Departamento de Eletrónica, Telecomunicações e Informática

# Simulação e Otimização

## Relatório Final do Mini-Projeto de Simulação de Recurso

**André Oliveira, 107637**


Mestrado em Engenharia Informática

**Professor**: Nuno Lau

8 de julho de 2025

# Índice

# Capítulo 1

# Introdução

O presente relatório foi desenvolvido no âmbito da unidade curricular de Simulação e Otimização, integrada no Mestrado em Engenharia Informática da Universidade de Aveiro. Este trabalho visa aplicar técnicas de simulação a problemas com características distintas, explorando a sua modelação computacional e análise de resultados.

Ao longo do projeto, foram utilizados métodos numéricos e estatísticos para representar e estudar o comportamento de sistemas dinâmicos e estocásticos. As abordagens adotadas permitiram não só compreender o impacte de diferentes parâmetros no desempenho dos sistemas simulados, como também avaliar a eficácia dos métodos implementados.

A realização deste trabalho permitiu reforçar competências na construção de simuladores, na análise de dados obtidos e na avaliação de desempenho de sistemas complexos. Os resultados obtidos são discutidos e interpretados de forma a fundamentar as decisões tomadas ao longo do desenvolvimento.

# Capítulo 2

# Exercício 1

## 2.1 Introdução

A simulação de eventos discretos (*Discrete Event Simulation* – DES) é uma ferramenta essencial para a análise de sistemas estocásticos e complexos, nos quais o estado do sistema evolui ao longo do tempo em resposta a eventos que ocorrem em instantes discretos. Esta abordagem permite estudar e otimizar o desempenho de sistemas compostos por múltiplos recursos e entidades que competem por atendimento, sendo particularmente útil quando a modelação analítica não é viável.

Neste exercício, foi modelado e simulado um sistema de atendimento composto por dois servidores do tipo A e um servidor do tipo B. Os clientes chegam ao sistema segundo uma distribuição exponencial com média de 1 minuto e, no momento da chegada, são classificados como clientes do tipo 1 (com probabilidade de 0,8) ou do tipo 2 (com probabilidade de 0,2).

Clientes do tipo 1 podem ser atendidos por qualquer servidor, preferindo os do tipo A sempre que disponíveis. O tempo de serviço segue uma distribuição exponencial com média de 0,8 minutos. Por outro lado, clientes do tipo 2 requerem atendimento simultâneo por um servidor do tipo A e um servidor do tipo B, com tempos de serviço uniformemente distribuídos entre 0,5 e 0,7 minutos. Quando não existem recursos disponíveis, os clientes são colocados em filas separadas, por tipo, seguindo uma política *FIFO* (First-In, First-Out).

O sistema simulado pode ser enquadrado no contexto dos *Queuing Systems*, apresentando os principais componentes clássicos:

- **Processo de chegadas**: Os clientes chegam segundo um processo estocástico de Poisson, com tempos entre chegadas $A_i \sim \text{Exp}(\lambda)$, onde $\lambda = 1$ cliente/minuto, ou seja, uma distribuição exponencial com média de 1 minuto.
- **Mecanismo de serviço**: O sistema dispõe de múltiplos servidores com capacidades diferenciadas. Os tempos de serviço são exponenciais com média $E(S) = 0{,}8$ para clientes do tipo 1 e uniformes entre 0,5 e 0,7 para clientes do tipo 2.
- **Disciplina de fila**: As filas são separadas por tipo de cliente e seguem uma política *FIFO*. Clientes do tipo 2 têm prioridade se ambos os tipos de servidores estiverem livres.

Apesar da estrutura do sistema não permitir uma representação direta pela notação clássica $GI/G/s$, é possível, de forma simplificada, aproximar o atendimento dos clientes do tipo 1 a um modelo $M/M/2$. Para os clientes do tipo 2, tal aproximação não é válida, dado que os seus tempos de serviço não seguem uma distribuição exponencial e requerem simultaneamente dois recursos distintos.

A simulação foi desenvolvida em `Python`, implementando manualmente todos os elementos essenciais de uma simulação de eventos discretos: lista de eventos futuros ordenada por tempo, relógio de simulação

explícito, variáveis de estado e acumuladores estatísticos. A execução decorre durante um horizonte de 1000 minutos simulados.

Durante a simulação, são recolhidas as seguintes estatísticas:

- **Tempo médio de espera em fila** por tipo de cliente;
- **Tempo médio total no sistema** por tipo de cliente (fila + serviço);
- **Número médio de clientes em fila**;
- **Número médio de clientes no sistema** (fila + serviço);
- **Percentagem de utilização de cada servidor**, discriminada por tipo de cliente, obtida como a razão entre o tempo total de serviço prestado e o tempo total simulado.

Estas estatísticas permitem avaliar o desempenho do sistema em regime estacionário, possibilitando a análise de métricas como as descritas acima. Além da configuração base, foram também simulados cenários alternativos com a adição de um servidor do tipo A ou do tipo B, com o objetivo de comparar o impacte de cada opção na redução do maior dos atrasos médios em fila, considerando ambos os tipos de cliente.

## 2.2 Modelação do Sistema

A modelação do sistema foi realizada de acordo com os princípios da Simulação de Eventos Discretos, recorrendo à identificação explícita dos principais componentes que regem a dinâmica de um sistema de filas com múltiplos servidores e clientes. Esta secção apresenta a formalização do modelo conceptual e lógico, bem como os elementos fundamentais da simulação.

### 2.2.1 Entidades e Recursos

De acordo com a definição clássica de sistemas de eventos discretos, as entidades representam os objetos dinâmicos de interesse na simulação. No sistema em estudo, as entidades são os clientes, que são gerados ao longo do tempo, entram em filas, competem por recursos, são atendidos e eventualmente deixam o sistema. Cada entidade possui atributos que definem o seu comportamento no sistema. No caso dos clientes, os principais atributos são:

- **Tipo**: identifica se o cliente é do tipo 1 ou tipo 2, influenciando os recursos que requer;
- **Tempo de chegada**: instante em que o cliente entra no sistema, usado para calcular tempos de espera;
- **Tempo de início de serviço**: registado no momento em que o cliente começa a ser atendido;
- **Tempo de serviço**: duração do atendimento, determinada aleatoriamente com base na distribuição definida para o respetivo tipo de cliente.

Os recursos são os elementos que as entidades requisitam para serem atendidas. Neste modelo, os recursos são os servidores: dois do tipo A e um do tipo B. Clientes do tipo 1 requerem um único servidor (preferencialmente do tipo A), enquanto clientes do tipo 2 necessitam simultaneamente de um servidor do tipo A e um do tipo B. Os recursos são alocados durante o tempo de serviço e libertados no seu término.

### 2.2.2 Estado do Sistema

O estado de um sistema de simulação de eventos discretos é definido como o conjunto de variáveis que contêm toda a informação necessária para descrever completamente o sistema num determinado instante de tempo. Estas variáveis refletem não apenas o número de entidades ativas, mas também a situação dos recursos e o tempo atual da simulação.

No modelo desenvolvido, o estado do sistema é composto pelas seguintes variáveis:

- **Tempo de simulação**: valor atual do relógio da simulação;
- **Ocupação dos servidores**: vetores booleanos que indicam se cada servidor do tipo A ou B está ocupado;
- **Tipo de cliente em serviço**: registo do tipo de cliente atualmente em atendimento em cada servidor;
- **Filas de espera**: listas ordenadas com os tempos de chegada dos clientes em espera, separadas por tipo;
- **Lista de eventos futuros**: fila de prioridade que contém os eventos agendados para execução futura, ordenados pelo instante em que devem ocorrer.

Estas variáveis, em conjunto, definem o estado completo do sistema num dado instante e permitem a sua evolução ao longo do tempo de simulação. A atualização do estado ocorre a cada evento, sendo estas variáveis essenciais para determinar o comportamento dinâmico do modelo.

### 2.2.3 Fluxogramas e Grafo de Eventos

De forma a representar visualmente a lógica de funcionamento do sistema, foram construídos fluxogramas para os três principais eventos do modelo: `Chegada`, `Partida do tipo 1` e `Partida do tipo 2`. Estes fluxogramas descrevem a sequência de decisões e ações executadas em cada evento, conforme definido na lógica da implementação.

O fluxograma do evento de chegada, ilustrado na Figura 2.1, inicia-se com a geração de um novo evento de chegada e a determinação do tipo de cliente. Caso o cliente seja do tipo 1, verifica-se se existe um servidor do tipo A ou B disponível para o atender. Se for do tipo 2, verifica-se a disponibilidade simultânea de um servidor A e de um servidor B. Se os recursos estiverem ocupados, o cliente é encaminhado para a fila respetiva.

