Relatório de Miniprojeto: Cadeias de Markov a Tempo Contínuo - CTMC

Aplicação: Controle de Admissão

Arthur Cadore Matuella Barcella
Deivid Fortunato Frederico

20 de Maio de 2025

Engenharia de Telecomunicações - IFSC-SJ

Sumário

1. Introdução

O presente relatório tem como objetivo apresentar o desenvolvimento de um miniprojeto na disciplina de Engenharia de Telecomunicações, com foco na modelagem e análise de sistemas utilizando Cadeias de Markov a Tempo Contínuo (CTMC). O projeto foi realizado por Arthur Cadore Matuella Barcella e Deivid Fortunato Frederico, alunos do curso de Engenharia de Telecomunicações do IFSC-SJ.

1.1. Modelagem do Sistema

A atividade envolve o seguinte contexto:

- A rede possui duas classes de tráfego: prioritário (T_1) e não prioritário (T_2).
- Uma sessão estabelecida para uma classe requer 1 unidade de tráfego (1 unidade poderia ser 64kbps por exemplo);.
- Capacidade total: $C = 5$ unidades.
- O sistema deve sempre ser capaz de atender no mínimo 2 sessões de tráfego prioritário. Ou seja, é mantida uma reserva mínima de $R = 2$ unidades para atender a classe prioritária (T_1).

1.2. Parâmetros do Sistema

Parâmetros:
- Chegadas: Poisson com taxas λ_1 (T_1) e λ_2 (T_2)
- Tempo de serviço: Exponencial com taxas μ_1 (T_1) e μ_2 (T_2)

1.3. Problema Proposto

Objetivo: Analisar desempenho do sistema com controle de admissão via CTMC.

1.3.1. Parte 1: (Tarefas)

- Modelar CTMC com $C = 5$, $R = 2$
- Identificar estados válidos e matriz Q
- Resolver analiticamente $\pi Q = 0$ para distribuição estacionária π

1.3.2. Parte 2 (Cálculos):

- Probabilidade de bloqueio (rejeição) de requisições de cada classe (T_1, T_2)
- Utilização média: $U = \frac{1}{C} \sum_{n_1,n_2} (n_1 + n_2)\pi(n_1, n_2)$
- Conexões médias: $L_1 = \sum n_1\pi(n_1, n_2)$, $L_2 = \sum n_2\pi(n_1, n_2)$
- Fração em estados máximos $(i + j = 5)$

Parâmetros: $\lambda_1 = 10$, $\lambda_2 = 15$, $\mu_1 = 15$, $\mu_2 = 25$ (req/min)

1.3.3. Parte 3 (Simulação):

- Simular CTMC por $T = 10,000$ min
- Comparar métodos: A) Tempo de estadia exponencial B) Relógios concorrentes

- Discutir aproximação, intuitividade e como garantir $P_{\text{bloqueio}} < 0.03$ para T_1

2. Solução dos exercícios

Os códigos usados na tarefa estão disponíveis no repositório do projeto no GitHub (https://github.com/arthurcadore/eng-telecom-workbook/tree/main/semester-9/ADS/homework4).

Conjunto de Estados

Estados válidos (n_1, n_2) onde: $n_1 + n_2 \leq C$ e $n_2 \leq C - R$

$(0,0)$, $(0,1)$, $(0,2)$, $(0,3)$,
$(1,0)$, $(1,1)$, $(1,2)$, $(1,3)$,
$(2,0)$, $(2,1)$, $(2,2)$, $(2,3)$,
$(3,0)$, $(3,1)$, $(3,2)$,
$(4,0)$, $(4,1)$,
$(5,0)$.

