Analysis of Hospital Simulation in Python

Lucía Molina, Marta Pérez, and Sofía Pérez

April 4, 2025

1 Introduction

This project presents a discrete event simulation of a hospital environment using `SimPy`, a Python-based simulation framework. The primary objective is to model and analyze the patient flow through various hospital departments, considering different medical conditions and their corresponding treatments.

The simulation includes key hospital resources such as emergency rooms, doctors, nurses, radiology rooms, operating rooms, and waiting areas. Patients arrive at the hospital with varying symptoms and severity levels (Mild, Severe, or Critical) and are directed to the appropriate treatment pathways. The hospital operates under predefined constraints, including limited medical staff and infrastructure, which influence patient wait times and treatment efficiency.
General Process Diagram [Arrival] – [Triage] – [Diagnosis] – [Treatment] – [Discharge]

2 Execution Steps

In order for the simulation to run correctly, it is essential that the required libraries are installed beforehand, these include: simpy, numpy, pandas, and tqdm. Once the environment is properly set up, the simulation must follow these steps for correct execution:

First, run the main file. This will display the simulation output in the terminal and simultaneously generate a .csv file containing the simulation results, allowing for a clearer and more structured analysis.
Although the Jupyter Notebook has already been created, you can re-run each code cell to verify that everything functions as expected.

This simulation can be executed under two different scenarios. In our case, we chose to develop the mass emergency scenario, as it was more interesting to implement. However, if the normal situation is executed instead, the generated .csv file will contain different data. Therefore, re-running the notebook after this change would yield different results based on the newly simulated information.

3 Project Structure

The project is divided into multiple directories, each handling a specific aspect of the hospital simulation:

```
hospital_simulation/
          main.py
          config.py

          models/
                patient.py
                staff.py
                hospital.py
                event.py

          simulation/
                scheduler.py
                triage.py
                diagnosis.py
                treatment.py

          utils/
                randomizer.py
```

- `main.py`: Defines the simulation and the arrival of patients at the hospital.
- `hospital.py`: Contains the hospital definition, available resources, and medical processes.
- `event.py`: Models events occurring within the simulation.
- `patient.py`: Defines patient attributes.

- `staff.py`: Includes classes for medical staff (Doctors, Nurses, and Assistants).
- `scheduler.py`: Responsible for scheduling events within the simulation environment.
- `triaje.py, diagnosis.py, treatment.py, discharge.py`: Implement the different medical procedures that may occur during the simulation.
- `logger.py`: Logs events and activities within the simulation.
- `randomizer.py`: Generates random patients with specific characteristics.

This modular approach follows software engineering best practices, improving maintainability, readability, and scalability. By keeping related functionalities together, we ensure separation of concerns, making it easier to update, debug, and expand specific parts of the project without affecting others. For example, the hospital logic is independent of how events are scheduled, allowing modifications to patient triage (triage.py) without affecting the event scheduler (scheduler.py). The utility functions prevent redundancy, making the system more efficient and reusable. This structure also facilitates collaboration, as different team members can work on separate modules simultaneously without conflicts.

4 Code Analysis

4.1 main.py File

The `main.py` file sets up and runs the simulation. It imports various libraries and modules to manage the hospital simulation.

```
import simpy
import random
import numpy as np
from config import *
from models.patient import Patient
from models.hospital import Hospital
from simulation.scheduler import Scheduler
from models.event import Event
from simulation.triage import perform_triage
from simulation.diagnosis import perform_diagnosis
from simulation.treatment import perform_treatment
```

```
from utils.randomizer import generate_patient
import pandas as pd
from tqdm import tqdm
from itertools import product
import os
```

The main function `main()` is responsible for initializing the simulation environment and setting up the scenario.

```
random.seed(random_seed)  # Set random seed for
    reproducibility
    np.random.seed(random_seed)


    arrival_rate_map = {
        "mass_emergency": emergency_arrival_interval,
        "normal": normal_arrival_interval
    }

    arrival_rate = arrival_rate_map.get(scenario,
        normal_arrival_interval)  # Si no se encuentra el
        escenario, usa el valor por defecto

    # Initialize simulation environment and hospital
    env = simpy.Environment()
    hospital = Hospital(env, capacity_emergencies,
        num_doctors, num_nurses, capacity_waiting_room,
        capacity_surgery_rooms)
    scheduler = Scheduler(env)

    env.process(patient_arrival(env, hospital, scheduler,
        arrival_rate))  # Start patient arrival process
    env.run(until=simulation_duration)  # Run the simulation
```

In our simulation, we aimed to represent two different scenarios. The first one is a normal situation on a regular day without any special or extraordinary events. This corresponds to option number 1 in our menu.