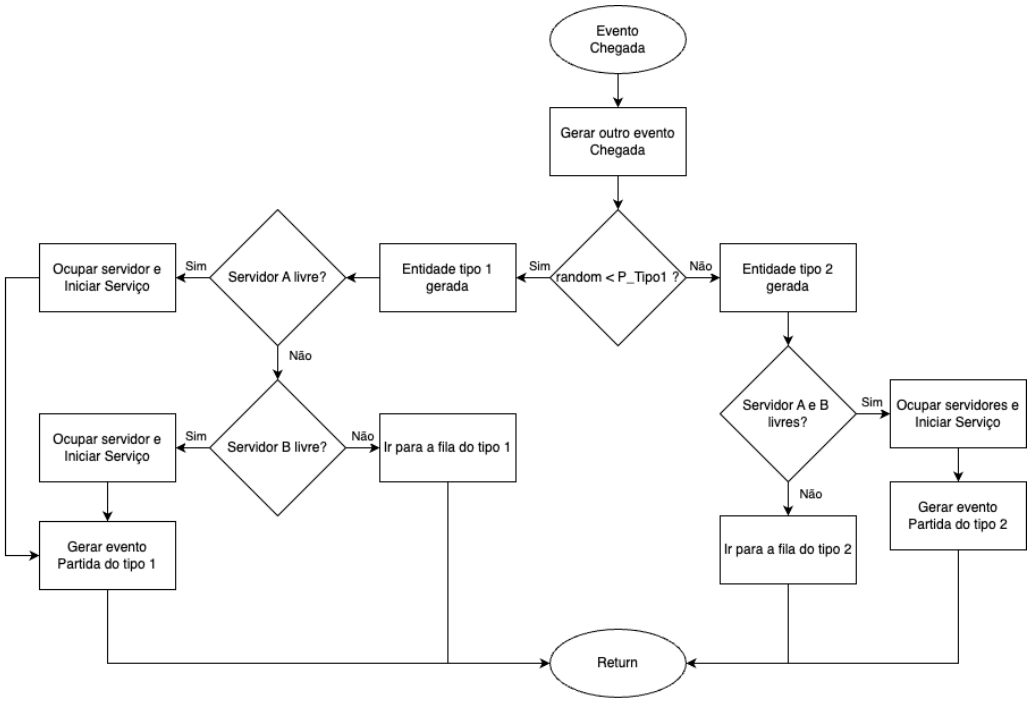


Figura 2.1: Fluxograma do evento de chegada

A Figura 2.2 apresenta o fluxograma correspondente ao evento de partida do tipo 1. Este inicia-se com a libertação do servidor ocupado e dá prioridade ao atendimento de um cliente do tipo 2, caso exista um em espera e os servidores A e B estejam disponíveis. Se tal não for possível, verifica-se se há clientes do tipo 1 em espera e algum servidor disponível para os atender.

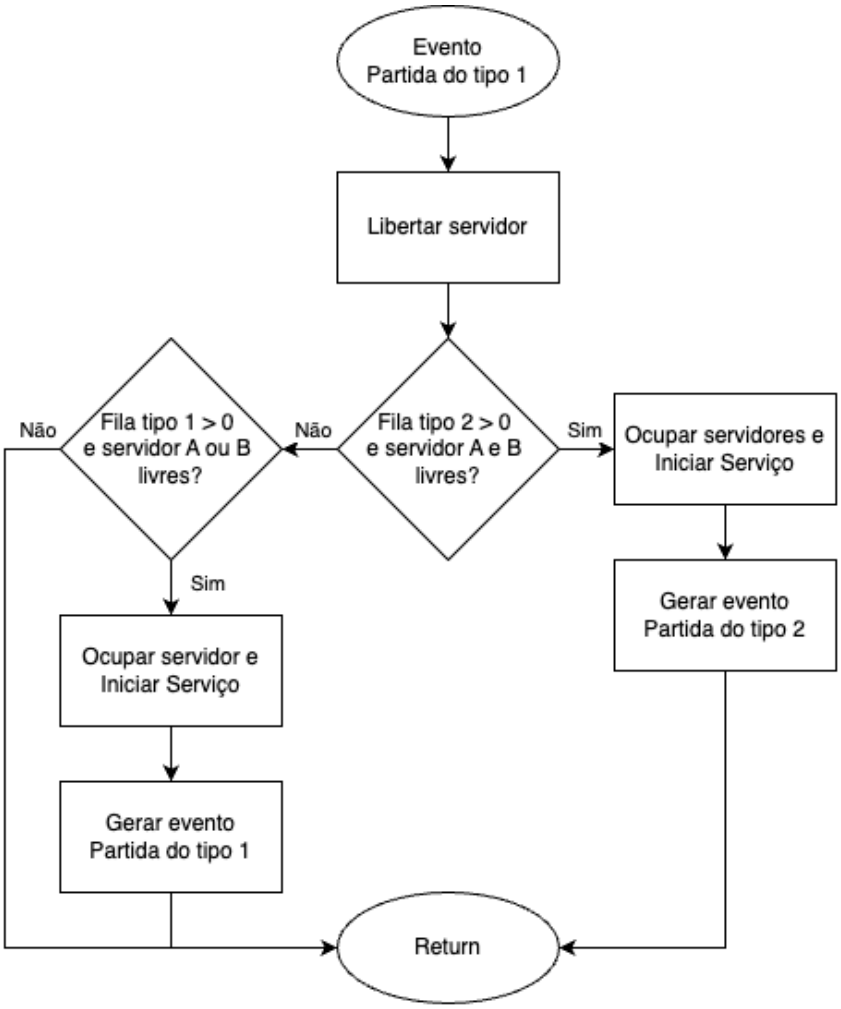


Figura 2.2: Fluxograma do evento de partida do tipo 1

A Figura 2.3 mostra o fluxograma do evento de partida do tipo 2. Este evento liberta os dois servidores utilizados (A e B) e verifica a existência de clientes do tipo 2 em espera. Se existirem, inicia-se um novo atendimento; caso contrário, tenta-se atender um cliente do tipo 1.

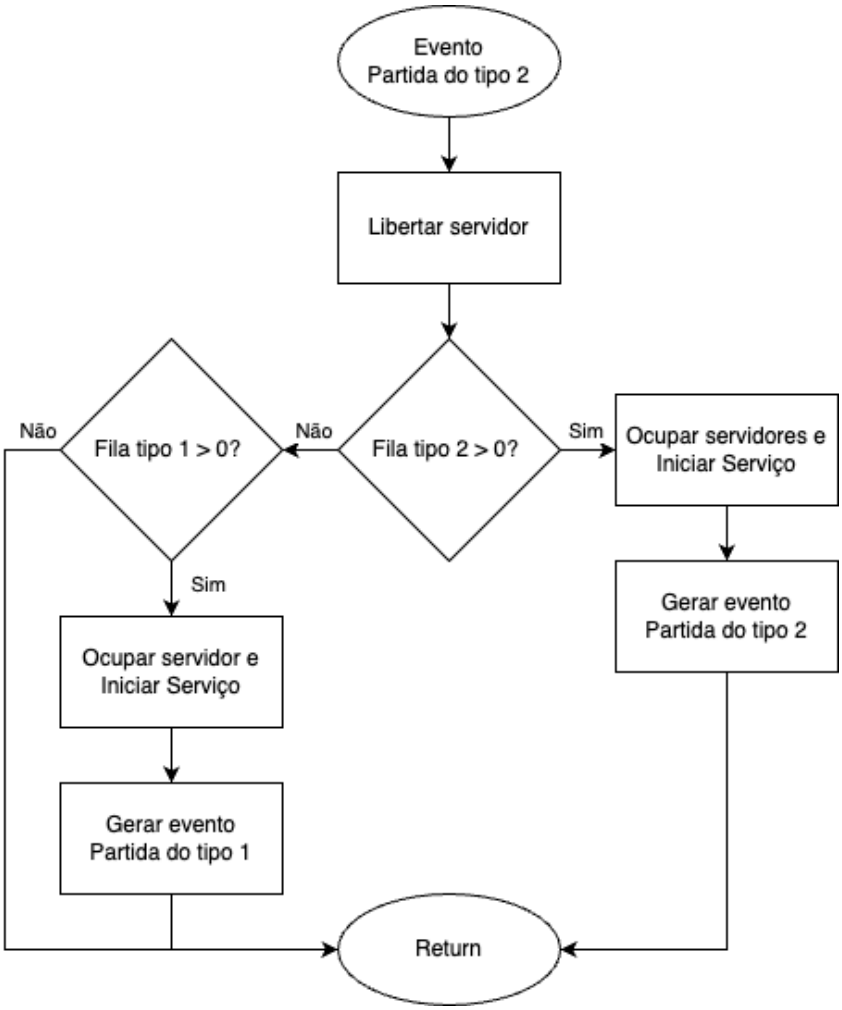


Figura 2.3: Fluxograma do evento de partida do tipo 2

Além dos fluxogramas, foi construído o grafo de eventos apresentado na Figura 2.4. Este grafo sintetiza as relações de agendamento entre eventos, evidenciando como a ocorrência de um evento pode originar outros. Os nós representam os eventos `Chegada`, `Partida do tipo 1` e `Partida do tipo 2`.

