Practical 1

Agent-Based Model for a Grocery Store

1. Creating the Agent Population

Agents in our model will represent customers and staff members in the grocery store.

Customers:

- Attributes: Number of items to purchase, budget, shopping speed.
- Behavior: Enter the store, select items, proceed to checkout, and leave.

-Staff Members:

- Attributes: Number, role (cashier, stocker), efficiency.
- Behavior: Serve customers at checkout, restock shelves.

2. Defining Agent Behavior

Initialization:

- Create a number of customers and staff members.
- Assign initial attributes such as budget for customers and efficiency for staff.

Customer Behavior:

- Move around the store to select items.
- Queue at checkout.
- Pay and leave the store when done.
- Variation: Some customers might leave without purchasing due to long queues or out-of-stock items.

- **Staff Behavior**:

- Cashiers: Serve customers at checkout.

- Stockers: Check stock levels and restock shelves.

-Store Environment:

- Layout: Define the layout of aisles and checkout counters.
- Inventory: Track item availability and restocking.

Example of running the model

```
grocery_model = GroceryStoreModel(num_customers=50, num_staff=5)
for i in range(100):
    grocery_model.step()
```

4. Visualizing the Model Output

To visualize the model output, we can create charts to monitor:

- Number of customers in the store over time.
- Queue lengths at checkout.
- Stock levels of popular items.
- Average wait times at checkout.

Here's a basic example using `matplotlib` for visualizing the number of customers in the store over time:

```
import matplotlib.pyplot as plt

customer_counts = []
for i in range(100):
    grocery_model.step()
    customer_counts.append(len(grocery_model.schedule.agents))

plt.plot(range(100), customer_counts)
plt.xlabel('Time Steps')
plt.ylabel('Number of Customers in Store')
```

```
plt.title('Customer Flow in Grocery Store')
plt.grid(True)
plt.show()
```

Practical-2

To design and develop an agent-based model (ABM) for a restaurant scenario, incorporating factors like agent population (customers and restaurant staff), agent behavior (ordering food, waiting, word of mouth effect), product discards, and delivery time, we'll outline the steps and provide a Python implementation using the `mesa` library, which is suitable for ABM in Python.

1. Installing Required Libraries

First, ensure you have `mesa` installed. You can install it using pip:

bash

```
pip install mesa
```

2. Designing the Agent-Based Model

Let's break down the components and then provide the Python implementation.

Agent Population

-Customers: Agents representing customers who enter the restaurant, order food, and may leave reviews.

Restaurant Staff:Agents representing restaurant staff who take orders, prepare food, and manage customer interactions.

Agent Behavior

-Customer Behavior:

- Enter the restaurant at random intervals.
- Choose items from the menu and place orders.
- Wait for food delivery.
- Optionally leave reviews based on their experience.

- Restaurant Staff Behavior:

- Take orders from customers.

- Prepare food.
- Deliver food to customers.
- Manage customer interactions and reviews.

Additional Considerations

- Word of Mouth Effect: Customers may leave positive or negative reviews based on their experience, influencing future customers.
- Product Discards:Track wasted food due to over-preparation or spoilage.
- *Delivery Time Measure the time taken from order placement to food delivery.