The second is a more unusual and therefore more interesting scenario to analyze, which involves a mass emergency. In this case, we simulate an event such as a major traffic accident, the collapse of a building, or any situation that results in a large number of people being taken to the hospital. This corresponds to option number 2 in our menu.

```
def choose_scenario():
    """Displays a menu to select the scenario for the
        simulation."""
    print("Select␣the␣type␣of␣simulation:")
    print("1.␣Normal␣emergency")
```

```
    print("2. Mass emergency")
    choice = input("Enter the number of the desired option (1
        or 2): ")

    if choice == "1":
        return "normal"
    elif choice == "2":
        return "mass_emergency"
    else:
        print("Invalid option. 'Normal emergency' will be
            selected by default.")
        return "normal"
```

To carry out both simulations, we have created a menu where you can choose which simulation you want to run. This way, we avoid having both simulations running at the same time, which makes it difficult to understand what is happening in the hospital when it appears on the screen. As a result, it is much clearer for the person executing it.

```
if __name__ == "__main__":
    """
    Main entry point for the hospital simulation. This
        function runs a series of simulations with varying
        hospital capacities,
    generates patient data, and saves the results to a CSV
        file.
    """

    # Print a message indicating the start of the simulation
    print("Simulaci n Hospitalaria")

    # Obtener la selecci n del usuario para el tipo de
        simulaci n
    scenario = choose_scenario()

    # Ejecutar la simulaci n con el escenario seleccionado
    main(scenario=scenario)

    # Capacity configurations for various hospital resources
    capacities_emergencies = [1, 2, 3, 4, 5]
    capacities_doctors = [1, 2, 3, 4, 5]
    capacities_nurses = [2, 4, 6, 8, 10]
    capacities_waiting_room = [5, 10, 20, 30]
    capacities_operating_room = [1, 2, 3, 5]

    # List to store all simulation data
    all_data = []

    # Total number of simulations based on combinations of
```

```
    capacities
total_simulations = (
    len(capacities_emergencies) * len(capacities_doctors)
        * len(capacities_nurses) * len(
        capacities_waiting_room) * len(
        capacities_operating_room)
)

# Run simulations with progress bar
for capacity in tqdm(product(capacities_emergencies,
    capacities_doctors, capacities_nurses,
    capacities_waiting_room, capacities_operating_room),
    total=total_simulations):
    env = simpy.Environment()
    hospital = Hospital(env, *capacity)

    # Generate and process 5 patients for each simulation
        run
    for i in range(5):
        patient = generate_patient()
        env.process(hospital.receive_patient(patient))

    # Run the simulation silently and collect the event
        data
    history = hospital.silent()
    all_data.extend(history)

# Create the "data_analysis" directory if it doesn't
    exist
if not os.path.exists("data_analysis"):
    os.makedirs("data_analysis")

# Save the collected data into a CSV file in the "
    data_analysis" folder
df = pd.DataFrame(all_data, columns=["Patient ID", "
    Symptom", "Severity", "Arrival Time", "Departure Time"
    ])

# Map severity and symptom values to numerical values for
    analysis
severity_map = {"Mild": 0, "Severe": 1, "Critical": 2}
symptom_map = {
    "Chest Pain": 0,
    "Fracture": 1,
    "Sore Throat": 2,
    "Difficulty Breathing": 3,
    "Lump": 4
}
```

```
    # Convert "Severity" and "Symptom" columns to numerical
        values
    df["Severity"] = df["Severity"].map(severity_map)
    df["Symptom"] = df["Symptom"].map(symptom_map)

    # Save the data as a CSV file
    df.to_csv("data_analysis/hospital_simulation.csv", index=
        False)

    # Print a message indicating the completion of the
        simulations and data saving
    print("\nSimulations␣completed.␣The␣data␣has␣been␣saved␣
        in␣'data_analysis/hospital_simulation.csv'")
```

In this function, the simulation is executed, and the Cartesian product is created, which results in a .csv file that is automatically generated when the code is run.
For the Cartesian product, we considered it important to include the gravity and symptoms of the patients. However, since these are strings, it made it difficult to analyze them later in the Jupyter notebook.
Therefore, what we considered appropriate was to convert each symptom and each severity level into a number by mapping them, so that the subsequent analysis of the simulation could be done without issues.
As a result, when you view the .csv file, the symptoms and severity levels are not listed as strings but as numbers, since leaving them as strings would cause problems during the later analysis.