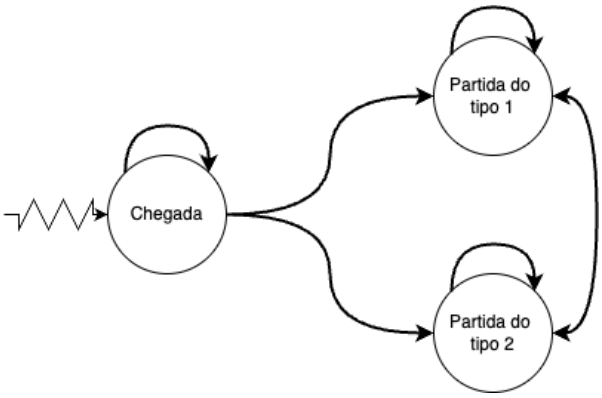


Figura 2.4: Grafo de eventos do sistema

No grafo de eventos:

- **Setas serrilhadas** representam eventos agendados no início da simulação, como a primeira chegada;

- **Setas grossas** indicam agendamentos com atraso positivo, como a partida de um cliente ou a próxima chegada.

Esta representação gráfica permite validar a lógica de execução do sistema e garantir que a modelação segue corretamente as boas práticas metodológicas da simulação de eventos discretos.

## 2.3 Metodologia de Implementação

Esta secção descreve os principais componentes implementados, com apoio em excertos de código que ilustram o funcionamento do modelo.

### 2.3.1 Inicialização e Parâmetros

Na fase inicial, foram definidos os parâmetros globais que controlam o comportamento da simulação, incluindo tempos médios, probabilidades e capacidades dos servidores. Estes parâmetros são centralizados no módulo `config.py` para facilitar a manutenção e experimentação.

```
MEAN_INTERARRIVAL = 1.0
MEAN_SERVICE_TYPE1 = 0.8
UNIF_SERVICE_TYPE2_MIN = 0.5
UNIF_SERVICE_TYPE2_MAX = 0.7
P_TYPE1 = 0.8
SIM_TIME = 1000.0
NUM_SERVERS_A = 2
NUM_SERVERS_B = 1
USE_SIMPY = False
VERBOSE = False
SEED = None
```

### 2.3.2 Gestão de Eventos e Lógica do Modelo

A simulação é controlada por uma lista de eventos futuros (`event_list`) organizada como uma fila de prioridade, utilizando o módulo `heapq`. A cada iteração do ciclo principal, o evento com menor tempo é extraído e processado, avançando o relógio de simulação para o tempo desse evento.

Os três tipos de eventos principais são:

- `arrival` – chegada de um novo cliente ao sistema;
- `departure_type1` – término do serviço de um cliente do tipo 1;
- `departure_type2` – término do serviço de um cliente do tipo 2.

O evento de `arrival` é responsável por duas ações: agendar a próxima chegada e encaminhar o cliente atual. O agendamento da chegada seguinte é feito com base numa variável exponencial, modelando um processo de Poisson com média `MEAN_INTERARRIVAL`:

```
interarrival = exponential(config.MEAN_INTERARRIVAL)
schedule_event(clock + interarrival, "arrival")
```

De seguida, o cliente é classificado como tipo 1 ou tipo 2 com base na variável `P_TYPE1`. Para clientes do tipo 1, tenta-se primeiro alocar um servidor do tipo A. Se não houver, tenta-se o tipo B. Se ambos estiverem ocupados, o cliente é adicionado à fila de espera `queue_type1`:

```
if random.random() < config.P_TYPE1:
    idx_A = find_free_server(servers_A)
    if idx_A is not None:
        serve_type1(idx_A, "A")
    else:
        idx_B = find_free_server(servers_B)
        if idx_B is not None:
            serve_type1(idx_B, "B")
        else:
            queue_type1.append(clock)
```

Para clientes do tipo 2, é necessário garantir a disponibilidade simultânea de um servidor A e de um servidor B. Caso isso não seja possível, o cliente é adicionado à fila `queue_type2`:

```
else:
    idx_A = find_free_server(servers_A)
    idx_B = find_free_server(servers_B)
    if idx_A is not None and idx_B is not None:
        serve_type2(idx_A, idx_B)
    else:
        queue_type2.append(clock)
```

A gestão das partidas dá prioridade a clientes tipo 2. No evento de `departure_type1`, após libertar o servidor, o sistema verifica se há cliente tipo 2 em fila e servidores A e B livres. Se for o caso, inicia-se o atendimento do cliente tipo 2. Caso contrário, tenta-se atender um cliente tipo 1, com base na disponibilidade do servidor libertado ou de outros.

O evento de `departure_type1` trata do término do atendimento de um cliente do tipo 1. O primeiro passo consiste na libertação do servidor que estava ocupado, identificado através do parâmetro `info`, que contém o tipo e o índice do servidor:

```
server_type, server_idx = info

if server_type == "A":
    servers_A[server_idx] = False
    server_A_type[server_idx] = None
else:
    servers_B[server_idx] = False
    server_B_type[server_idx] = None
```

Após libertar o servidor, o sistema tenta dar prioridade a um cliente do tipo 2. Para isso, verifica se existem clientes do tipo 2 em espera (`queue_type2`) e se há um servidor A e um servidor B livres. Caso estas condições se verifiquem, o cliente do tipo 2 é retirado da fila, e é iniciado o seu atendimento:

```
idx_A = server_idx if server_type == "A" else find_free_server(servers_A)
idx_B = server_idx if server_type == "B" else find_free_server(servers_B)
if idx_A is not None and idx_B is not None and try_serve_type2_from_queue(idx_A,
    idx_B):
    return
```

Se não for possível iniciar um atendimento do tipo 2, o sistema tenta atender um cliente do tipo 1 da fila correspondente, verificando se existe alguém em espera e se há um servidor disponível. O servidor A é sempre a primeira opção:

```
if idx_A is not None and try_serve_type1_from_queue(idx_A, "A"):
    return
if idx_B is not None and try_serve_type1_from_queue(idx_B, "B"):
    return
```

O evento de `departure_type2` trata do fim do serviço de um cliente do tipo 2, que ocupa simultaneamente um servidor do tipo A e um do tipo B. Os dois servidores são libertados:

```
server_idx_A, server_idx_B = indices

servers_A[server_idx_A] = False
server_A_type[server_idx_A] = None
servers_B[server_idx_B] = False
server_B_type[server_idx_B] = None
```

Tal como no evento anterior, a prioridade é dada aos clientes do tipo 2 que estejam em fila. Se existirem, o cliente é imediatamente atendido. Caso contrário, o sistema reutiliza o servidor do tipo A para atender um cliente do tipo 1:

```
if try_serve_type2_from_queue(server_idx_A, server_idx_B):
    return
if try_serve_type1_from_queue(server_idx_A, "A"):
    return
```

Não se tenta reutilizar o servidor B neste ponto, uma vez que os clientes do tipo 1 só usam o servidor B se não houver servidores A disponíveis, e, neste caso, um servidor do tipo A é libertado.

### 2.3.3 Funções de Atendimento

O início do atendimento aos clientes é encapsulado nas funções `serve_type1` e `serve_type2`, que definem os tempos de serviço e atualizam os estados dos servidores e acumuladores estatísticos. Estas funções são chamadas tanto no momento da chegada, se houver recursos disponíveis, como após o fim de um atendimento, para processar clientes em fila.

A função `serve_type1` inicia o atendimento de um cliente do tipo 1 no servidor indicado (A ou B), calcula o tempo de serviço a partir de uma distribuição exponencial e agenda o evento de partida correspondente:

```
def serve_type1(idx, server_type):
    service_time = exponential(config.MEAN_SERVICE_TYPE1)
    if server_type == "A":
        servers_A[idx] = True
        server_A_type[idx] = "type1"
        stats.server_A_time_type1[idx] += service_time
    else:
        servers_B[idx] = True
        server_B_type[idx] = "type1"
        stats.server_B_time_type1[idx] += service_time
    schedule_event(clock + service_time, "departure_type1", (server_type, idx))
```

```
12      return service_time
```

A função `serve_type2` realiza o mesmo processo para clientes do tipo 2, que requerem simultaneamente um servidor do tipo A e um do tipo B. O tempo de serviço é gerado por uma distribuição uniforme:

```
1  def serve_type2(idx_A, idx_B):
2      service_time = uniform(config.UNIF_SERVICE_TYPE2_MIN, config.
           UNIF_SERVICE_TYPE2_MAX)
3      servers_A[idx_A] = True
4      servers_B[idx_B] = True
5      server_A_type[idx_A] = "type2"
6      server_B_type[idx_B] = "type2"
7      stats.server_A_time_type2[idx_A] += service_time
8      stats.server_B_time_type2[idx_B] += service_time
9      schedule_event(clock + service_time, "departure_type2", (idx_A, idx_B))
10     return service_time
```

As funções auxiliares `try_serve_type1_from_queue` e `try_serve_type2_from_queue` verificam se há clientes em espera nas respetivas filas e, se possível, iniciam o atendimento usando os servidores indicados. Estas funções também calculam o tempo de espera e atualizam as estatísticas acumuladas, nomeadamente o atraso e o tempo total no sistema:

```
1  def try_serve_type1_from_queue(idx, server_type):
2      if queue_type1:
3          arrival_time = queue_type1.popleft()
4          delay = clock - arrival_time
5          stats.delays_type1.append(delay)
6          service_time = serve_type1(idx, server_type)
7          stats.waiting_times_type1.append(delay + service_time)
8          return True
9      return False
```

```
1  def try_serve_type2_from_queue(idx_A, idx_B):
2      if queue_type2:
3          arrival_time = queue_type2.popleft()
4          delay = clock - arrival_time
5          stats.delays_type2.append(delay)
6          service_time = serve_type2(idx_A, idx_B)
7          stats.waiting_times_type2.append(delay + service_time)
8          return True
9      return False
```

Esta estrutura modular permite reutilizar a lógica de atendimento em múltiplos pontos da simulação, assegurando consistência e clareza no encaminhamento das entidades.