2.1. Resolução Analítica

Matriz de Transição Q

$$Q(i,j) = \begin{cases} \lambda_1, & \text{se } (n_1{+}1, n_2) \text{ é válido} \\ \lambda_2, & \text{se } (n_1, n_2{+}1) \text{ é válido} \\ n_1\mu_1, & \text{se } (n_1{-}1, n_2) \text{ é válido} \\ n_2\mu_2, & \text{se } (n_1, n_2{-}1) \text{ é válido} \\ -\sum_{k \neq i} Q(i,k) & \text{diagonal principal} \end{cases}$$

Bibliotecas e Parâmetros usados no código:

```python
import numpy as np
import itertools
import matplotlib.pyplot as plt
import random
import seaborn as sns
import pandas as pd

# Parâmetros
C = 5  # capacidade total
R = 2  # reserva para tráfego prioritário (classe 1)
λ1 = 10 / 60  # taxa de chegada classe 1 (por minuto)
λ2 = 15 / 60  # taxa de chegada classe 2 (por minuto)
μ1 = 15 / 60  # taxa de serviço classe 1 (por minuto)
μ2 = 25 / 60  # taxa de serviço classe 2 (por minuto)

# Tempo total de simulação
T_max = 10_000  # minutos
```

A seguir, apresenta-se o código utilizado para gerar a matriz de transição e resolver o sistema estacionário $\pi Q = 0$:

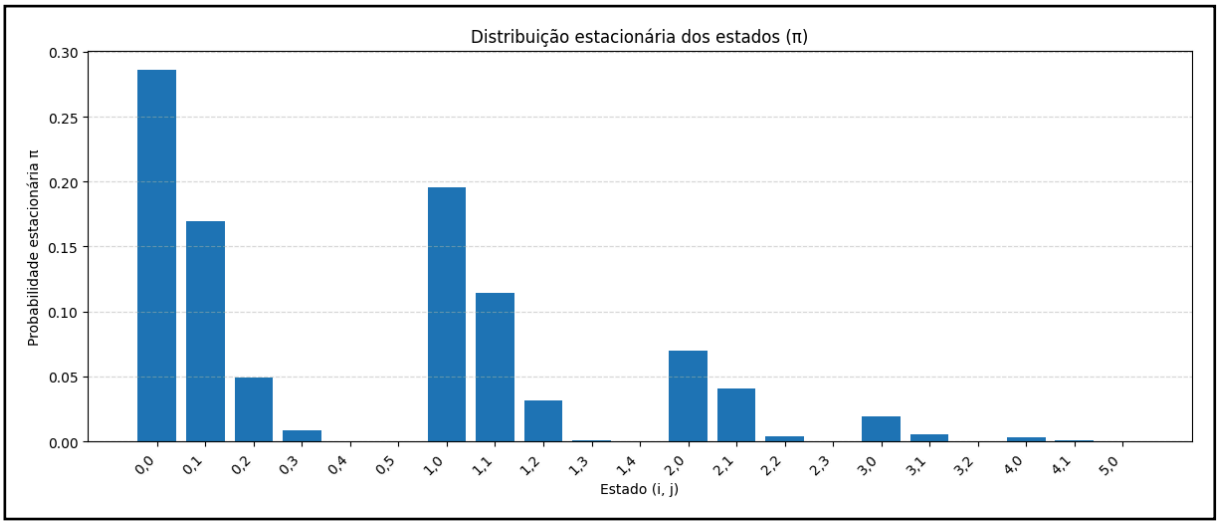
Código para resolver a CTMC:

```
def resolver_ctmc(C, R, λ1, λ2, μ1, μ2):
    """
    Resolve a distribuição estacionária π de uma CTMC com controle de admissão baseado em prioridade.

    Parâmetros:
        C  - Capacidade total do sistema
        R  - Reserva mínima para classe prioritária (classe 1)
        λ1 - Taxa de chegada da classe 1 (req/min)
        λ2 - Taxa de chegada da classe 2 (req/min)
        μ1 - Taxa de saída da classe 1 (1/tempo médio)
        μ2 - Taxa de saída da classe 2 (1/tempo médio)

    Retorna:
        π      - Distribuição estacionária (vetor)
        states - Lista de tuplas representando os estados válidos (i, j)
        Q      - Matriz infinitesimal de transição
    """