4.2 File `hospital.py`

This file contains the `Hospital` class, which represents the core of the medical simulation. It manages all the hospital resources, including the staff, the rooms, and the clinical processes.

```
import simpy
import random
import numpy as np
from models.staff import Doctor, Nurse, AuxiliaryNurse
from models.patient import Patient
from simulation.triage import perform_triage
import config
import sys
import io
```

The `Hospital` class initializes the various resources using `simpy.Resource`, and creates lists with `Doctor`, `Nurse`, and `AuxiliaryNurse` objects, which

represent hospital staff. Additionally, it assigns common symptoms to medical specialties through a dictionary.

```
def silent(self):
        """
        Runs the simulation without printing any messages to
           the console.

        This method redirects the standard output to a buffer
           , runs the simulation for a period of time,
        and then restores the original standard output. The
           method returns the simulation history data.

        Returns:
            list: The history of events that occurred during
               the simulation.
        """
        # Save the original standard output
        old_stdout = sys.stdout
        sys.stdout = io.StringIO()  # Redirect the output to
           an empty buffer

        self.env.run(until=config.simulation_duration)

        # Restore the original standard output
        sys.stdout = old_stdout

        return self.history  # Return the stored data
```

This function generates the simulation, which is then saved in the .csv file. We created the function so that this does not display on the screen when running the main program. Instead, it is directly saved to the file and can be viewed from there. This way, it is clearer for the person running the program, and both parts can be seen clearly without confusion.

Main functions of the class:

- `receive_patient`: main flow for each patient. Executes triage, waiting, examinations, treatment, and discharge.
- `perform_discharge`: simulates the medical discharge process after treatment.
- `emergency_attention`, `urgent_consultation`, `general_consultation`: clinical processes with variable durations.
- `wait_in_room`: simulates the patient's waiting period before being attended.

- `perform_xray`, `perform_surgery`: specialized procedures.
- `consult_specialist`: directs the patient to a medical office according to their symptom.
- `register_event`: stores event information in the `history` list for later analysis.
- `silent`: runs the simulation without printing messages, useful for testing or large-scale simulations.

Each procedure uses resources and random durations to represent uncertainty and hospital system load. For example, a surgery can last between 30 and 90 minutes, followed by a recovery time between 30 and 60 minutes.
Flow example: A critical patient with respiratory difficulty might go through:

1. Triage
2. Waiting room
3. X-ray
4. Surgery
5. Post-surgical recovery
6. Medical discharge

The modular implementation and the use of `SimPy` make it easy to realistically simulate multiple patients and resources in parallel.

4.3 `event.py` File

This file defines the `Event` class, which represents an event in the system, such as the arrival of a patient or the completion of a diagnosis.

```
class Event:
    """Represents a medical event in the hospital simulation.

    Attributes:
        event_type (str): The type of event (e.g., "Diagnosis
            ", "Treatment").
        time (float): The time when the event occurs in the
            simulation.
        patient (str): The identifier of the patient
            associated with the event.
    """
```

```
    def __init__(self, event_type: str, time: float, patient:
        str) -> None:
        """Initializes an Event instance.

        Args:
            event_type (str): The type of event occurring.
            time (float): The timestamp of the event in the
                simulation.
            patient (str): The ID or name of the patient
                involved in the event.
        """
        self.event_type = event_type
        self.time = time
        self.patient = patient

    def __repr__(self) -> str:
        """Returns a string representation of the Event
            instance.

        Returns:
            str: A formatted string displaying the event
                details.
        """
        return f"Event({self.event_type}, {self.time}, {self.
            patient})"
```