### 2.3.4 Recolha de Estatísticas

As estatísticas do sistema são recolhidas e acumuladas ao longo do tempo. Por exemplo, o número médio de clientes em fila é obtido através da área sob a curva do comprimento da fila, dividida pelo tempo total simulado. A utilização dos servidores é calculada com base no tempo em que estiveram ocupados.

```python
mean_delay_type1 = sum(delays_type1) / len(delays_type1) if delays_type1 else 0
mean_delay_type2 = sum(delays_type2) / len(delays_type2) if delays_type2 else 0
mean_waiting_time_type1 = (
    sum(waiting_times_type1) / len(waiting_times_type1)
    if waiting_times_type1
    else 0
)
mean_waiting_time_type2 = (
    sum(waiting_times_type2) / len(waiting_times_type2)
    if waiting_times_type2
    else 0
)
mean_area_num_in_queue_type1 = area_num_in_queue_type1 / config.SIM_TIME
mean_area_num_in_queue_type2 = area_num_in_queue_type2 / config.SIM_TIME
mean_num_in_system_type1 = area_num_in_system_type1 / config.SIM_TIME
mean_num_in_system_type2 = area_num_in_system_type2 / config.SIM_TIME
```

Cada métrica corresponde a uma das definições estudadas em regime estacionário:

- **Tempo médio de espera em fila** por tipo de cliente, calculado como:

$$d = \lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} D_i$$

  onde $D_i$ representa o tempo de espera em fila do cliente $i$;

- **Tempo médio total no sistema** por tipo de cliente (fila + serviço), dado por:

$$w = \lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} W_i$$

  onde $W_i$ é o tempo total no sistema do cliente $i$;

- **Número médio de clientes em fila**, obtido por:

$$Q = \lim_{T\to\infty} \frac{1}{T} \int_0^T Q(t)\, dt$$

  onde $Q(t)$ representa o número de clientes em fila no instante $t$;

- **Número médio de clientes no sistema** (fila + serviço), calculado como:

$$L = \lim_{T\to\infty} \frac{1}{T} \int_0^T L(t)\, dt$$

  onde $L(t)$ é o número total de clientes no sistema no instante $t$;

- **Percentagem de utilização de cada servidor**, discriminada por tipo de cliente, obtida como a razão entre o tempo total de serviço prestado e o tempo total simulado.

### 2.3.5 Finalização e Impressão dos Resultados

No final da simulação, as estatísticas recolhidas são formatadas e apresentadas no ecrã. Os resultados incluem as métricas por tipo de cliente e por tipo de servidor, separando claramente as contribuições de cada classe para o desempenho global do sistema.

```
print(
    "\n---------------------------- Simulation Report ----------------------------"
)
print(
    f"Steady-state average delay - Type 1: {format_time(mean_delay_type1)}, Type 2: {format_time(mean_delay_type2)}"
)
print(
    f"Steady-state average waiting time - Type 1: {format_time(mean_waiting_time_type1)}, Type 2: {format_time(mean_waiting_time_type2)}"
)
print(
    f"Steady-state average number in queue - Type 1: {mean_area_num_in_queue_type1:.2f}, Type 2: {mean_area_num_in_queue_type2:.2f}"
)
print(
    f"Steady-state average number in system - Type 1: {mean_num_in_system_type1:.2f}, Type 2: {mean_num_in_system_type2:.2f}"
)

print("\nServer utilization:")
for i in range(len(server_A_time_type1)):
    perc1 = 100 * server_A_time_type1[i] / config.SIM_TIME
    perc2 = 100 * server_A_time_type2[i] / config.SIM_TIME
    print(f"\tServer A{i+1} - Type 1: {perc1:.2f}%, Type 2: {perc2:.2f}%")

for i in range(len(server_B_time_type1)):
    perc1 = 100 * server_B_time_type1[i] / config.SIM_TIME
    perc2 = 100 * server_B_time_type2[i] / config.SIM_TIME
    print(f"\tServer B{i+1} - Type 1: {perc1:.2f}%, Type 2: {perc2:.2f}%")
print(
    "--------------------------------------------------------------------------------\n"
)
```

Esta arquitetura modular promoveu não só a legibilidade e reutilização do código, como também facilitou a realização de experiências adicionais e a comparação entre diferentes configurações do sistema.

### 2.3.6 Implementação alternativa com `SimPy`

Adicionalmente, foi desenvolvida uma implementação alternativa do modelo utilizando a biblioteca `SimPy`, respeitando os mesmos princípios lógicos. Esta versão aproveita a abordagem orientada a processos disponibilizada pela biblioteca, mantendo a fidelidade ao comportamento especificado no enunciado.

## 2.4 Resultados

### 2.4.1 Médias Observadas na Configuração Base (2A+1B)

Para caracterizar o desempenho da configuração base, foi realizada uma série de 100 simulações independentes com diferentes *seeds*. Os valores apresentados a seguir representam as médias das principais métricas de desempenho recolhidas ao longo dessas execuções:

- **Tempo médio de espera em fila**:
  - Tipo 1: 0h 0m 21s

  - Tipo 2: 0h 0m 37s

- **Tempo médio total no sistema**:
  - Tipo 1: 0h 1m 10s
  - Tipo 2: 0h 1m 13s
- **Número médio de clientes em fila**:
  - Tipo 1: 0.02
  - Tipo 2: 0.03
- **Número médio de clientes no sistema**:
  - Tipo 1: 0.66
  - Tipo 2: 0.15
- **Taxa média de utilização dos servidores**:
  - Servidor A1 – Tipo 1: 37.25%, Tipo 2: 8.25%
  - Servidor A2 – Tipo 1: 20.72%, Tipo 2: 3.83%
  - Servidor B1 – Tipo 1: 6.08%, Tipo 2: 12.08%

Estes resultados mostram que, na configuração base, o sistema opera com baixas taxas de ocupação para os servidores do tipo B, sendo a carga mais acentuada nos servidores A, especialmente o primeiro. O tempo médio de espera em fila é reduzido para ambos os tipos de cliente, refletindo uma utilização eficiente dos recursos nesta configuração.

### 2.4.2 Resposta ao Exercício 1.2

De forma a avaliar o impacte da alteração da configuração do sistema, foi realizada uma experiência comparativa com três cenários distintos:

- **2A+1B** – configuração base;
- **3A+1B** – adição de um servidor do tipo A;
- **2A+2B** – adição de um servidor do tipo B.

Para cada configuração, a simulação foi repetida 100 vezes com diferentes *seeds*, tendo sido calculado, em cada repetição, o atraso médio em fila para ambos os tipos de cliente. A métrica principal analisada foi o máximo dos atrasos médios para os dois tipos de cliente, refletindo o pior desempenho registado em cada cenário.

A Figura 2.5 resume os resultados obtidos. Verifica-se que a configuração `3A+1B` proporciona uma redução significativa do atraso médio para clientes do tipo 2, sem penalizar os do tipo 1. Já a configuração `2A+2B` apresenta melhorias equilibradas para ambos os tipos, embora menos acentuadas para clientes do tipo 2.

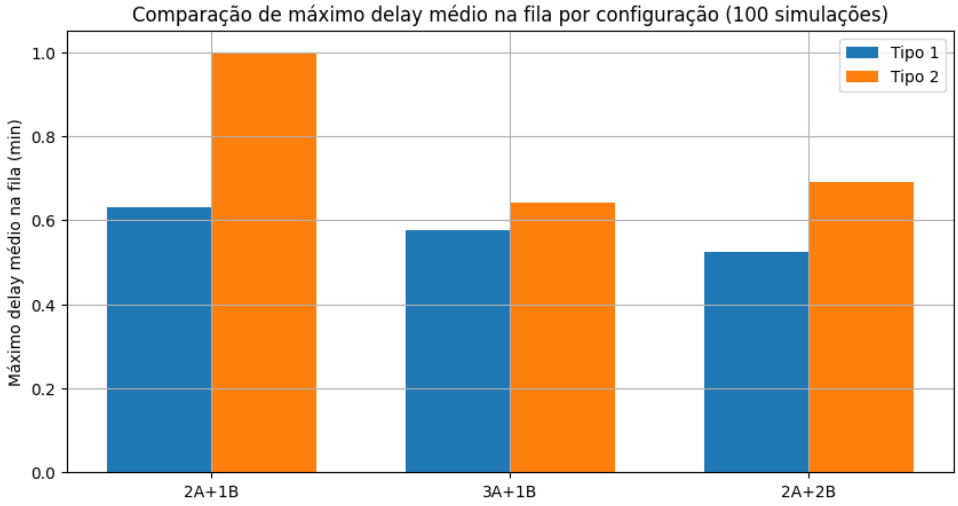


Figura 2.5: Comparação do máximo do atraso médio em fila por tipo de cliente e por configuração (100 simulações).