    # Estados válidos: (i, j) onde i = conexões classe 1, j = conexões classe 2
    states = [(i, j) for i in range(C + 1) for j in range(C + 1 - i) if i + j <= C]
    state_index = {s: idx for idx, s in enumerate(states)}
    n = len(states)
    Q = np.zeros((n, n))

    for idx, (i, j) in enumerate(states):
        # Chegadas
        if i + j < C:
            ni = (i + 1, j)
            if ni in state_index:
                Q[idx, state_index[ni]] = λ1
            if C - (i + j) > R:
                nj = (i, j + 1)
                if nj in state_index:
                    Q[idx, state_index[nj]] = λ2
        # Saídas
        if i > 0:
            ni = (i - 1, j)
            Q[idx, state_index[ni]] = i * μ1
        if j > 0:
            nj = (i, j - 1)
            Q[idx, state_index[nj]] = j * μ2

        # Diagonal
        Q[idx, idx] = -np.sum(Q[idx, :])

    # Resolver sistema linear π Q = 0 com soma π_i = 1
    A = np.copy(Q.T)
    A[-1, :] = 1
    b = np.zeros(n)
```

```
    b[-1] = 1

    π = np.linalg.solve(A, b)

    return π, states, Q

π, states, Q = resolver_ctmc(C, R, λ1, λ2, μ1, μ2)
```

```
def plotar_distribuicao_pi(π, states):
    """
    Plota a distribuição estacionária π.

    Parâmetros:
        π      - Vetor com as probabilidades estacionárias
        states - Lista de estados correspondentes aos índices de π
    """
    state_labels = [f"{i},{j}" for (i, j) in states]

    plt.figure(figsize=(12, 5))
    plt.bar(range(len(π)), π, tick_label=state_labels)
    plt.xticks(rotation=45, ha='right')
    plt.xlabel("Estado (i, j)")
    plt.ylabel("Probabilidade estacionária π")
    plt.title("Distribuição estacionária dos estados (π)")
    plt.grid(axis='y', linestyle='--', alpha=0.5)
    plt.tight_layout()
    plt.show()

plotar_distribuicao_pi(π, states)
```