4.4 `patient.py` File

The `patient.py` file defines the `Patient` class, which has attributes such as the patient's severity, the symptom they present, and their identifier.

```
class Patient:
    """Represents a patient in the hospital simulation.

    Attributes:
        severity (str): The severity level of the patient's
            condition (e.g., "Mild", "Moderate", "Critical").
        symptom (str): The main symptom or reason for the
            patient's visit.
        id (int): A unique identifier for the patient.
    """

    def __init__(self, severity: str, symptom: str, id: int)
        -> None:
        """Initializes a new Patient instance.

        Args:
```

```
            severity (str): The severity level of the patient
                's condition.
            symptom (str): The main symptom of the patient.
            id (int): A unique identifier for the patient.
        """
        self.severity = severity
        self.symptom = symptom
        self.id = id

    def __repr__(self) -> str:
        """Returns a string representation of the Patient
            object.

        Returns:
            str: A formatted string containing the patient's
                ID, severity, and symptom.
        """
        return f"Patient(ID: {self.id}, Severity: {self.
            severity}, Symptom: {self.symptom})"
```

4.5 `staff.py` File

The `staff.py` file defines the object-oriented data structures that represent the hospital staff. These classes model the **Doctors**, **Nurses**, and **Nursing Assistants**, allowing the storage of relevant information such as name, shift, years of service, and specialty (in the case of doctors).

- **Doctor**: This class includes additional attributes such as the medical specialty (e.g., Cardiologist, Pulmonologist) and allows distinguishing professionals based on their area of care. Each instance contains:
  - `specialty`: doctor's medical field.
  - `name`, `id`: personal identifiers.
  - `shift`: assigned time slot (morning, afternoon, or night).
  - `years_of_service`: years of work experience at the center.
- **Nurse** and **AuxiliaryNurse**: These classes share the same structure as `Doctor`, except for the absence of the `specialty` field. They represent the nursing staff and assistants, respectively.
- **Text Representation**: Each class overrides the `__repr__()` method, allowing for a clear and readable representation of their attributes when objects of these classes are printed. This is useful for tracking and tracing the professionals in the console during the simulation.

4.6 File `triage.py`

This module manages the triage process, determining the severity of patients and directing them to the appropriate treatment based on their condition. The main function is `perform_triage`, whose code is partially shown below:

```
import numpy as np

def perform_triage(env, patient, hospital):
    """Performs triage on a patient and directs them to the
       appropriate care level.

    This function evaluates the severity of the patient's
       condition and assigns them to the correct
    treatment path: emergency care for critical cases, urgent
        consultation for severe cases, or
    general consultation for less severe cases.

    Args:
        env (simpy.Environment): The simulation environment.
        patient: The patient undergoing triage.
        hospital: The hospital instance managing patient care
            .

    Yields:
        simpy.Event: A process representing the triage
            duration and subsequent patient care actions.
    """
    print(f"{env.now}␣\t|␣Triage␣completed␣|␣Patient␣{patient
       .id}␣|␣Level:␣{patient.severity}")
    hospital.register_event(patient, "Triage", env.now)

    severity = patient.severity
    symptom = patient.symptom
    triage_duration = np.random.uniform(3, 7)  # Randomized
       triage duration

    yield env.timeout(triage_duration)  # Simulate triage
       processing time

    if severity == "Critical":
        print(f"{env.now}␣\t|␣Emergency:␣Patient␣{patient.id}
            ␣transferred␣to␣emergency␣care.")
        yield env.process(hospital.emergency_attention(
            patient))  # Send to emergency care
    elif severity == "Severe":
        print(f"{env.now}␣\t|␣Urgent␣consultation␣for␣patient
            ␣{patient.id}.")
```

```
            yield env.process(hospital.urgent_consultation(
                patient))  # Send to urgent consultation
        else:
            print(f"{env.now} \t| Normal consultation for patient
                {patient.id}.")
            yield env.process(hospital.general_consultation(
                patient))  # Send to general consultation

            # After consultation, the patient is discharged
            yield env.process(hospital.perform_discharge(patient)
                )
```

In this function, we have differentiated between patients who are critical, severe, or mild, so that depending on their condition, they are assigned to different rooms. This makes the process more efficient and more closely resembles a real hospital.