A análise permite responder diretamente à questão colocada no exercício 1.2:

> *"Check which one is better in reducing the maximum of the average delay in queue for both types of customers."*

Constata-se que a adição de um servidor do tipo A (`3A+1B`) conduz à maior redução do valor máximo entre os atrasos médios em fila, comparativamente com a adição de um servidor do tipo B. Esta configuração reduz de forma mais eficaz o ponto crítico observado nos clientes do tipo 2, que requerem simultaneamente servidores A e B. Importa ainda sublinhar que, na configuração `3A+1B`, os atrasos médios em fila máximos para ambos os tipos de cliente são inferiores ao atraso máximo médio do cliente do tipo 2 na configuração `2A+2B`, o que evidencia uma melhoria mais equilibrada e global do desempenho do sistema. Assim, segundo o critério estabelecido, a configuração `3A+1B` é claramente a mais vantajosa.

### 2.4.3 Validação do Modelo

A validade do modelo foi verificada através de diversos testes de consistência e plausibilidade. Em primeiro lugar, confirmou-se a conservação do fluxo de entidades, garantindo que o número de clientes que estiveram no sistema corresponde ao número de chegadas simuladas, sem perdas nem duplicações. Além disso, foi observada a convergência dos atrasos médios para valores próximos de zero em condições de baixa carga, ou seja, quando o tempo médio entre chegadas é significativamente superior ao tempo médio de serviço, um comportamento esperado num sistema bem dimensionado.

Estas verificações permitiram confirmar que a implementação reproduz corretamente a lógica do sistema modelado, assegurando coerência nos resultados. Importa ainda referir que, se estivessem disponíveis dados teóricos ou resultados de referência para este sistema, seria possível realizar comparações adicionais com esses valores, reforçando a validação do modelo através de análise de aderência quantitativa.

### 2.4.4 Justificação do Número de Simulações

As experiências foram realizadas com 100 simulações com diferentes *seeds* para cada configuração testada, número escolhido com o objetivo de reduzir a variabilidade estatística e garantir estimativas mais robustas das métricas de desempenho. Este volume de replicações permite efetuar comparações fiáveis entre diferentes configurações do sistema com base em médias estáveis. Além disso, possibilita

analisar a variabilidade das métricas, como o máximo dos atrasos médios por tipo de cliente, o que é particularmente relevante em avaliações comparativas. Este nível de replicação está de acordo com as boas práticas metodológicas da simulação estocástica, uma vez que uma única simulação corresponderia apenas a uma realização aleatória do processo e não permitiria inferências estatísticas fiáveis.

## 2.5 Conclusão

Através da simulação de eventos discretos, foi possível modelar e analisar de forma rigorosa o comportamento de um sistema de atendimento com múltiplos tipos de clientes e recursos diferenciados. A implementação permitiu capturar as particularidades de cada tipo de entidade e a complexidade da sua interação com os recursos disponíveis, nomeadamente a necessidade de atendimento simultâneo para os clientes do tipo 2.

A análise dos resultados obtidos para a configuração base demonstrou um desempenho estável, com baixos tempos médios de espera e utilização equilibrada dos servidores. A experimentação com diferentes configurações revelou que a adição de um servidor do tipo A (`3A+1B`) é a estratégia mais eficaz para reduzir o pior caso dos atrasos médios em fila, especialmente beneficiando os clientes do tipo 2 sem comprometer o desempenho para os restantes.

O modelo implementado cumpriu os objetivos propostos, demonstrando capacidade para apoiar decisões sobre alocação de recursos em sistemas reais de atendimento. Além disso, o processo de validação e replicação assegurou a fiabilidade dos resultados, em conformidade com as boas práticas metodológicas da simulação estocástica. Este exercício destacou, assim, o valor da simulação como ferramenta de apoio à decisão em contextos onde a análise analítica seria difícil ou impraticável.

# Capítulo 3

# Exercício 2

## 3.1 Introdução

O estudo da dinâmica de populações, nomeadamente a interação entre espécies predadoras e presas, constitui uma aplicação clássica da modelação e simulação de sistemas dinâmicos contínuos. Estes sistemas são geralmente descritos por equações diferenciais que representam a evolução de variáveis dependentes do tempo, como o número de indivíduos de cada espécie. Quando não é possível resolver estas equações de forma analítica, recorre-se a métodos numéricos de integração como alternativa eficaz.

Neste projeto, pretende-se simular a evolução temporal de duas populações interdependentes: presas $x(t)$ e predadores $y(t)$, segundo o modelo clássico de Lotka-Volterra adaptado. As equações diferenciais que descrevem a evolução destas populações são:

$$\frac{dx(t)}{dt} = \alpha \cdot x(t) - \beta \cdot x(t) \cdot y(t)$$
$$\frac{dy(t)}{dt} = \delta \cdot x(t) \cdot y(t) - \gamma \cdot y(t)$$

onde:

- $x(t)$ representa o número de presas no instante $t$;
- $y(t)$ representa o número de predadores no instante $t$;
- $\alpha$ é a taxa de crescimento natural das presas na ausência de predadores;
- $\beta$ é a taxa de predação (interação entre espécies);
- $\delta$ é a taxa com que as presas consumidas contribuem para o aumento da população de predadores;
- $\gamma$ é a taxa de mortalidade natural dos predadores.

A resolução destas equações será efetuada recorrendo a dois métodos numéricos de integração:

1. **Método de Euler (Forward Euler)**: método explícito simples, baseado numa aproximação linear da derivada, adequado para simulações rápidas e com complexidade reduzida;
2. **Método de Runge-Kutta de quarta ordem (RK4)**: método mais preciso e estável, que considera múltiplas avaliações da derivada por passo temporal para melhorar a estimativa da solução.

Foi desenvolvido um programa de simulação que implementa ambas as abordagens, permitindo ao utilizador escolher o método desejado ou executar ambos para comparação. Os parâmetros do modelo ($x(0)$, $y(0)$, $\alpha$, $\beta$, $\delta$, $\gamma$, $\Delta t$, $t_{\text{final}}$) são totalmente configuráveis, podendo ser introduzidos através da linha de comandos ou de um ficheiro (`config.py`).

Por fim, será realizada uma análise comparativa entre os métodos de Euler e Runge-Kutta, com o objetivo de avaliar a precisão numérica de cada um, identificar divergências nas soluções obtidas e discutir a adequação de cada técnica ao contexto da simulação de populações biológicas.

## 3.2 Fundamentos Teóricos da Aproximação Numérica

Na simulação da evolução das populações de presas e predadores segundo o modelo de Lotka-Volterra adaptado, é necessário resolver um sistema de equações diferenciais ordinárias (EDO) de primeira ordem. Estas equações descrevem a variação contínua no tempo do número de indivíduos de cada espécie, em função das suas interações ecológicas.

De acordo com a classificação lecionada na unidade curricular, o modelo de Lotka-Volterra adaptado corresponde a um sistema contínuo no tempo, pois o seu estado evolui de forma contínua, sendo descrito por equações diferenciais da forma genérica:

$$\frac{dx(t)}{dt} = F_x(x(t), y(t), t) = \alpha \cdot x(t) - \beta \cdot x(t) \cdot y(t), \quad \frac{dy(t)}{dt} = F_y(x(t), y(t), t) = \delta \cdot x(t) \cdot y(t) - \gamma \cdot y(t)$$

Trata-se de um sistema de primeira ordem, dado que apenas envolve derivadas de primeira ordem das variáveis de estado $x(t)$ e $y(t)$, e é não linear, uma vez que contém produtos entre variáveis, como $x(t) \cdot y(t)$, que não podem ser escritos como combinações lineares simples.

Adicionalmente, é um sistema autónomo, pois as equações não dependem explicitamente do tempo $t$, apenas do estado atual do sistema.

Embora em casos particulares estas equações possam ser resolvidas analiticamente, em situações reais como esta recorre-se geralmente à integração numérica. Este processo consiste em estimar a evolução das variáveis ao longo do tempo através de métodos iterativos, que aproximam a solução contínua por uma sequência discreta de valores, a partir de condições iniciais e parâmetros conhecidos.

### 3.2.1 Método de Euler (Forward Euler)

O método de Euler é uma das abordagens mais simples para a integração numérica de EDOs. Baseia-se numa aproximação linear da derivada, utilizando o valor atual da função:

$$x(t + \Delta t) \approx x(t) + \Delta t \cdot F_x(x(t), y(t), t), \quad y(t + \Delta t) \approx y(t) + \Delta t \cdot F_y(x(t), y(t), t)$$

Apesar da sua simplicidade e baixo custo computacional, o método de Euler é apenas condicionalmente estável e pode introduzir erros de truncamento consideráveis se o passo temporal $\Delta t$ não for suficientemente pequeno. O erro global acumulado é da ordem de $O(\Delta t)$.

### 3.2.2 Método de Runge-Kutta de Quarta Ordem (RK4)

O método de Runge-Kutta de quarta ordem (RK4) é amplamente utilizado para resolver EDOs com maior precisão. A sua principal vantagem face ao método de Euler reside no uso de múltiplas avaliações da derivada por passo temporal, resultando numa estimativa mais precisa da evolução das variáveis.

