numerical methods.

2.2. Discrete-Event simulation paradigms

Discrete event simulation (see figure 1) involves the modeling of a system as it evolves over time using a representation in which the state of system variables change quickly at separate points in time[1].

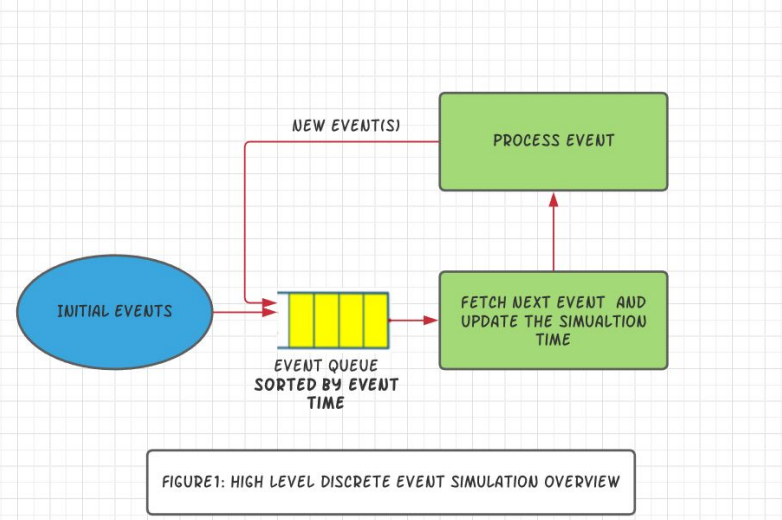

FIGURE1: HIGH LEVEL DISCRETE EVENT SIMULATION OVERVIEW

Discrete event simulation is a great tool for studying models that are far too complicated for analytical or numerical approaches. However, it requires a significant programming effort, and the data collection process required for effective simulation can be time-consuming and costly[2]. Additionally, simulation languages can often be difficult to use or even to debug. For this reason, many simulation languages or paradigms were developed to help simulation

professionals achieve clarity in their code and make the data collection process more seamless[3].

There are several world views when it comes to simulation handling, but the most widely known paradigms, or modeling approaches, are Event Scheduling and Process Interaction.

In the *Event-scheduling* approach, we focus on the events and how they affect the state of the system. This modeling approach requires us to keep track of every event in the simulation.

In the Process-Interaction approach, our focus is on a generic customer and the sequences of events and activities it undergoes as it goes through the system. The simulation language used for this modeling approach keeps track of all the events and how they evolve through the system[1][4].

3. Introduction to the SimPy framework

SimPy is a discrete-event simulation framework written in Python. It was originally created by Klaus G. Müller and Tony Vignaux in 2002 as an open source project. In 2008, Ontje Lünsdorf and Stefan Scherfke started contributing to the project and both became project maintainers in 2011. The first stable version of SimPy was released in 2013 as SimPy 3, while the latest version SimPy 4.0.1 was released on April 15th, 2020.

3.1. SimPy installation

SimPy is an easy to install Python 3 package that has no other package dependencies, though it does require Python 3.6 or greater to run. Installation is available using the following methods:

- PIP
    - From a computer's shell, use the command: `python -m pip install simpy`
- Installing From Source Code
    - Download the latest version of source code (currently 4.0.1) manually from http://pypi.python.org/pypi/SimPy/
    - Extract the archive to a desired location
    - From the same directory, use the following terminal command: `python setup.py install`

3.2. SimPy overview

All of the components that we simulate in SimPy are modeled with processes that live in an environment. They interact with each other and with said environment through events.

Processes essentially simulate entities that evolve in time, e.g. one customer who needs to be served by a server at a restaurant. These processes are described using Python generators. Process functions create events and yield them in order to wait for them to be triggered during the duration of the simulation[5]. Please note that when using a generator function, we are not returning, but rather yielding—that is the main difference between a generator function and a normal function.

