TP03 : Simulation de Monte Carlo et Événements Discrets

Partie 1 : Simulation d'Événements Discrets dans un Centre d'Appels avec des Intervalles d'Arrivées Exponentiels

Objectifs

- Modéliser un centre d'appels avec un nombre limité d'agents.
- Utiliser la loi exponentielle pour générer les arrivées des appels et les durées de service.
- Analyser les temps d'attente, la durée des appels et l'utilisation des agents.
- Afficher des histogrammes pour visualiser la distribution des temps d'attente et des durées d'appels.

1- Paramètres et outils de base

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

ARRIVAL_Time = 1 #1 client par minute

SERVICE_Time = 1 #1 appel par minute

SIMULATION_TIME = 480
NUM_AGENTS = 3

#Génération CDF
def exponential_random_variable(rate):
    u = random.random()  # valeur aléatoire uniforme entre 0 et 1
    return -math.log(1 - u) / rate  # Applique l'inversion de la CDF
```

2- Représentation d'un appel entrant

Cette fonction modélise le comportement d'un client appelant le centre. Elle enregistre le temps d'attente et le temps de service.

Objectif : Modéliser le comportement individuel d'un appel entrant (client).

- Elle est exécutée chaque fois qu'un client fait un appel.
- Elle gère :
  1. L'enregistrement du temps d'arrivée.
  2. La demande de service auprès d'un agent (temps d'attente).
  3. La durée de l'appel (temps de service).
  4. Le suivi des temps d'attente et de service dans des listes pour analyse

```
def client(env, name, counter, wait_times, call_times):
    arrival_time = env.now
    print(f"{name} appelle le centre d'appel à {arrival_time:.2f}")

    with counter.request() as req:
        yield req  # Attente que l'agent soit libre
        wait_time = env.now - arrival_time
        wait_times.append(wait_time)

        print(f"{name} commence l'appel à {env.now:.2f} après avoir
attendu {wait_time:.2f} minutes.")

        # Calcul du temps de service
        call_time = exponential_random_variable(SERVICE_Time)
        yield env.timeout(call_time)  # Durée de l'appel

        call_times.append(call_time)
        print(f"{name} termine son appel à {env.now:.2f}. Durée de
l'appel: {call_time:.2f} minutes.")
```

3- Génération des appels entrants

Objectif : Générer des appels entrants selon un processus aléatoire (loi exponentielle).

- Elle crée et planifie l'arrivée des clients dans le temps.
- Elle simule :
  - Les intervalles entre les arrivées des appels (temps aléatoires entre les clients).
  - La création d'un nouveau processus `client` pour chaque appel.
- Elle ne modélise pas directement le comportement des clients mais s'assure qu'ils arrivent selon une distribution réaliste.

```
def Generate_call(env, counter, wait_times, call_times):
    last_arrival_time = 0
    client_count = 0

    while env.now < ST:
        inter_arrival_time =    exponential_random_variable(AT)  #
Intervalle entre les appels suivant une loi exponentielle
        arrival_time = last_arrival_time + inter_arrival_time
```

```
        if arrival_time < ST:
            last_arrival_time = arrival_time
            client_count += 1
            yield env.timeout(inter_arrival_time)

            env.process(client(env, f"Client-{client_count}", counter,
wait_times, call_times))
        else:
            break  # Si le temps dépasse la durée de simulation
```

4- Exécution de la simulation

Cette partie initialise la simulation, exécute les appels et collecte les résultats (temps d'attente, durées d'appel).

```
def run_simulation():
    env = simpy.Environment()
    counter = simpy.Resource(env, capacity=NUM_AGENTS)
    wait_times = []
    call_times = []

    env.process(setup(env, counter, wait_times, call_times))  # Démarre
les appels entrants
    env.run()

    return wait_times, call_times

wait_times, call_times = run_simulation()
```

5- Affichage des résultats

```
average_wait_time = sum(wait_times) / len(wait_times) if wait_times
else 0
print(f"Temps d'attente moyen : {average_wait_time:.2f} minutes")

average_call_time = sum(call_times) / len(call_times) if call_times
else 0
print(f"Durée moyenne des appels : {average_call_time:.2f} minutes")

average_inter_arrival_time = sum(call_times) / len(call_times) if
call_times else 0
print(f"Intervalle moyen entre les appels :
{average_inter_arrival_time:.2f} minutes")
total_service_time = sum(call_times)  #Temps total passé par les agents
en service
```