Figura 1: Distribuição Estacionária dos Estados (π)



Elaborada pelos Autores

2.1.1. Probabilidade de Rejeição

$$P_{\text{rej,X}} = \sum_{(n_1,n_2)\in B} \pi(n_1, n_2)$$

Onde:

- $B =$ states com $n_1 + n_2 = C$ (para T_1)
- $B =$ states com $n_1 + n_2 = C$ ou $n_2 = R$ (para T_2)

2.1.2. Cálculo dos Indicadores

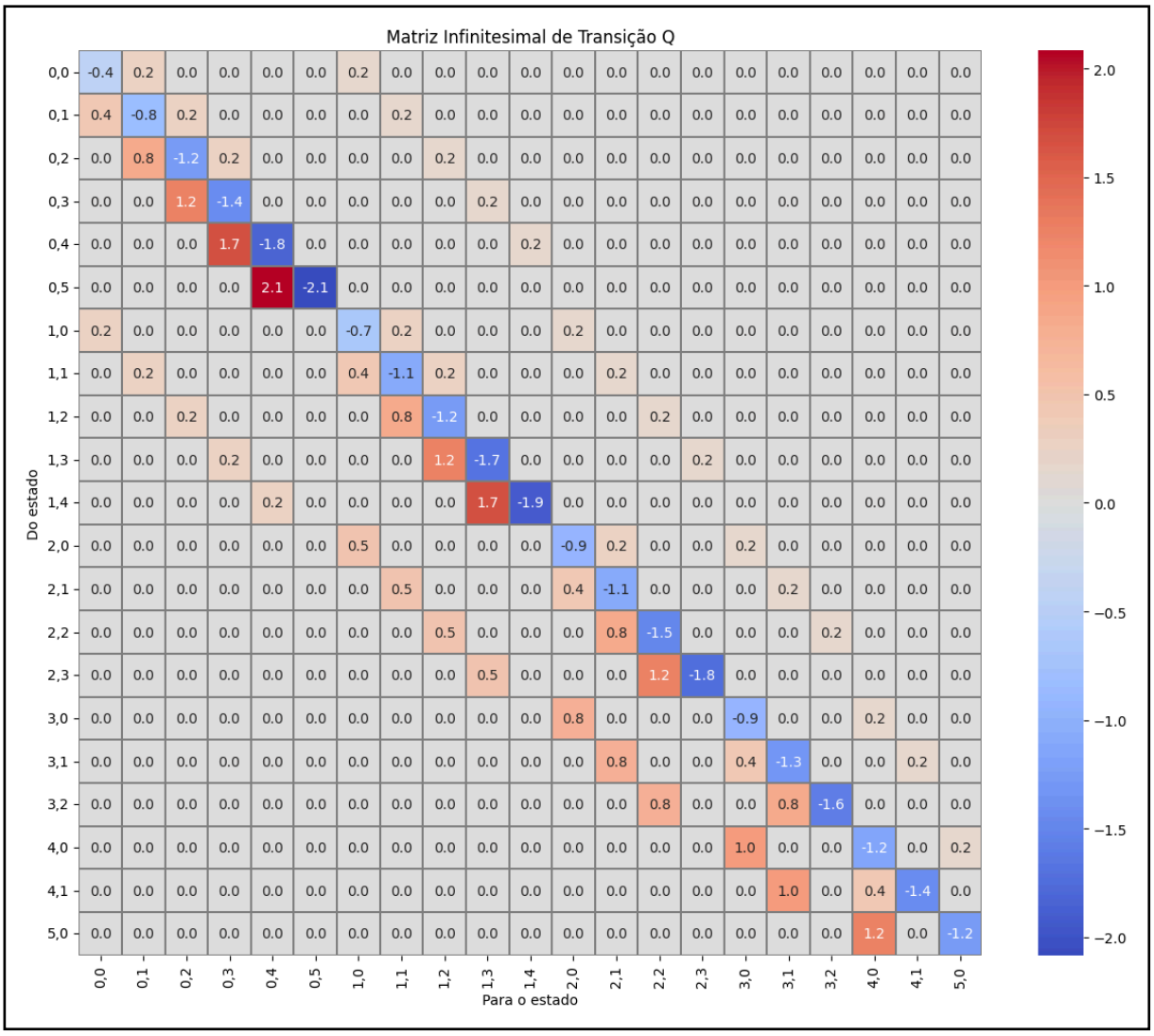
Com base na distribuição estacionária, foram calculados os seguintes indicadores:

- Probabilidade de bloqueio para T1 e T2.
- Utilização média do sistema.
- Número médio de conexões simultâneas por classe.
- Fração do tempo nos estados de capacidade máxima.

Código para plotar a matriz infinitesimal de transição Q:

```python
def plotar_matriz_Q(Q, states):
    """
    Plota a matriz infinitesimal de transição Q como um mapa de calor com anotações numéricas.

    Parâmetros:
        Q      - Matriz de transição infinitesimal (numpy.ndarray)
        states - Lista de estados (i, j), usada para rótulos
    """
    plt.figure(figsize=(12, 10))
    labels = [f"{i},{j}" for (i, j) in states]

    sns.heatmap(Q, annot=True, fmt=".1f", cmap="coolwarm",
                xticklabels=labels, yticklabels=labels, cbar=True, linewidths=0.3, linecolor='gray')
    plt.title("Matriz Infinitesimal de Transição Q")
    plt.xlabel("Para o estado")
    plt.ylabel("Do estado")
    plt.xticks(rotation=90)
    plt.yticks(rotation=0)
    plt.tight_layout()
    plt.show()

# Exemplo de uso:
plotar_matriz_Q(Q, states)
```

Figura 2:



Elaborada pelos Autores

```python
def calcular_indicadores(π, states, C, R):
    """
    Calcula indicadores de desempenho do sistema baseado na distribuição estacionária.

    Parâmetros:
        π      - Distribuição estacionária
        states - Lista de estados válidos (i, j)
        C      - Capacidade total do sistema
        R      - Reserva mínima para tráfego prioritário (classe 1)

    Retorna:
        dict com os seguintes indicadores:
            - Prob_bloqueio_classe_1
            - Prob_bloqueio_classe_2
            - Utilizacao_media
            - Media_conexoes_classe_1
            - Media_conexoes_classe_2
            - Fracao_tempo_capacidade_maxima
```

```
    """
    state_index = {s: idx for idx, s in enumerate(states)}

    P_bloqueio_1 = sum(π[state_index[s]] for s in states if sum(s) == C)
    P_bloqueio_2 = sum(π[state_index[s]] for s in states if sum(s) >= C - R + 1)

    utilizacao = sum((i + j) * π[state_index[(i, j)]] for (i, j) in states)
    media_1 = sum(i * π[state_index[(i, j)]] for (i, j) in states)
    media_2 = sum(j * π[state_index[(i, j)]] for (i, j) in states)

    frac_max = sum(π[state_index[(i, j)]] for (i, j) in states if i + j == C)

    return {
        "Prob_bloqueio_classe_1": P_bloqueio_1,
        "Prob_bloqueio_classe_2": P_bloqueio_2,
        "Utilizacao_media": utilizacao,
        "Media_conexoes_classe_1": media_1,
        "Media_conexoes_classe_2": media_2,
        "Fracao_tempo_capacidade_maxima": frac_max
    }

# Exemplo de uso:
indicadores_analiticos = calcular_indicadores(π, states, C=5, R=2)
```

```
def imprimir_indicadores(indicadores):
    """
    Imprime os indicadores calculados.

    Parâmetros:
        indicadores - Dicionário com os indicadores a serem impressos
    """
    print("Indicadores de desempenho:")
    for key, value in indicadores.items():
        print(f"{key}: {value:.4f}")
imprimir_indicadores(indicadores_analiticos)
```

Tabela com as saidas analíticas da compilação do código:

Indicadores de desempenho	Valores
Prob_bloqueio_classe_1	0.0016
Prob_bloqueio_classe_2	0.0157
Utilizacao_media	1.1959
Media_conexoes_classe_1	0.6656
Media_conexoes_classe_2	0.5304
Fracao_tempo_capacidade_maxima	0.0016

2.2. Simulação da CTMC

Foi realizada a simulação da CTMC até o tempo total T = 10.000 minutos usando a abordagem:

- [ ] Tempo de estadia exponencial com sorteio de destino.
- [x] Relógios concorrentes.