When an event is yielded, the process gets suspended and SimPy resumes the process when the event occurs or is triggered. SimPy has the ability to resume multiple processes that are waiting for the same event in the order in which they were yielded.

In SimPy, the most important event type is the **Timeout** event. **Timeout** events are triggered after a certain amount of simulated time has passed and they allow the process to hold its previous state for the given amount of time. All types of events in SimPy can be created using the **Environment** method.

Example:

```
env = simpy.Environment()
env.timeout
```

3.3. Tutorial with few basic examples

3.3.1. **Coffee_machine.py**: our first simulation in SimPy

Our first simulation example is a coffee machine process. We will simulate the machine's usage at an office for example where the customers' interarrival rate is following an exponential distribution with λ = 1/3.

Our coffee_machine process requires a reference to the SimPy Environment **(env)** to be able to create new events. The coffee machine behavior is described in an infinite while loop. Generator function will pass the control flow back to the simulation every time it reaches a yield statement. Once the event occurs, the simulation will resume the function at this statement.

In the **coffee_machine** generator function, the function yields a Timeout event that is following an exponential distribution with λ = 1/3. Whenever the event is triggered, we increment the number of customers using the machine. The function announces its new state by printing the current simulation time (**env.now)** and the customer count.

Simulation code:

```
# a basic example: simulating a coffee machine usage where customer's
interarrival time is 3 mins
```

```
import numpy as np
import simpy

def coffee_machine(env):
    customer_count = 0
    while True:
        yield env.timeout(np.random.exponential(3))
        customer_count += 1
        print(f"At {env.now:0.3f} minutes, customer {customer_count}
used the coffee machine")

# create a simpy environment
env = simpy.Environment()

# initialize a simpy process
env.process(coffee_machine(env))

print("coffee machine simulation output:\n")

# Run the process
env.run(until=100)
```

Coffee machine simulation output:

```
At 1.768 minutes, customer 1 used the coffee machine
At 7.120 minutes, customer 2 used the coffee machine
At 7.254 minutes, customer 3 used the coffee machine
At 7.286 minutes, customer 4 used the coffee machine
At 16.953 minutes, customer 5 used the coffee machine
At 17.672 minutes, customer 6 used the coffee machine
At 18.639 minutes, customer 7 used the coffee machine
At 25.565 minutes, customer 8 used the coffee machine
At 26.480 minutes, customer 9 used the coffee machine
At 28.730 minutes, customer 10 used the coffee machine
At 29.118 minutes, customer 11 used the coffee machine
At 29.614 minutes, customer 12 used the coffee machine
At 34.761 minutes, customer 13 used the coffee machine
At 35.801 minutes, customer 14 used the coffee machine
At 37.212 minutes, customer 15 used the coffee machine
At 42.193 minutes, customer 16 used the coffee machine
At 47.746 minutes, customer 17 used the coffee machine
At 49.165 minutes, customer 18 used the coffee machine
At 49.255 minutes, customer 19 used the coffee machine
At 50.460 minutes, customer 20 used the coffee machine
```

```
At 52.386 minutes, customer 21 used the coffee machine
At 52.721 minutes, customer 22 used the coffee machine
At 54.412 minutes, customer 23 used the coffee machine
At 62.859 minutes, customer 24 used the coffee machine
```

3.3.2. **Traffic_lights.py**: Another simple example

In this example, we are going to discuss a traffic lights process. A traffic light stop alternates between green, yellow and red. Traffic lights in general are dynamic and change with respect to many factors, but for the purpose of our example we are going to assume that the duration of each light is predefined.
Every time the light changes, the generator function will print out the current simulation clock time.