3. Python Implementation

```
from mesa import Agent, Model
from mesa.time import RandomActivation
from mesa.space import MultiGrid
from mesa.datacollection import DataCollector
import random
import matplotlib.pyplot as plt
 Constants
NUM_CUSTOMERS = 50
NUM_STAFF = 5
MENU_ITEMS = ['Burger', 'Pizza', 'Pasta', 'Salad']
MENU_PRICES = {'Burger': 10, 'Pizza': 12, 'Pasta': 15, 'Salad': 8}
MAX_WAIT_TIME = 30  # Maximum waiting time in simulation steps
REVIEWS = {0: "Terrible", 1: "Bad", 2: "Okay", 3: "Good", 4: "Excellent"}  # Review scale

class CustomerAgent(Agent):
    def __init__(self, unique_id, model):
        super().__init__(unique_id, model)
        self.money = random.randint(20, 50)  # Customer's budget
        self.order = random.choice(MENU_ITEMS)  # Random initial order
```

```
        self.wait_time = 0  # Time waited for food
        self.review = None  # Customer's review (None initially)
    def step(self):
        if self.wait_time >= MAX_WAIT_TIME:
            self.model.remove_agent(self)
            return
        if self.wait_time == 0:
            # Place order when first entering
            self.model.place_order(self)
        self.wait_time += 1
    def receive_food(self):
        # Customer receives food, resets wait time
        self.wait_time = 0
    def leave_review(self):
        # Decide on review based on experience
        satisfaction = random.randint(0, 4)
        self.review = REVIEWS[satisfaction]
        return self.review


class StaffAgent(Agent):
    def __init__(self, unique_id, model):
        super().__init__(unique_id, model)

    def step(self):
        if self.model.current_customer:
            # Process current customer's order
            self.model.process_order()

    def take_order(self, customer):
```

```
            customer.order = random.choice(MENU_ITEMS)

    def prepare_food(self):
        # Simulate food preparation time
        self.model.advance_time(random.randint(1, 5))

    def deliver_food(self):
        # Deliver food to customer
        self.model.current_customer.receive_food()
        self.model.current_customer = None


class RestaurantModel(Model):
    def __init__(self, num_customers, num_staff):
        self.num_customers = num_customers
        self.num_staff = num_staff
        self.schedule = RandomActivation(self)
        self.grid = MultiGrid(1, 1, False)
        self.current_customer = None
        self.time = 0
        self.product_discards = 0

        # Create staff agents
        for i in range(self.num_staff):
            staff_agent = StaffAgent(i, self)
            self.schedule.add(staff_agent)

        # Create customer agents
        for i in range(self.num_customers):
            customer_agent = CustomerAgent(i, self)
```

```
            self.schedule.add(customer_agent)

        self.running = True

    def step(self):
        self.schedule.step()
        self.time += 1

    def place_order(self, customer):
        self.current_customer = customer
        for agent in self.schedule.agents:
            if isinstance(agent, StaffAgent):
                agent.take_order(customer)
    def process_order(self):
        for agent in self.schedule.agents:
            if isinstance(agent, StaffAgent):
                agent.prepare_food()
                agent.deliver_food()
    def advance_time(self, time):
        self.time += time
    def remove_agent(self, agent):
        self.schedule.remove(agent)

    def run_model(self, steps):
        for _ in range(steps):
            self.step()

# Data collector for reviews
def compute_reviews(model):
    reviews = [agent.review for agent in model.schedule.agents if isinstance(agent,
CustomerAgent) and agent.review]
```

```
    review_counts = {review: reviews.count(review) for review in REVIEWS.values()}
    return review_counts

# Running the simulation
model = RestaurantModel(NUM_CUSTOMERS, NUM_STAFF)
model.run_model(100)

# Data collection
data_collector = DataCollector(model_reporters={"Reviews": compute_reviews})
data_collector.collect(model)

# Visualization (bar chart of reviews)
model_data = data_collector.get_model_vars_dataframe()
reviews_data = model_data['Reviews'][0]
labels, values = zip(*reviews_data.items())
indexes = range(len(labels))

plt.bar(indexes, values, align='center', alpha=0.5)
plt.xticks(indexes, labels)
plt.ylabel('Number of Reviews')
plt.title('Customer Reviews')

plt.show()
```

Practical-4

System Dynamics Model: Spread of Contagious Disease

1. Creating a Stock and Flow Diagram

In System Dynamics, stocks represent accumulations over time, and flows represent the rates of change into or out of stocks. For a contagious disease model, typical stocks and flows include:

Stocks:

- Susceptible (S): Individuals who can catch the disease.
- Infected (I): Individuals currently infected and capable of spreading the disease.
- Recovered (R): Individuals who have recovered and are immune (or deceased).