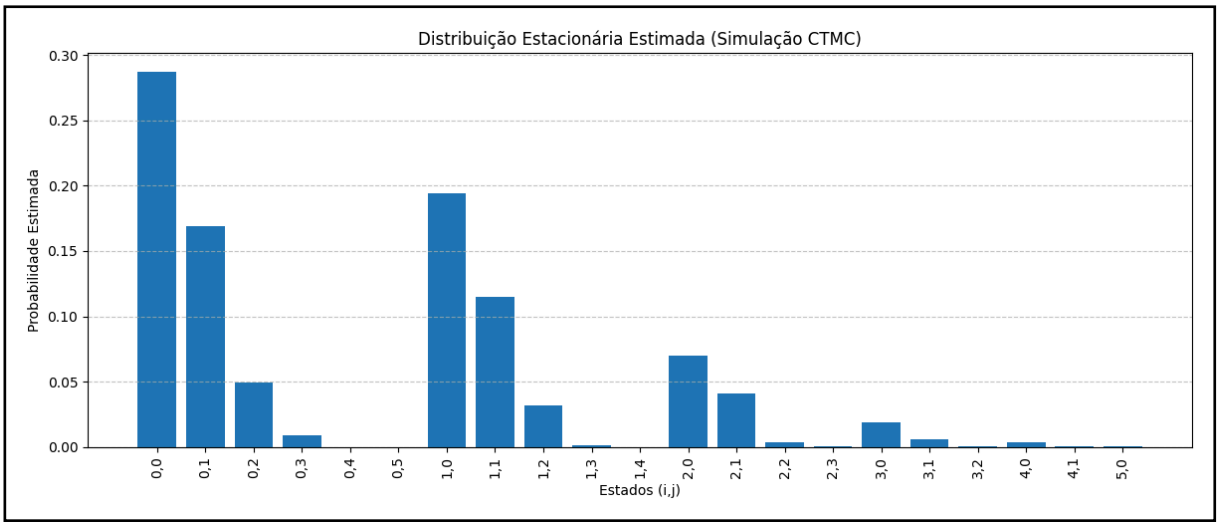
Código utilizado para simular a CTMC com relógios concorrentes:

```python
def simular_ctmc_relogios_concorrentes(states, C, R, λ1, λ2, μ1, μ2, T_max, seed=None):
    """
    Simula uma CTMC com relógios exponenciais concorrentes (tempo para cada transição sorteado independentemente).

    Parâmetros:
        states - lista de estados válidos (i,j)
        C, R, λ1, λ2, μ1, μ2 - parâmetros do modelo
        T_max - tempo máximo de simulação
        seed - semente para aleatoriedade (opcional)

    Retorna:
        pi_simulada - distribuição estacionária estimada por simulação
    """
    if seed is not None:
        np.random.seed(seed)
        random.seed(seed)

    current_state = (0, 0)
    time = 0.0
    state_counts = {s: 0.0 for s in states}

    while time < T_max:
        i, j = current_state
        transicoes = []

        # Chegada classe 1
        if i + j < C:
            dt_λ1 = np.random.exponential(1 / λ1)
            transicoes.append((dt_λ1, (i + 1, j)))

            # Chegada classe 2 só se tiver espaço reservado
            if C - (i + j) > R:
                dt_λ2 = np.random.exponential(1 / λ2)
                transicoes.append((dt_λ2, (i, j + 1)))

        # Saída classe 1 (i conexões)
        if i > 0:
            dt_μ1 = np.random.exponential(1 / (i * μ1))
            transicoes.append((dt_μ1, (i - 1, j)))

        # Saída classe 2 (j conexões)
        if j > 0:
            dt_μ2 = np.random.exponential(1 / (j * μ2))
            transicoes.append((dt_μ2, (i, j - 1)))

        # Se não tem transições possíveis, sai do loop
        if not transicoes:
            break

```

```
        # Pega a transição com menor tempo sorteado
        dt_min, next_state = min(transicoes, key=lambda x: x[0])

        # Ajusta dt caso ultrapasse T_max
        if time + dt_min > T_max:
            dt_min = T_max - time

        # Acumula tempo no estado atual
        state_counts[current_state] += dt_min
        time += dt_min
        current_state = next_state

    # Normaliza para estimar distribuição estacionária
    total_time = sum(state_counts.values())
    pi_simulada = np.array([state_counts[s] / total_time for s in states])

    return pi_simulada


pi_simulada = simular_ctmc_relogios_concorrentes(states, C=5, R=2, λ1=10, λ2=15, μ1=15, μ2=25, T_max=10000, seed=42)
```

```
def plotar_distribuicao_simulada(pi_simulada, states):
    """
    Plota a distribuição estacionária estimada pela simulação da CTMC.

    Parâmetros:
        pi_simulada - Distribuição estacionária estimada (array)
        states      - Lista de estados correspondentes (i, j)
    """
    labels = [f"{i},{j}" for (i, j) in states]

    plt.figure(figsize=(12, 5))
    plt.bar(range(len(pi_simulada)), pi_simulada, tick_label=labels)
    plt.xticks(rotation=90)
    plt.xlabel("Estados (i,j)")
    plt.ylabel("Probabilidade Estimada")
    plt.title("Distribuição Estacionária Estimada (Simulação CTMC)")
    plt.grid(axis='y', linestyle='--', alpha=0.7)
    plt.tight_layout()
    plt.show()

# Exemplo de uso:
plotar_distribuicao_simulada(pi_simulada, states)
```