Simulation code:

```
import simpy
import numpy as np

def traffic_lights_process(env):

    while True:
        print("Light turned green at t=" + str(env.now))
        green_light_duration = 60
        yield env.timeout(green_light_duration)

        print("Light turned yellow at t=" + str(env.now))
        yellow_light_duration = 10
        yield env.timeout(yellow_light_duration)

        print("Light turned red at t=" + str(env.now))
        red_light_duration = 40
        yield env.timeout(red_light_duration)

#set a seed to reproduce the results
np.random.seed(0)

#create a simpy environment
env = simpy.Environment()

#instantiate a simpy process
env.process(traffic_lights_process(env))
```

```
print("Traffic light simulation begins")

#run the process for 1000 seconds
env.run(until=1000)

print("Traffic light simulation completes")
```

Simulation output:

```
Light turned red at t=840
Light turned green at t=880
Light turned yellow at t=940
Light turned red at t=950
Light turned green at t=990
Traffic light simulation completes
```

3.3.3. **Clothing_store.py**: simulating a clothing store

The following example was adapted from a PyData conference presentation on simulation[7]. It is using an object-oriented approach.

Suppose you are an operations research scientist who is tasked to create a simulation for a clothing company that specialized in special occasion clothing rental (wedding dresses, suits, cocktail dresses, formal dresses, etc.). The clothing company wants to know when orders are being made and when customers are returning the items they ordered. The simulation needs to keep track of how long the customer is going to hold on to the rented items, how many items the store has, and how many days are between each customer's order.

Simulation code:

```
import simpy

class Store(object):
    def __init__(self, env, num_items, holding_time):
        self.env =  env
        self.items = simpy.Resource(env, num_items)
        self.holding_time =  holding_time

    def send_order(self, customer):
        print(f"sending item for {customer}")
```

```
        yield self.env.timeout(self.holding_time)
        print(f"item from {customer} returns to the store")

def order(env, customer, store):
    print(f"{customer} is requesting an item")

    with store.items.request() as request:
        yield request
        yield env.process(store.send_order(customer))


def setup(env, num_items, holding_time, order_period):
    store = Store(env, num_items, holding_time)
    num_customers = 10
    for i in range(1,num_customers):
        env.process(order(env, 'Customer %d' % i, store))

    while True:
        yield env.timeout(order_period)
        i += 1
        env.process(order(env, 'Customer %d' % i, store))

NUM_ITEMS = 100    # number of items in the store
HOLDING_TIME = 2  # how long the customer is keeping the item in days
ORDER_PERIOD = 3  # time period between each order

env = simpy.Environment()
env.process(setup(env, NUM_ITEMS, HOLDING_TIME, ORDER_PERIOD))

# run the simulation for 7 days
env.run(until=7)
```

Simulation output:

```
Customer 1 is requesting an item
Customer 2 is requesting an item
Customer 3 is requesting an item
Customer 4 is requesting an item
Customer 5 is requesting an item
Customer 6 is requesting an item
Customer 7 is requesting an item
Customer 8 is requesting an item
Customer 9 is requesting an item
```

```
sending item for Customer 1
sending item for Customer 2
sending item for Customer 3
sending item for Customer 4
sending item for Customer 5
sending item for Customer 6
sending item for Customer 7
sending item for Customer 8
sending item for Customer 9
item from Customer 1 returns to the store
item from Customer 2 returns to the store
item from Customer 3 returns to the store
item from Customer 4 returns to the store
item from Customer 5 returns to the store
item from Customer 6 returns to the store
item from Customer 7 returns to the store
item from Customer 8 returns to the store
item from Customer 9 returns to the store
Customer 10 is requesting an item
sending item for Customer 10
item from Customer 10 returns to the store
Customer 11 is requesting an item
sending item for Customer 11
```

3.3.4. **Inventory.py:** simulating an (***s,S***) inventory in SimPy

Our final simulation example will discuss a safety stock inventory model and include example output analysis. In this simulation, customers request to purchase units of a singular product (demand) from an arbitrary store, which works to maintain a minimum daily level of inventory (***s***). If on any given day, customer demand depletes the store's inventory below ***s***, then they will order enough inventory until they reach a target level of safety stock for inventory (***S***). For every order the store makes, it must wait a constant lead time (***L***) to receive the inventory. Every day the store collects revenue from the customer demand of inventory at a given price, while incurring costs for holding inventory units from one day to the next (***h***) and cost incurred by ordering more units of inventory from a vendor.