- Flows:

- Transmission (T): Rate at which the disease spreads from susceptible to infected individuals.
- Recovery (Rc): Rate at which infected individuals recover and move to the recovered stock.
- Mortality (M): Rate at which infected individuals die.

2. Adding a Plot to Visualize Dynamics

Let's implement a basic System Dynamics simulation using Python and visualize the dynamics over time.

```
import numpy as np
import matplotlib.pyplot as plt

# Parameters
transmission_rate = 0.3  # T: Transmission rate
recovery_rate = 0.1      # Rc: Recovery rate
mortality_rate = 0.05    # M: Mortality rate

# Initial conditions
initial_susceptible = 990
initial_infected = 10
initial_recovered = 0
```

```
# Simulation function
def simulate_disease_spread(days):
    susceptible = [initial_susceptible]
    infected = [initial_infected]
    recovered = [initial_recovered]

    for day in range(1, days):
        new_infected = transmission_rate * susceptible[-1] * infected[-1] / (susceptible[-1] +
infected[-1])
        new_recovered = recovery_rate * infected[-1]
        new_deaths = mortality_rate * infected[-1]

        susceptible.append(susceptible[-1] - new_infected)
        infected.append(infected[-1] + new_infected - new_recovered - new_deaths)
        recovered.append(recovered[-1] + new_recovered)

    return susceptible, infected, recovered

# Simulation
days = 100
susceptible, infected, recovered = simulate_disease_spread(days)

# Plotting
plt.figure(figsize=(10, 6))
plt.plot(range(days), susceptible, label='Susceptible')
plt.plot(range(days), infected, label='Infected')
plt.plot(range(days), recovered, label='Recovered')
plt.xlabel('Days')
plt.ylabel('Population')
plt.title('Spread of Contagious Disease: System Dynamics Model')
plt.legend()
```

```
plt.grid(True)
plt.show()
```

Practical-5

Discrete-Event Simulation Model: Manufacturing and Shipping Process

1. Creating a Simple Model

Our model will simulate the production of products in a factory and their subsequent shipping to customers.

Events:

- Arrival of orders: Customers place orders for products.
- Production: Products are manufactured based on orders.
- Shipping: Manufactured products are shipped to customers.

- Entities:

- Orders: Each order specifies the quantity of products.
- Products: Manufactured items.

Activities:

- Production: Manufacture products based on order requirements.

Shipping: Deliver products to customers.

2. Adding Resources

Resources in our model will include:

- Production Line: Machines and labor for manufacturing.
- Shipping Department: Vehicles and personnel for delivery.