Aplicando este método ao sistema de Lotka-Volterra adaptado, consideram-se as funções $F_x(x, y, t)$ e $F_y(x, y, t)$, mesmo que neste caso o tempo $t$ não surja explicitamente nas equações.

Os coeficientes intermédios são definidos da seguinte forma:

$$K_{1x} = \Delta t \cdot F_x(x(t), y(t), t), \qquad K_{1y} = \Delta t \cdot F_y(x(t), y(t), t)$$

$$K_{2x} = \Delta t \cdot F_x\left(x(t) + \frac{K_{1x}}{2},\ y(t) + \frac{K_{1y}}{2},\ t + \frac{\Delta t}{2}\right), \qquad K_{2y} = \Delta t \cdot F_y\left(x(t) + \frac{K_{1x}}{2},\ y(t) + \frac{K_{1y}}{2},\ t + \frac{\Delta t}{2}\right)$$

$$K_{3x} = \Delta t \cdot F_x\left(x(t) + \frac{K_{2x}}{2},\ y(t) + \frac{K_{2y}}{2},\ t + \frac{\Delta t}{2}\right), \qquad K_{3y} = \Delta t \cdot F_y\left(x(t) + \frac{K_{2x}}{2},\ y(t) + \frac{K_{2y}}{2},\ t + \frac{\Delta t}{2}\right)$$

$$K_{4x} = \Delta t \cdot F_x(x(t) + K_{3x},\ y(t) + K_{3y},\ t + \Delta t), \qquad K_{4y} = \Delta t \cdot F_y(x(t) + K_{3x},\ y(t) + K_{3y},\ t + \Delta t)$$

A atualização final das variáveis é dada por:

$$x(t + \Delta t) \approx x(t) + \frac{K_{1x} + 2K_{2x} + 2K_{3x} + K_{4x}}{6}, \quad y(t + \Delta t) \approx y(t) + \frac{K_{1y} + 2K_{2y} + 2K_{3y} + K_{4y}}{6}$$

Este método tem erro local de ordem $O(\Delta t^5)$ e erro global acumulado de ordem $O(\Delta t^4)$, sendo particularmente eficaz em simulações que exigem elevada precisão e estabilidade.

## 3.3 Metodologia de Implementação

A arquitetura do programa segue o ciclo clássico de simulação abordado na unidade curricular: `initialize()`, `update()` e `observe()`.

### 3.3.1 Funções diferenciais do modelo

As equações diferenciais que regem a dinâmica das populações são implementadas nas funções `dx()` e `dy()`. Estas representam, respetivamente, a variação temporal da população de presas $x(t)$ e de predadores $y(t)$, segundo o modelo de Lotka-Volterra adaptado:

$$\frac{dx}{dt} = \alpha x - \beta xy, \quad \frac{dy}{dt} = \delta xy - \gamma y$$

Em `Python`, estas funções foram definidas da seguinte forma:

```
def dx(x, y, alpha, beta):
    return alpha * x - beta * x * y

def dy(x, y, delta, gamma):
    return delta * x * y - gamma * y
```

Estas funções são utilizadas em ambos os métodos de integração, garantindo a separação entre o modelo matemático e os algoritmos numéricos que o resolvem.

### 3.3.2 `initialize()`

A função `initialize()` é responsável pela configuração do estado inicial da simulação. Recebe como argumentos os valores iniciais da população de presas $x_0$ e predadores $y_0$, e devolve os vetores de armazenamento das variáveis ao longo do tempo:

```
def initialize(x0, y0):
    t = 0.0
    times = [t]
    xs = [x0]
    ys = [y0]
    return t, x0, y0, times, xs, ys
```

### 3.3.3 observe()

Durante a simulação, a função `observe()` regista os valores de $t$, $x(t)$ e $y(t)$ a cada iteração, permitindo reconstruir a trajetória temporal de ambas as populações:

```
def observe(t, x, y, times, xs, ys):
    times.append(t)
    xs.append(x)
    ys.append(y)
```

### 3.3.4 update()

#### Método de Euler

A função `update_euler()` implementa o método de Euler para a atualização do estado do sistema. A derivada de cada variável é avaliada no instante atual e utilizada para estimar o próximo valor:

```
def update_euler(x, y, alpha, beta, delta, gamma, dt):
    x_new = x + dx(x, y, alpha, beta) * dt
    y_new = y + dy(x, y, delta, gamma) * dt
    return x_new, y_new
```

#### Método de Runge-Kutta de Quarta Ordem (RK4)

A função `update_rk4()` implementa o método RK4, utilizando quatro avaliações sucessivas da função derivada em pontos intermédios para melhorar a estimativa final:

```
def update_rk4(x, y, alpha, beta, delta, gamma, dt):
    k1x = dt * dx(x, y, alpha, beta)
    k1y = dt * dy(x, y, delta, gamma)

    k2x = dt * dx(x + k1x / 2, y + k1y / 2, alpha, beta)
    k2y = dt * dy(x + k1x / 2, y + k1y / 2, delta, gamma)

    k3x = dt * dx(x + k2x / 2, y + k2y / 2, alpha, beta)
    k3y = dt * dy(x + k2x / 2, y + k2y / 2, delta, gamma)

    k4x = dt * dx(x + k3x, y + k3y, alpha, beta)
    k4y = dt * dy(x + k3x, y + k3y, delta, gamma)

    x_new = x + (k1x + 2 * k2x + 2 * k3x + k4x) / 6
    y_new = y + (k1y + 2 * k2y + 2 * k3y + k4y) / 6
    return x_new, y_new
```

### 3.3.5 Função `simulate()`

A função `simulate()` encapsula o ciclo completo de simulação, permitindo escolher entre os métodos `euler` ou `rk4`:

```
def simulate(x0, y0, alpha, beta, delta, gamma, dt, t_final, method):
    t, x, y, times, xs, ys = initialize(x0, y0)

    while t < t_final:
        if method == "euler":
            x, y = update_euler(x, y, alpha, beta, delta, gamma, dt)
        elif method == "rk4":
            x, y = update_rk4(x, y, alpha, beta, delta, gamma, dt)
        else:
            raise ValueError("Metodo invalido. Use 'euler' ou 'rk4'.")
        t += dt
        observe(t, x, y, times, xs, ys)

    return times, xs, ys
```

Esta organização modular permite realizar simulações flexíveis e facilmente comparáveis entre métodos, com possibilidade de visualização posterior dos resultados num *jupyter notebook* ou ambiente gráfico.

## 3.4 Resultados

Nesta secção comparam-se os métodos de Euler e de Runge-Kutta de quarta ordem (RK4) na simulação do sistema de Lotka-Volterra adaptado. A análise considera a evolução temporal, o comportamento no espaço de fases, o erro absoluto, a estabilidade com diferentes passos temporais, a conservação da função de Lyapunov e o custo computacional.

Todos os métodos foram comparados com uma solução de referência gerada com RK4 e um passo temporal muito pequeno ($\Delta t = 0.001$), de forma a garantir alta precisão.

### 3.4.1 Evolução Temporal das Populações

Na Figura 3.1, apresenta-se a evolução das populações de presas $x(t)$ e predadores $y(t)$. O método de Euler apresenta desvios crescentes da solução de referência ao longo do tempo, enquanto o RK4 mantém-se muito próximo da solução esperada.

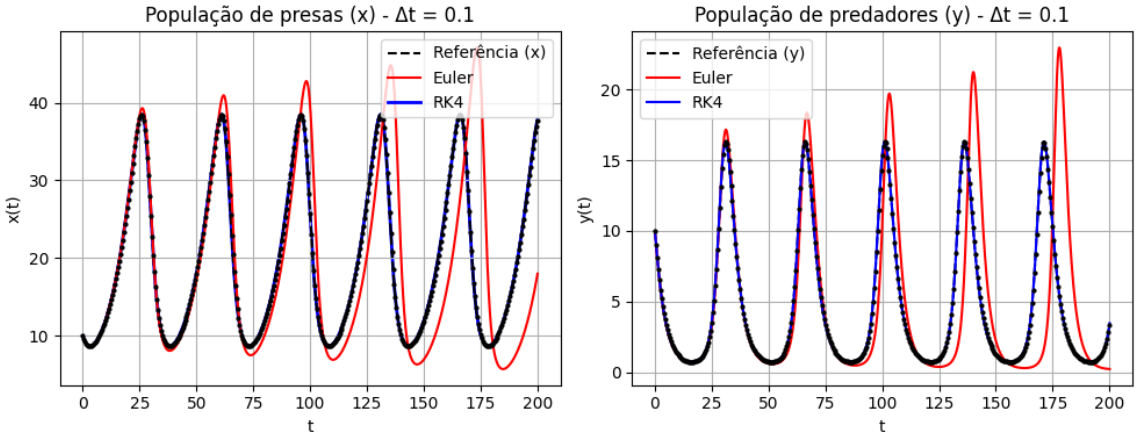


Figura 3.1: Evolução temporal das populações de presas e predadores.

### 3.4.2 Diagrama de Fase

A Figura 3.2 mostra o comportamento cíclico no espaço de fases. O RK4 segue trajetórias fechadas, em linha com a conservação energética do sistema. Já o método de Euler tende a amplificar a trajetória ao longo do tempo, indicando perda de estabilidade.