Simulation Code (adapted from Paul Grogan[6]):

```
import simpy
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

%matplotlib inline
sns.set()

h = 2.0
r = 100
L = 2.0

def warehouse_run(env, order_cutoff, order_target):
    global inventory, balance, num_ordered

    inventory = order_target
    balance = 0.0
    num_ordered = 0

    while True:
        interarrival = generate_interarrival()
        yield env.timeout(interarrival)
        balance -= inventory * h * interarrival
        demand = generate_demand()
        if demand < inventory:
            # sell equal to demand, increase balance and decrease
inventory
            balance += r*demand
            inventory -= demand
```

```
            print(f"{env.now:.2f} Sold {demand}")
        else:
            # sell what we can, reduce inventory to 0
            balance += r*inventory
            inventory = 0
            print(f"{env.now:.2f} Sold {inventory}, now out of
stock")

        if inventory < order_cutoff and num_ordered==0:
            # if inventory is below safety level and we haven't
ordered
            # new inventory yet, then we want to place a new order
            env.process(handle_order(env, order_target))

def handle_order(env, order_target):
    global inventory, balance, num_ordered

    num_ordered = order_target - inventory # calculate # of inventory
to order
    print(f"{env.now:.2f} We placed an order for {num_ordered}
units")
    cost = 50 * num_ordered # calculate cost of order
    balance -= cost # subtact cost of order from profit balance
    yield env.timeout(L) # wait for delay period
    inventory += num_ordered # reset inventory to previous inventory
+ new inventory
    num_ordered = 0 # reset number of inventory to order back to 0
    print(f"{env.now:.2f} received order of {num_ordered} units")

def generate_interarrival():
    return np.random.exponential(1./5) # lambda = 5

def generate_demand():
    return np.random.randint(1,5)

obs_time = []
inventory_level = []

def observe(env):
    global inventory

    while True:
        obs_time.append(env.now)
```

```
        inventory_level.append(inventory)
        yield env.timeout(0.1) # record observations 10 times a day


np.random.seed(77)
env = simpy.Environment()

s = 10
S = 30
env.process(warehouse_run(env, s, S))
env.process(observe(env))
env.run(until=5.0)
```

Simulation Output:

```
0.50 Sold 1
0.59 Sold 2
0.62 Sold 1
0.70 Sold 2
0.78 Sold 1
1.02 Sold 1
1.18 Sold 2
1.29 Sold 4
1.65 Sold 4
2.16 Sold 4
2.16 We placed an order for 22 units
2.23 Sold 2
2.24 Sold 2
2.25 Sold 2
2.65 Sold 0, now out of stock
2.69 Sold 0, now out of stock
2.90 Sold 0, now out of stock
2.92 Sold 0, now out of stock
3.43 Sold 0, now out of stock
3.54 Sold 0, now out of stock
4.08 Sold 0, now out of stock
4.16 received order of 0 units
4.24 Sold 2
4.32 Sold 4
4.67 Sold 2
4.67 Sold 3
4.88 Sold 1
4.94 Sold 1
4.94 We placed an order for 21 units
```

The results of the simulation can be easily collected using python objects, such as lists or numpy arrays, and then visualized to aid in output analysis. Figure 2 below demonstrates how an analyst could generate potential profitability outcomes across parameterized simulation runs. For e.g, fig 2c (10,30) below shows that inventory is likely to be depleted a little after the start of day 1, replenished on day 3, and then likely depleted again by day 4. The same analyst could also use the simulation output to determine that the (5,30) inventory system is likely the most profitable for their business (given by maximum end balance) across the simulated safety and target inventory levels.