3. Creating 3D Animation

To visualize the model, we'll use Python with libraries like `SimPy` for discrete-event simulation and `Matplotlib` for creating a basic 3D animation.

```
import simpy
import random
import matplotlib.pyplot as plt
import matplotlib.animation as animation

# Parameters
RANDOM_SEED = 42
SIM_TIME = 100
NUM_PRODUCTS = 10

# Model components
class Factory:
    def __init__(self, env, num_machines, num_workers):
        self.env = env
        self.machine = simpy.Resource(env, capacity=num_machines)
        self.worker = simpy.Resource(env, capacity=num_workers)

    def produce(self, product):
        yield self.env.timeout(random.uniform(0.5, 1.5))  # Time to produce product

class Shipping:
    def __init__(self, env, num_vehicles, num_drivers):
        self.env = env
        self.vehicle = simpy.Resource(env, capacity=num_vehicles)
        self.driver = simpy.Resource(env, capacity=num_drivers)

    def deliver(self, product):
```

```
        yield self.env.timeout(random.uniform(1, 3))  # Time to deliver product

def customer(env, factory, shipping):
    for i in range(NUM_PRODUCTS):
        yield env.timeout(1)  # Time between orders
        env.process(order(env, factory, shipping, f'Order_{i}'))

def order(env, factory, shipping, order_name):
    print(f'{env.now:.2f}: New order {order_name}')

    # Production process
    with factory.worker.request() as worker_req:
        yield worker_req
        yield env.process(factory.produce(order_name))

    # Shipping process
    with shipping.driver.request() as driver_req:
        yield driver_req
        yield env.process(shipping.deliver(order_name))

# Simulation setup
env = simpy.Environment()

# Create factory and shipping resources
factory = Factory(env, num_machines=2, num_workers=4)
shipping = Shipping(env, num_vehicles=3, num_drivers=2)

# Start simulation process
env.process(customer(env, factory, shipping))
env.run(until=SIM_TIME)
```

```

4. Modeling Delivery

In the above simulation, orders are processed sequentially. The `customer` function generates orders periodically, and each order is processed through the `order` function, which simulates both production and shipping processes.

5. Creating 3D Animation (Basic Example)

For a basic 3D animation using `Matplotlib`, you can visualize the movement of products from production to shipping:

# Basic 3D Animation Example (simplified)

fig = plt.figure()

ax = fig.add_subplot(111, projection='3d')

def animate(i):

    ax.clear()

    ax.set_title(f'Simulation Time: {i}')

    ax.set_xlim(0, 10)

    ax.set_ylim(0, 10)

    ax.set_zlim(0, 10)

    # Example: Plotting production and shipping movements

    ax.scatter(random.uniform(1, 9), random.uniform(1, 9), random.uniform(1, 9), c='blue', label='Production')

    ax.scatter(random.uniform(1, 9), random.uniform(1, 9), random.uniform(1, 9), c='green', label='Shipping')

    ax.legend()

ani = animation.FuncAnimation(fig, animate, frames=SIM_TIME, interval=1000)

plt.show()
```

Practical-7

Verification involves checking that the simulation model is implemented correctly and behaves as expected according to its specifications. Here's how you can verify a simulation model:

1.Code Review and Testing:

- Bank Model Example: Review the code to ensure that entities (e.g., customers, tellers) are correctly simulated, queues are managed properly, and transactions are processed according to the bank's rules.

- Manufacturing Model Example: Check that the production processes, resource allocations (e.g., machines, workers), and scheduling mechanisms (e.g., job prioritization) are implemented correctly.

2. Unit Testing:

- Test individual components of the simulation model to verify their correctness.

- Bank Model: Test scenarios such as customer arrival rates, service times, and queue behaviors.

- Manufacturing Model: Test scenarios involving production rates, resource constraints, and job completion times.

3. Cross-Verification: - Compare results from different simulation runs using the same inputs to ensure consistency.

- Bank Model: Verify that statistical outputs such as average waiting times, queue lengths, and service utilization match expected values.

- Manufacturing Model: Ensure that production output, resource utilization rates, and job throughput align with expected performance metrics.

Validation :

1. Data Collection:

- Gather historical data or conduct observations of the real system to capture key performance metrics.

- Bank Model: Collect data on customer arrival patterns, service times, and queue lengths.

- Manufacturing Model:Obtain data on production rates, downtime events, and resource utilization.

2.Comparison with Real Data:

- Compare the simulation outputs (e.g., average waiting times, throughput rates) with the collected real-world data.

- Bank Model: Validate if simulated queue lengths and customer waiting times match observed data.

- Manufacturing Model: Validate if simulated production output and resource utilization align with actual performance metrics.

3. Expert Review:

- Seek feedback from domain experts to evaluate the simulation's realism and relevance.

- Bank Model: Consult bank managers or operations experts to validate if the model accurately reflects operational dynamics.

- Manufacturing Model: Engage production managers or engineers to validate if the model captures critical factors influencing production efficiency.

1. Data Collection: Collect data on customer arrival rates, average service times, and queue lengths from the bank's operational records.