This flow ensures care proportional to the severity of the case, optimizing resources and prioritizing critical situations. The triage process acts as the first clinical and logistical filter for the efficient allocation of the hospital's medical services.

4.7 File `diagnosis.py`

The `diagnosis.py` file simulates the process of diagnosing a patient. Depending on the patient's symptoms, an X-ray may be required before the diagnosis.

```
import random
import numpy as np

def perform_diagnosis(env, patient, hospital):
    """Simulates the diagnosis process for a patient in the
        hospital.

    This function manages the diagnosis of a patient, which
        may include
    performing an X-ray if required. The process takes a
        random amount of time.

    Args:
        env (simpy.Environment): The simulation environment.
        patient (Patient): The patient undergoing diagnosis.
        hospital (Hospital): The hospital where the diagnosis
            takes place.

    Yields:
```

```
        simpy.Event: A process representing the time taken
            for diagnosis
        and possible X-ray examination.
    """
    print(f"{env.now}: Diagnosis for {patient.id}")

    # If the patient's symptom requires an X-ray before
        diagnosis
    if patient.symptom in ["Fracture", "Respiratory
        Difficulty"]:
        yield env.process(hospital.perform_xray(patient))

    # Simulate the time taken for diagnosis (random between
        10 and 20 minutes)
    yield env.timeout(np.random.uniform(10, 20))

    print(f"{env.now}: Diagnosis completed for {patient.id}")
```

This process simulates the diagnosis of a patient, which takes between 10 and 20 time units. If necessary, the patient undergoes an X-ray prior to the diagnosis.

4.8 File `treatment.py`

The treatment.py module defines the **performtreatment()** function, which models the medical intervention phase in the hospital workflow. This function receives as parameters the simulation environment **(env)**, the patient, and the hospital object, to coordinate care according to the severity of the case.

```
def perform_treatment(self, env, patient, hospital):
    """Manages the treatment process for a patient.

    This function determines the appropriate treatment based
        on the patient's severity.
    Critical patients undergo surgery, while non-critical
        patients receive standard treatment.

    Args:
        self: The object instance (if applicable).
        env (simpy.Environment): The simulation environment.
        patient: The patient undergoing treatment.
        hospital: The hospital instance managing the
            treatment process.

    Yields:
        simpy.Event: A process representing the time taken
            for treatment and discharge.
    """
```

```
        print(f"{env.now}: Treatment for patient {patient.id}")

        if patient.severity == "Critical":
            # If the patient is in critical condition, they
               require surgery
            yield env.process(hospital.perform_surgery(patient))
        else:
            # Standard treatment for non-critical patients
            yield env.timeout(np.random.uniform(10, 30))  #
               Treatment duration

        print(f"{env.now}: Treatment completed for patient {
            patient.id}")

        # Patient is discharged after treatment
        yield env.process(hospital.perform_discharge(patient))
```

In this function, we have made a distinction between critical patients and the rest, where the former are sent to surgery and the others continue with the normal process.
Once either treatment path is completed, the patient's discharge is processed via `hospital.perform_discharge()`, indicating that the care process has been fully completed.

4.9 `scheduler.py` File

This file defines the `Scheduler` class, which is responsible for scheduling events within the simulation. Events may include patient arrivals, diagnoses, treatments, and medical discharges.

```
import simpy
from models.patient import Patient

class Scheduler:
    """Manages the scheduling and execution of events in the
       simulation.

    The Scheduler class is responsible for storing scheduled
       events and
    processing them within the simulation environment.

    Attributes:
        env (simpy.Environment): The simulation environment.
        events (list): A list storing all scheduled events.
    """

    def __init__(self, env: simpy.Environment) -> None:
```

```
        """Initializes the Scheduler with a simulation
           environment.

        Args:
            env (simpy.Environment): The simulation
               environment where events will be scheduled.
        """
        self.env = env
        self.events = []

    def schedule_event(self, event, patient: Patient) -> None
        :
        """Schedules an event and starts its processing.

        Args:
            event: The event to be scheduled.
            patient (Patient): The patient associated with
               the event.
        """
        self.events.append(event)
        self.env.process(self.process_event(event, patient))

    def process_event(self, event, patient: Patient):
        """Processes the scheduled event by waiting for the
           required duration.

        Args:
            event: The event to be processed.
            patient (Patient): The patient associated with
               the event.

        Yields:
            simpy.Event: A process representing the waiting
               time for the event.
        """
        yield self.env.timeout(event.time)
```

The `Scheduler` manages the events through a list, where new events for each patient can be added and processed sequentially.