```
utilization = total_service_time / (ST * NUM_AGENTS)
print(f"Utilisation moyenne des agents : {utilization:.2f}")

plt.hist(wait_times, bins=5, edgecolor='black')
plt.xlabel('Temps d\'attente (en minutes)')
plt.ylabel('Nombre de clients')
plt.title('Distribution des temps d\'attente des clients')
plt.show()

plt.hist(call_times, bins=5, edgecolor='black')
plt.xlabel('Durée des appels (en minutes)')
plt.ylabel('Nombre de clients')
plt.title('Distribution des durées des appels des clients')
plt.show()
```

Partie 2 : Simulations des jeux de cartes avec Monte Carlo

Dans cette partie, on cherche à simuler différents jeux de cartes à l'aide du générateur de nombres aléatoires standard (en utilisant random.sample() pour tirer des cartes et random.randint() pour générer des tirages).

Jeux à simuler :

- **Sahara Ace :** Tirer un As parmi un jeu de 52 cartes. Il y a 4 As dans un jeu standard, donc la probabilité de gagner est de 4/52.
- **Tunisian Twins :** Tirer deux cartes avec le même rang. La probabilité de gagner est le nombre de paires de cartes ayant le même rang (ex. deux 7) parmi les 52 cartes.
- **Royal Flush :** Tirer une quinte flush royale. Il existe seulement 4 quintes flush royales possibles (une par couleur).

1- Définir les cartes d'un jeu

```
suits = ['hearts', 'diamonds', 'clubs', 'spades']
ranks = ['2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K',
'A']
deck = []
for suit in suits:
    for rank in ranks:
        deck.append(f"{rank} of {suit}")
```

2- Simuler le jeu "Sahara Ace"

```
def simulate_sahara_ace(num_trials):
    wins = 0
    for _ in range(num_trials):
```

```
        if random.randint(1, 52) <= 4:  # 4 As
            wins += 1
    return wins / num_trials
```

3- Simuler le jeu "Tunisian Twins"

```
def simulate_tunisian_twins(num_trials):
    wins = 0
    for _ in range(num_trials):
        cards = random.sample(deck, 2)
        rank1 = cards[0].split(' ')[0]
        rank2 = cards[1].split(' ')[0]
        if rank1 == rank2:  # Vérifie le rang
            wins += 1
    return wins / num_trials
```

4- Simuler le jeu "Royal Flush"

```
def simulate_royal_flush(num_trials):
    wins = 0
    royal_flush_ranks = ['10', 'J', 'Q', 'K', 'A']

    for _ in range(num_trials):
        suit = random.choice(suits)
        royal_flush_hand = []
        royal_flush_hand.append(f"10 of {suit}")
        royal_flush_hand.append(f"J of {suit}")
        royal_flush_hand.append(f"Q of {suit}")
        royal_flush_hand.append(f"K of {suit}")
        royal_flush_hand.append(f"A of {suit}")

        available_cards = []
        for rank in ranks:
            available_cards.append(f"{rank} of {suit}")

        # Tirage de 5 cartes
        hand = random.sample(available_cards, 5)

        # Extraire rangs et couleurs cartes tirées
        hand_ranks = []
        hand_suits = []
        for card in hand:
            hand_ranks.append(card.split(' ')[0])
            hand_suits.append(card.split(' ')[2])
```

```
# Vérification quinte flush royale
        if set(hand_ranks) == set(royal_flush_ranks) and
len(set(hand_suits)) == 1:
            wins += 1

    return wins / num_trials
```

5- Analyse des résultats

```
num_trials_list = [10, 100, 1000, 10000, 100000]
results_sahara_ace = []
results_tunisian_twins = []
results_royal_flush = []

# test différents nombres d'essais
for num_trials in num_trials_list:
    results_sahara_ace.append(simulate_sahara_ace(num_trials))
    results_tunisian_twins.append(simulate_tunisian_twins(num_trials))
    results_royal_flush.append(simulate_royal_flush(num_trials))

plt.plot(num_trials_list, results_sahara_ace, label="Sahara Ace")
plt.plot(num_trials_list, results_tunisian_twins, label="Tunisian
Twins")
plt.plot(num_trials_list, results_royal_flush, label="Royal Flush")
plt.xscale('log')
plt.xlabel('Nombre de tirages')
plt.ylabel('Proportion de victoires')
plt.title('Proportion de victoires simulées pour différents jeux de
cartes')
plt.legend()
plt.grid(True)
plt.show()
```

Questions :

1. Comment la taille de l'échantillon (nombre d'essais) affecte-t-elle les résultats de chaque simulation ?

2. Quel est l'impact de la probabilité théorique sur les résultats de la simulation ? Comparer par exemple la probabilité théorique de gagner à "Sahara Ace" (4/52) avec la proportion obtenue dans la simulation.

3. Pourquoi la simulation "Royal Flush" a-t-elle une probabilité de victoire beaucoup plus faible que les autres jeux ?