Figure 2. Various Simulated (s,S) Inventory Systems

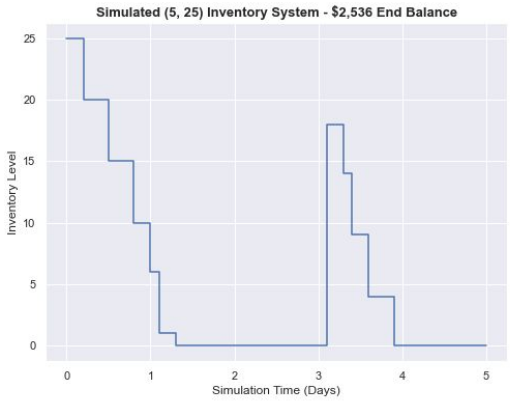

Fig 1a. (5,25)

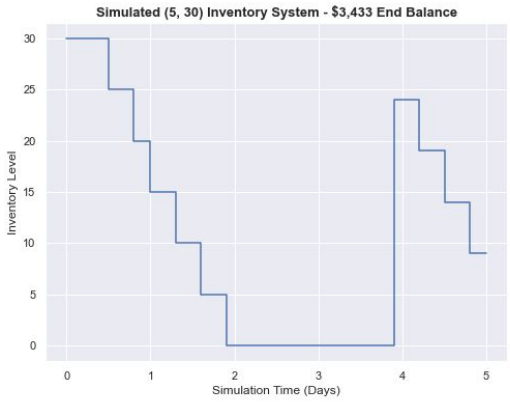

Fig 1b. (5,30)

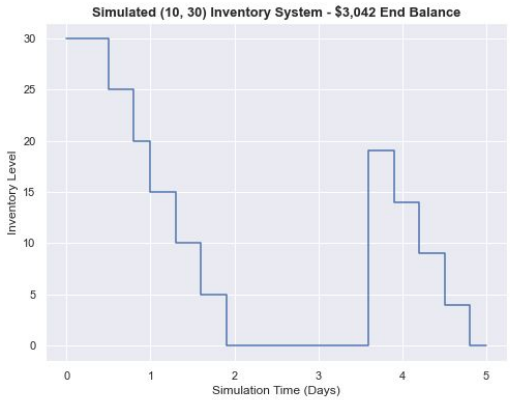

Fig 1c. (10,30)

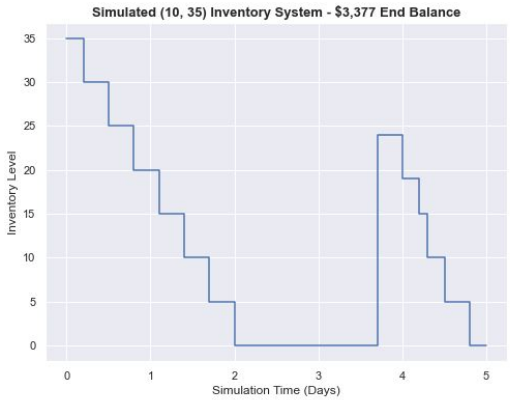

Fig 1c. (10,35)

3.4. Advantages

A major advantage of SimPy is that it's a discrete event simulation framework that takes on the process-interaction "worldview" and manages the event scheduling for the user. SimPy, by design, handles movement from one event to another in its internal functions, while making simulation control easy to control through its process(), run(), and timeout() modules.

A second major advantage of SimPy is its intrinsic flexibility as a Python package. The package is heavily reliant on combining generators with its `timeout` function, which allows programmers to iterate through simulation processes and acquire results with ease. A simple example of this is the traffic light example above, in 11 simple lines of code a user can encode a full simulation process with 3 potential outputs per event. Similarly, output analysis (discussed above) can be combined with other popular Python packages (such as numpy, pandas, and matplotlib) to create easily calculated and highly interpretable results and visualizations. An example of this can be seen above with the output plot created in the (s,S) inventory model example.

Finally, another critical advantage of SimPy is that it is a free-to-use and open-source package. SimPy offers an easy-to-use, powerful simulation framework that requires no cost but access to a computer capable of running Python and SimPy's minimum requirements. While other popular simulation programs, such as ARENA, have free editions for students, the cost model for SimPy allows it to be adopted by nearly everyone, which potentially helps drive more widespread use of simulation as an analysis and decision making tool. Additionally, the open-source nature of the package allows for the Python and simulation community to collaborate to improve SimPy, and (1) continually improve the library over time and (2) provide support to each other to solve complicated problems and package issues as they arise.