4.10 `config.py` File

This module contains the global parameters that regulate the hospital environment simulation. By centralizing these variables, it makes it easier to adjust and experiment with different scenarios without directly modifying the simulation code.

```
"""Configuration file for the hospital simulation.
```

```
This module defines various parameters that control the
    behavior of the hospital
simulation, including staff numbers, resource capacities,
    patient arrival intervals,
and simulation duration.
"""

# Random seed for reproducibility
random_seed = 42

# Number of hospital staff
num_doctors = 60  # Total number of doctors available
num_nurses = 100  # Total number of nurses available
num_auxiliaries = 100  # Total number of auxiliary nurses
    available

# Hospital resource capacities
capacity_emergencies = 20  # Number of available emergency
    room beds
capacity_waiting_room = 80  # Maximum number of patients in
    the waiting room
capacity_xray_room = 4  # Number of available X-ray machines
capacity_surgery_rooms = 10  # Number of operating rooms
capacity_rooms = 10  # Number of patient recovery rooms

# Patient arrival intervals (in minutes)
normal_arrival_interval = 5  # Average time between normal
    patient arrivals
emergency_arrival_interval = 1  # Average time between
    emergency patient arrivals

# Simulation duration (in minutes)
simulation_duration = 480  # Total simulation time (8 hours)

# Doctor office capacities by specialty
doctor_office_capacity = {
    "Cardiologist": 1,  # Number of cardiology consultation
        rooms
    "Pulmonologist": 1,  # Number of pulmonology consultation
         rooms
    "Traumatologist": 1,  # Number of traumatology
        consultation rooms
    "Internist": 1,  # Number of internal medicine
        consultation rooms
    "Oncologist": 1  # Number of oncology consultation rooms
}
```

In this file, we have declared the parameters that define the hospital, such as

the number of doctors, nurses, and assistants, which will be used throughout the code to ensure they are consistent across all methods and functions
Importance: This file serves as the parametric foundation of the system, allowing a clear separation between simulation logic and configuration. In this way, it is possible to simulate hospitals of different sizes, care loads, or staff availability by simply modifying this module.

4.11 File `randomizer.py`

This file generates random patients with attributes such as severity and symptoms.

```
import random
from models.patient import Patient

# Global counter to assign unique IDs to each generated
    patient
patient_counter = 1

def generate_patient():
    """Generates a new patient with random severity and
       symptoms.

    This function creates a new patient instance, assigning a
        unique ID and randomly selecting
    the severity level and symptom from predefined lists.

    Returns:
        Patient: A new patient instance with assigned
           severity, symptom, and unique ID.
    """
    global patient_counter  # Keep track of patient IDs

    # Randomly assign severity level
    severity = random.choice(["Critical", "Severe", "Mild"])

    # Randomly assign a symptom
    symptom = random.choice(["Chest␣Pain", "Fracture", "Lump"
       , "Respiratory␣Difficulty", "Sore␣Throat"])

    # Create a new Patient instance
    patient = Patient(severity, symptom, patient_counter)
```

The function `generar_paciente` creates a patient with random severity and symptoms.

5 Created Files

The created files are two: a .csv file where we can see a detailed execution of our simulation, including the variables we considered most important, such as the patient ID, symptoms, severity, arrival time, and departure time. Additionally, we have created a Jupyter notebook where we conducted an analysis of the parameters listed in the .csv file, using various methods.

Clarification: If the file path for the .csv file in the notebook doesn't work, you should use the direct file path of the .csv file generated by the code. You need to change this in the second chunk of the code in the notebook, as reading this file generates a key variable for the simulation analysis.

6 Conclusions

This hospital simulation system efficiently models the arrival of patients, triage, and various medical procedures within a hospital. The use of `SimPy` allows managing the concurrency of different medical processes, providing a realistic and flexible simulation. The modularity of the code facilitates the expansion of the system with new procedures, resources, and events, making it ideal for experiments and performance analysis in a simulated hospital.

7 GitHub

https://github.com/maartaperez/IA