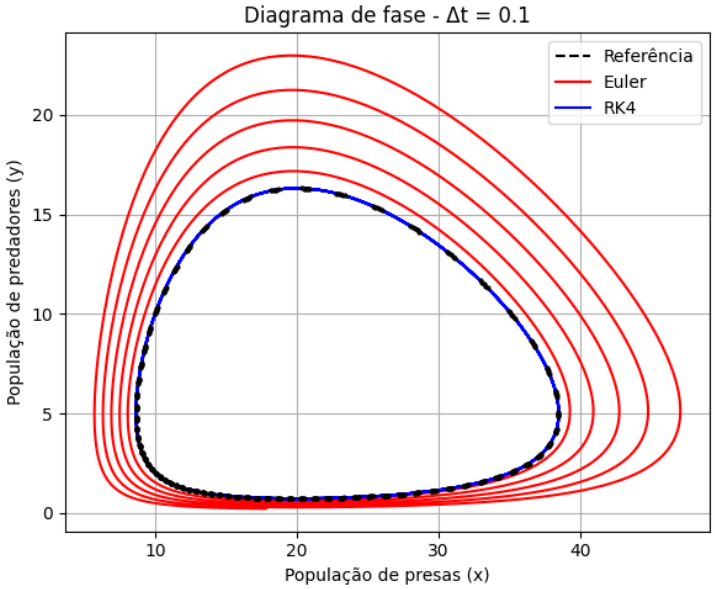


Figura 3.2: Diagrama de fase das populações $x(t)$ e $y(t)$.

### 3.4.3 Erro Absoluto ao Longo do Tempo

A Figura 3.3 representa o erro absoluto acumulado ao longo do tempo, calculado como:

$$\text{Erro total}(t) = |x_{\text{método}}(t) - x_{\text{ref}}(t)| + |y_{\text{método}}(t) - y_{\text{ref}}(t)|$$

Este erro foi avaliado ponto a ponto após interpolação dos valores de referência. O RK4 apresenta erros cerca de $10^6$ vezes inferiores aos do método de Euler.

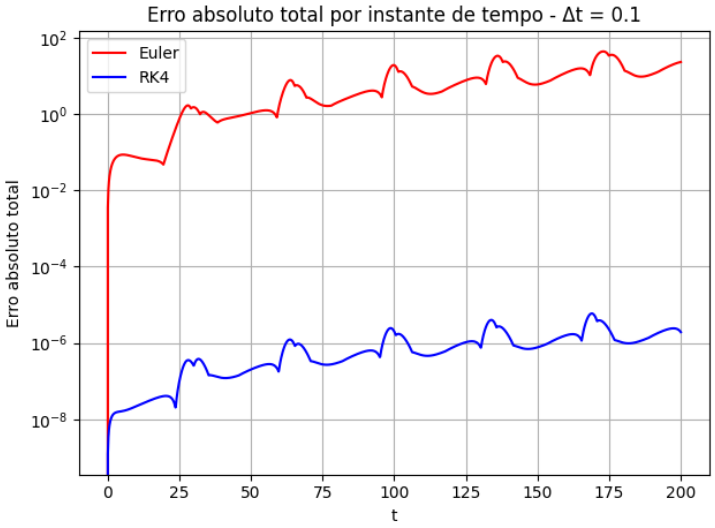


Figura 3.3: Erro absoluto total por instante de tempo.

### 3.4.4 Erro Médio vs Passo Temporal

A Figura 3.4 ilustra a evolução do erro médio absoluto com o tamanho do passo temporal, utilizando escala log-log. O erro médio foi definido como:

$$\text{Erro médio} = \frac{1}{N} \sum_{i=1}^{N} \left( |x_i - x_{\text{ref},i}| + |y_i - y_{\text{ref},i}| \right)$$

O RK4 mostra uma taxa de convergência muito superior, confirmando a sua ordem 4. O método de Euler exige passos muito pequenos para atingir erros aceitáveis.

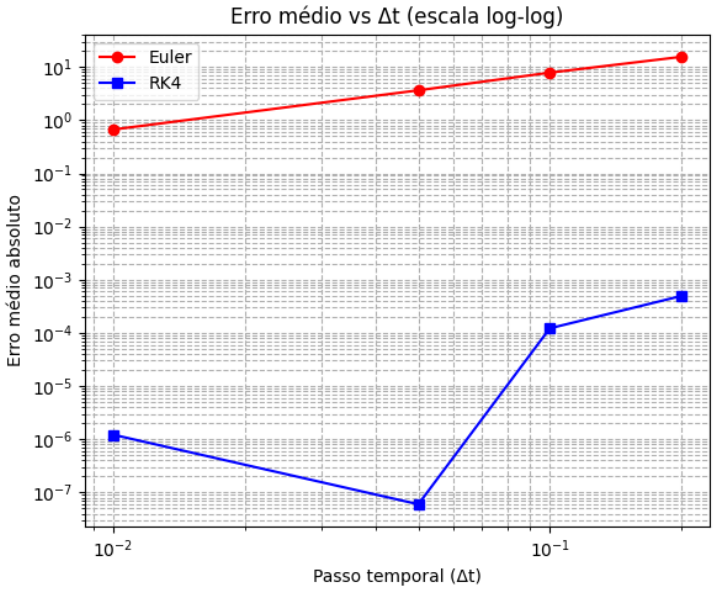


Figura 3.4: Erro médio absoluto vs $\Delta t$ (escala log-log).

### 3.4.5 Estabilidade dos Métodos

A estabilidade numérica foi estudada aplicando diferentes valores de $\Delta t$. A Figura 3.5 mostra que Euler perde rapidamente estabilidade para passos maiores. Por contraste, o RK4 mantém a fidelidade da solução para todos os $\Delta t$ testados, conforme a Figura 3.6.

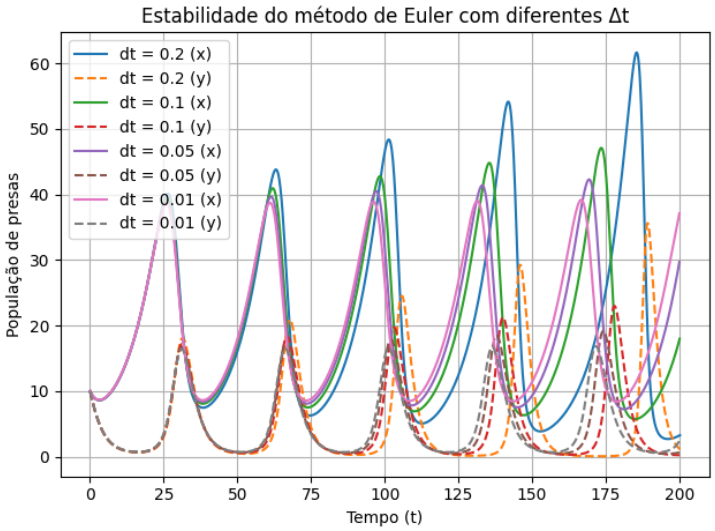


Figura 3.5: Estabilidade do método de Euler com diferentes $\Delta t$.

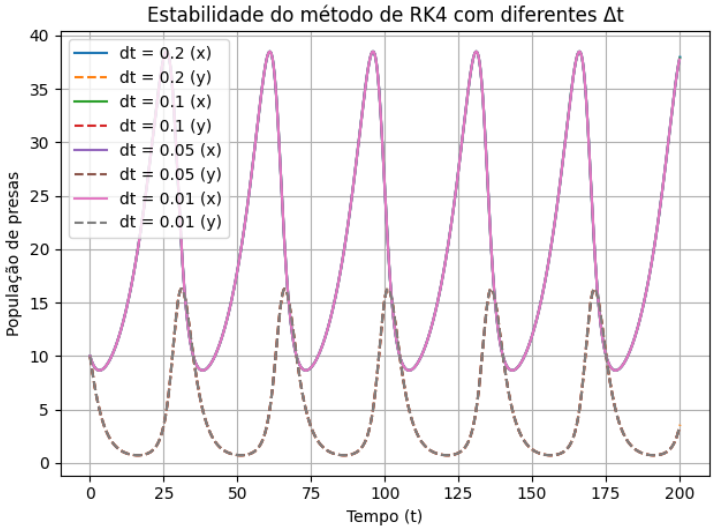


Figura 3.6: Estabilidade do método RK4 com diferentes $\Delta t$.

### 3.4.6 Conservação da Energia (Função de Lyapunov)

Foi calculada a função de Lyapunov $H(x, y)$ associada ao sistema, que idealmente deveria ser constante ao longo do tempo:

$$H(x, y) = \delta x - \gamma \ln x + \beta y - \alpha \ln y$$

Na Figura 3.7, observa-se que apenas o método RK4 preserva bem esta quantidade, enquanto o método de Euler apresenta deriva significativa.