3.5. Disadvantages

Some disadvantages for SimPy are grounded in the fact that it is an open-source, free to use python framework. The first being that it actually requires beginner to intermediate knowledge of the Python language, including a need for proficiency with object oriented programming fundamentals to maximize its value. For many people, this prerequisite knowledge can be a large barrier and may require weeks to months of training to gain. Next, the framework is completely programming based and lacks an easy to use GUI that other (albeit proprietary) simulation frameworks, such as ARENA, have in place. Connected to the first disadvantage, this leads to the requirement that users spend time learning how the framework is structured within the context of Python. Finally, proprietary software like ARENA, has a built-in dedicated support structure, whereas SimPy is entirely reliant on a community of users. In many ways this may not necessarily be a bad thing, and can even be an advantage for building wide support, but the fact there is no dedicated support can lead to major issues when a high priority simulation project is underway and experiences issues with the framework.

4. Future Work

For future work, advanced SimPy modules could be explored. Specifically, the **interrupt()** and **cancel()** modules offer a lot of advanced simulation control that add high value to SimPy's toolkit. The i**nterrupt()** method allows one event to interrupt another and this is very useful when modeling a machine breakdown. The **cancel()** method comes in handy when one wants to cancel a later event in the case when you have a thread waiting for one of two events; whichever occurs first will trigger the thread to resume its execution. The Resource class has some advanced features that can handle priority policies in the queue similar to what we saw in the call center simulation in Arena.

Additionally, future work could also expand upon and build more robust simulation examples, such as a busy call center or a chip manufacturing factory.

5. Conclusion

SimPy is a powerful simulation framework and it can be used for nearly any type of simulation. SimPy has become a strong competitor to most of the commercial simulation software and its maintainers are constantly adding new features to its repertoire. The language has several advanced features that are aimed to make a worthwhile reduction on programming efforts and increase in program clarity.

6. References:

[1] Averil, M. Law. *Simulation Modeling and Analysis.* McGraw Hill Education, 2015.

[2] Dave Goldsman, Simulation modeling for scientists and engineers, lecture 1, Fall 2020

[3] Schruben, L., & Yücesan, E. (1993). Modeling paradigms for discrete event simulation. *Operations Research Letters, 13*(5), 265-275. doi:10.1016/0167-6377(93)90049-m

[4] Dave Goldsman, Simulation modeling for scientists and engineers, General simulation principles lecture, Fall 2020

[5] Simpy.readthedocs.io. 2020. *Basic Concepts — Simpy 4.0.2.Dev1+G2973dbe Documentation*. [online] Available at: <https://simpy.readthedocs.io/en/latest/simpy_intro/basic_concepts.html#what-s-next> [Accessed 23 November 2020]

[6] Paul Grogan. "Inventory System Discrete Event Simulation in Python (Process Interaction)." *YouTube*, uploaded by Paul Grogan, 29 Oct. 2016, www.youtube.com/watch?v=Kmu9DNQamLw

[7] Meghan Heintz. *Launching a new warehouse with SimPy at Rent the Runway. PyData Conference, New York City, 2019*

7. Appendix:

The following resources were used in our initial research on how to use simpy:

- Simpy documentation: https://simpy.readthedocs.io/en/latest/
- API: https://simpy.readthedocs.io/en/3.0.1/api_reference/index.html
- Guide: https://simpy.readthedocs.io/en/latest/topical_guides/index.html
- Examples: https://simpy.readthedocs.io/en/latest/examples/index.html
- Real Python tutorial : https://realpython.com/simpy-simulating-with-python/
- *Practical data analysis cookbook* on O'Reilly (chapter 11 focuses on simulation using SimPy)
- *Hands on simulation modeling with Python* on O'Reilly