2. Model Calibration: Adjust simulation parameters (e.g., arrival rates, service times) to match the collected data.

3. Simulation Runs: Run the simulation multiple times with different scenarios (e.g., peak hours, average days) using calibrated parameters.

4. Validation: Compare simulated outputs (e.g., average waiting times, queue lengths) with real-world data. Ensure that the simulation accurately reflects observed patterns.

5. Expert Review: Obtain feedback from bank managers or customer service experts to assess if the model captures operational dynamics realistically.

6.Documentation: Document the validation process, including any discrepancies or adjustments made to the model, and present findings to stakeholders.

Practical-8

Defense Model for Aircraft Behavior Simulation

1. Define Objectives and Scope

- Simulate the interaction between defense systems (e.g., missile batteries, radar systems) and incoming aircraft threats to evaluate defense effectiveness.

- Focus on simulating various types of aircraft (e.g., fighters, bombers) and defense systems (e.g., surface-to-air missiles, anti-aircraft guns).

2. Model Components

Entities:

- Aircraft Types: Fighters, bombers, drones, etc.

- Defense Systems: Missile batteries, radar stations, anti-aircraft guns, etc.

Events:

- Aircraft Arrival: Generation of incoming aircraft with specified characteristics (speed, altitude, maneuvers).

- Defense System Activation: Triggered when an aircraft enters detection range.

3. Simulation Framework

Choose a suitable simulation framework or tool. For complex simulations like this, discrete-event simulation or agent-based modeling may be appropriate:

Discrete-Event Simulation: Tracks changes in state based on discrete events (e.g., aircraft arrival, missile launch).

-Agent-Based Modeling: Models individual entities (e.g., aircraft, defense systems) with behaviors and interactions.

4. Model Implementation

```
import simpy
import random

class Aircraft:
    def __init__(self, env, name, speed, altitude):
        self.env = env
        self.name = name
        self.speed = speed
        self.altitude = altitude
        self.arrival_time = env.now
        self.detected = False

class DefenseSystem:
    def __init__(self, env, name, range):
        self.env = env
        self.name = name
        self.range = range

def aircraft_generator(env, aircraft_list):
    while True:
        yield env.timeout(random.uniform(5, 10))  # Generate aircraft every 5-10 time units
        aircraft = Aircraft(env, f'Aircraft_{len(aircraft_list) + 1}', random.uniform(500, 1000),
random.uniform(10000, 20000))
        aircraft_list.append(aircraft)
        print(f"{env.now:.2f}: {aircraft.name} approaching")

def radar(env, aircraft, defense_systems):
    for system in defense_systems:
```

```
        if system.range >= aircraft.altitude:  # Aircraft detected
            aircraft.detected = True
            print(f"{env.now:.2f}: {system.name} detected {aircraft.name}")

# Simulation setup
env = simpy.Environment()

# Create defense systems
radar1 = DefenseSystem(env, 'Radar_1', 15000)
radar2 = DefenseSystem(env, 'Radar_2', 20000)
defense_systems = [radar1, radar2]

# Aircraft list
aircraft_list = []

# Start processes
env.process(aircraft_generator(env, aircraft_list))

# Main simulation loop
while True:
    if aircraft_list:
        aircraft = aircraft_list.pop(0)
        env.process(radar(env, aircraft, defense_systems))

    if env.now > 50:  # Simulation time limit
        break

    env.run(until=env.now + 1)  # Advance simulation time
```

5. Visualization and Analysis

Visualization: Use tools like Matplotlib or 3D visualization libraries to create animations or plots showing aircraft movements, defense system activations, and engagement outcomes.

Analysis: Collect and analyze simulation data to assess defense system performance metrics such as detection rates, response times, and effectiveness in neutralizing threats.

6. Iterative Refinement

Refine the model based on validation against real-world data or expert feedback. Adjust parameters and behaviors to better reflect actual defense scenarios and improve simulation accuracy.