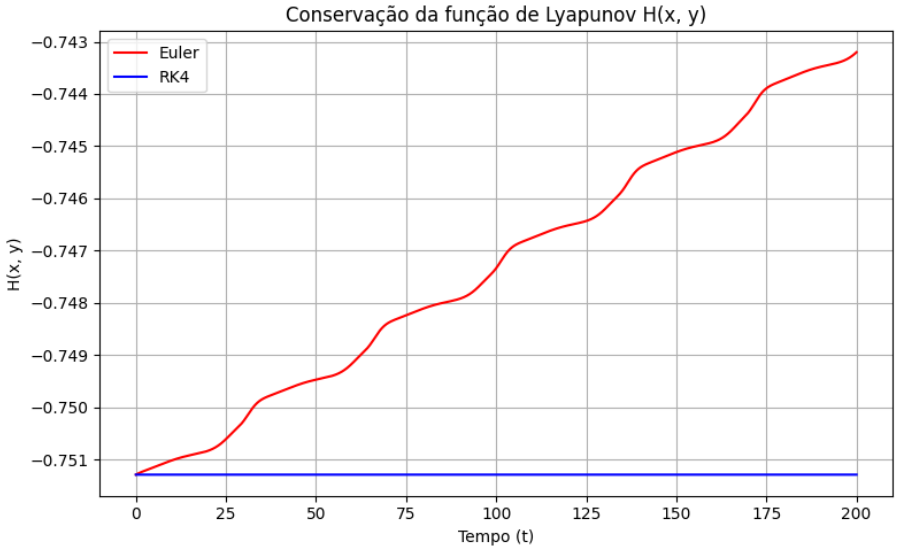


Figura 3.7: Conservação da função de Lyapunov $H(x, y)$.

### 3.4.7 Custo Computacional

A Figura 3.8 compara o tempo de execução dos métodos. Embora o RK4 seja mais exigente, o seu custo é compensado pelo aumento significativo na precisão.

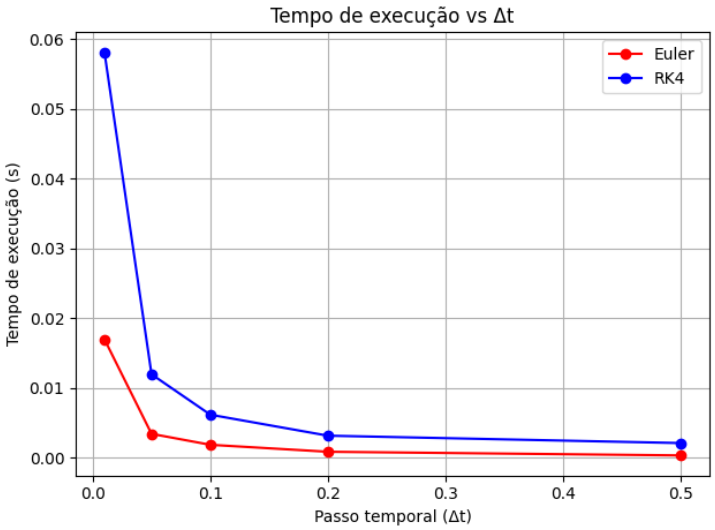


Figura 3.8: Tempo de execução vs passo temporal $\Delta t$.

### 3.4.8 Resposta ao Exercício 2.3

A análise permite responder diretamente à questão colocada no exercício 2.3:

> *"Compare the precision of the previous approaches."*

A comparação entre os métodos de Euler e de Runge-Kutta de quarta ordem (RK4) mostra diferenças significativas em termos de precisão:

- **Erro absoluto:** O método RK4 apresenta erros acumulados várias ordens de magnitude inferiores ao método de Euler, conforme evidenciado nas Figuras 3.3 e 3.4. Esta diferença é consistente ao longo do tempo e torna-se mais acentuada para passos temporais maiores.
- **Estabilidade:** O método de Euler evidencia problemas de estabilidade à medida que o passo temporal $\Delta t$ aumenta. Como se observa na Figura 3.5, para valores como $\Delta t = 0.2$, as trajetórias oscilam de forma irrealista e divergem da solução teórica. Isto indica que o método de Euler é apenas condicionalmente estável. Por contraste, o método de Runge-Kutta de quarta ordem (RK4) mostra-se estável mesmo com passos temporais maiores, mantendo a forma periódica correta das trajetórias, como se verifica na Figura 3.6.
- **Conservação de invariantes:** A função de Lyapunov $H(x, y)$, que idealmente deveria manter-se constante ao longo da simulação, é bem preservada pelo RK4 mas apresenta deriva crescente com o método de Euler (Figura 3.7).
- **Eficiência:** Embora o método RK4 exija mais cálculos por passo (quatro avaliações da derivada por iteração), o seu desempenho em termos de erro por intervalos de tempo (Figura 3.4) revela-se muito superior. Assim, em termos de precisão por custo computacional, o RK4 é claramente mais eficiente.

Conclui-se, portanto, que o método RK4 é consideravelmente mais preciso e robusto do que o método de Euler para este tipo de sistema não linear e oscilatório. O método de Euler pode ser usado, mas a custo de precisão limitada e maior instabilidade.

## 3.5 Conclusão

O estudo do sistema de Lotka-Volterra adaptado permitiu explorar e comparar dois métodos numéricos fundamentais de integração de equações diferenciais: o método de Euler e o método de Runge-Kutta de quarta ordem (RK4). Através da análise gráfica e quantitativa dos resultados, foi possível tirar conclusões claras quanto à precisão, estabilidade e eficiência de ambos os métodos.

O método de Euler, apesar da sua simplicidade e baixo custo computacional por iteração, revelou-se inadequado para simulações de longo prazo ou com passos temporais moderados, devido à sua fraca estabilidade e acumulação de erro. A sua precisão só se torna aceitável com passos extremamente pequenos, o que compromete a sua eficiência global.

Por outro lado, o método RK4 demonstrou elevada precisão mesmo com passos relativamente grandes, mantendo-se estável e fiel à dinâmica cíclica do sistema. Foi o único método capaz de preservar adequadamente a função de Lyapunov do sistema, evidenciando o seu bom comportamento na conservação de invariantes.

Adicionalmente, verificou-se que, embora mais exigente em termos de cálculos por iteração, o RK4 compensa esse custo com uma taxa de convergência significativamente superior. Isto traduz-se numa menor necessidade de passos para atingir um determinado nível de precisão, tornando-o mais eficiente para simulações rigorosas.

Em suma, para problemas contínuos no tempo com comportamento oscilatório e sensibilidade a erros numéricos, como o sistema de predador-presa, o método de Runge-Kutta de quarta ordem representa uma escolha claramente superior ao método de Euler. A análise realizada demonstra a importância de selecionar o método de integração adequado, tendo em conta não só a complexidade computacional mas também as exigências de estabilidade e fidelidade da simulação.

# Capítulo 4

# Conclusão

A realização deste projeto permitiu aplicar, de forma prática e fundamentada, os principais conceitos e metodologias da unidade curricular de Simulação e Otimização, através da resolução de dois exercícios com naturezas distintas, um focado na simulação de eventos discretos e outro na simulação de sistemas contínuos no tempo.

No **Exercício 1**, foi desenvolvido um modelo de atendimento com múltiplos tipos de clientes e servidores, recorrendo à Simulação de Eventos Discretos (DES). A implementação manual dos principais componentes da simulação (eventos, filas, servidores e lógica de decisão) permitiu explorar com profundidade a complexidade inerente à alocação de recursos em sistemas com restrições de simultaneidade. A recolha e análise estatística dos resultados possibilitou uma avaliação rigorosa do desempenho do sistema, evidenciando a importância de métricas como o tempo médio de espera, o número médio de clientes em fila e a taxa de utilização dos servidores. A experiência comparativa entre configurações demonstrou que a adição de um servidor do tipo A contribui mais eficazmente para a redução do maior atraso médio, beneficiando sobretudo os clientes do tipo 2, sem comprometer a qualidade de serviço para os restantes.

No **Exercício 2**, foi estudado o sistema dinâmico contínuo de Lotka-Volterra adaptado, representando a interação entre populações de presas e predadores. Recorrendo a integração numérica, compararam-se dois métodos, Euler e Runge-Kutta de quarta ordem (RK4), quanto à sua precisão, estabilidade e eficiência computacional. A análise demonstrou que, apesar da simplicidade do método de Euler, este apresenta limitações graves em termos de estabilidade e conservação de invariantes, especialmente para passos temporais moderados. Por contraste, o método RK4 revelou elevada robustez e precisão mesmo com passos maiores, mantendo a consistência da solução ao longo do tempo. Foram analisadas métricas como o erro absoluto, a conservação da função de Lyapunov, o tempo de execução e o número de passos, permitindo fundamentar a escolha do método mais adequado em cada contexto.

Em ambos os exercícios, foi seguida uma abordagem metodológica rigorosa, com atenção especial à validação dos modelos, à replicação de experiências para garantir robustez estatística e à visualização clara dos resultados. A modularidade das implementações facilitou a experimentação e comparação entre alternativas, promovendo boas práticas aplicadas à simulação.

Em suma, este projeto reforçou a utilidade da simulação como ferramenta de análise de sistemas complexos, quer estocásticos e discretos, quer determinísticos e contínuos, e consolidou competências técnicas essenciais como modelação, análise estatística e avaliação comparativa de desempenho. Os conhecimentos adquiridos são aplicáveis a uma vasta gama de problemas reais, com impacte na tomada de decisões informadas em engenharia e ciência dos sistemas.