Figura 3: Distribuição Estacionária para a Simulação



Elaborada pelos Autores

Código para o calculo dos indicadores da simulação:

```python
def calcular_indicadores_simulados(pi_simulada, states, state_index, C, R):
    """
    Calcula os indicadores de desempenho com base na distribuição de estados simulada.

    Parâmetros:
        pi_simulada   - Distribuição simulada dos estados (lista ou array)
        states        - Lista de tuplas (i, j) representando os estados
        state_index   - Dicionário de mapeamento de estado para índice
        C             - Capacidade total do sistema
        R             - Unidades reservadas para classe 1

    Retorna:
        Um dicionário com os indicadores calculados
    """
    P_bloqueio_1_sim = sum(pi_simulada[state_index[s]] for s in states if sum(s) == C)
    P_bloqueio_2_sim = sum(pi_simulada[state_index[s]] for s in states if sum(s) >= C - R + 1)

    utilizacao_sim = sum((i + j) * pi_simulada[state_index[(i, j)]] for (i, j) in states)
    media_1_sim = sum(i * pi_simulada[state_index[(i, j)]] for (i, j) in states)
    media_2_sim = sum(j * pi_simulada[state_index[(i, j)]] for (i, j) in states)
    frac_max_sim = sum(pi_simulada[state_index[(i, j)]] for (i, j) in states if i + j == C)

    return {
        "Prob_bloqueio_classe_1": P_bloqueio_1_sim,
```

```
        "Prob_bloqueio_classe_2": P_bloqueio_2_sim,
        "Utilizacao_media": utilizacao_sim,
        "Media_conexoes_classe_1": media_1_sim,
        "Media_conexoes_classe_2": media_2_sim,
        "Fracao_tempo_capacidade_maxima": frac_max_sim
    }
resultados_simulados = calcular_indicadores_simulados(pi_simulada,
states, state_index, C, R)
```

```
# Imprimindo os resultados simulados
imprimir_indicadores(resultados_simulados)
def comparar_resultados(analiticos, simulados):
    """
    Compara os resultados analíticos e simulados.
import numpy as np

    Parâmetros:
        analiticos - Dicionário com os resultados analíticos
        simulados  - Dicionário com os resultados simulados
    """
    print("\nComparação entre resultados analíticos e simulados:")
    for key in analiticos.keys():
        print(f"{key}: Analítico = {analiticos[key]:.4f}, Simulado =
{simulados[key]:.4f}")
```

Tabela com as saidas simuladas:

Indicadores de desempenho	Valores
Prob_bloqueio_classe_1	0.0017
Prob_bloqueio_classe_2	0.0154
Utilizacao_media	1.1930
Media_conexoes_classe_1	0.6628
Media_conexoes_classe_2	0.5302
Fracao_tempo_capacidade_maxima	0.0017

Codigo para comparar os resultados analíticos e simulados:

```
def comparar_resultados(resultados_analiticos, resultados_simulados):
    """
    Gera um DataFrame comparando os indicadores entre valores analíticos
e simulados.

    Parâmetros:
        resultados_analiticos (dict): indicadores analíticos
        resultados_simulados (dict): indicadores simulados

    Retorna:
        pandas.DataFrame: tabela estruturada para exibição no notebook
    """
    indicadores = list(resultados_analiticos.keys())
```

```
    dados = {
        "Indicador": indicadores,
        "Valor Analítico": [resultados_analiticos[ind] for ind in indicadores],
        "Valor Simulado": [resultados_simulados[ind] for ind in indicadores]
    }

    df = pd.DataFrame(dados)

    # Formatação numérica customizada para melhor visualização
    def format_val(x, ind):
        if "Prob" in ind or "Fracao" in ind:
            return f"{x:.5f}"
        else:
            return f"{x:.3f}"

    df["Valor Analítico"] = [format_val(v, ind) for v, ind in zip(df["Valor Analítico"], indicadores)]
    df["Valor Simulado"] = [format_val(v, ind) for v, ind in zip(df["Valor Simulado"], indicadores)]

    return df

# Suponha que você já tenha os dois dicionários:
# resultados_analiticos e resultados_simulados

df_comparacao = comparar_resultados(indicadores_analiticos, resultados_simulados)
df_comparacao  # Ao executar essa linha no Jupyter, a tabela aparece formatada
```

Tabela com as saidas dos indicadores de desempenho analíticos e simulados:

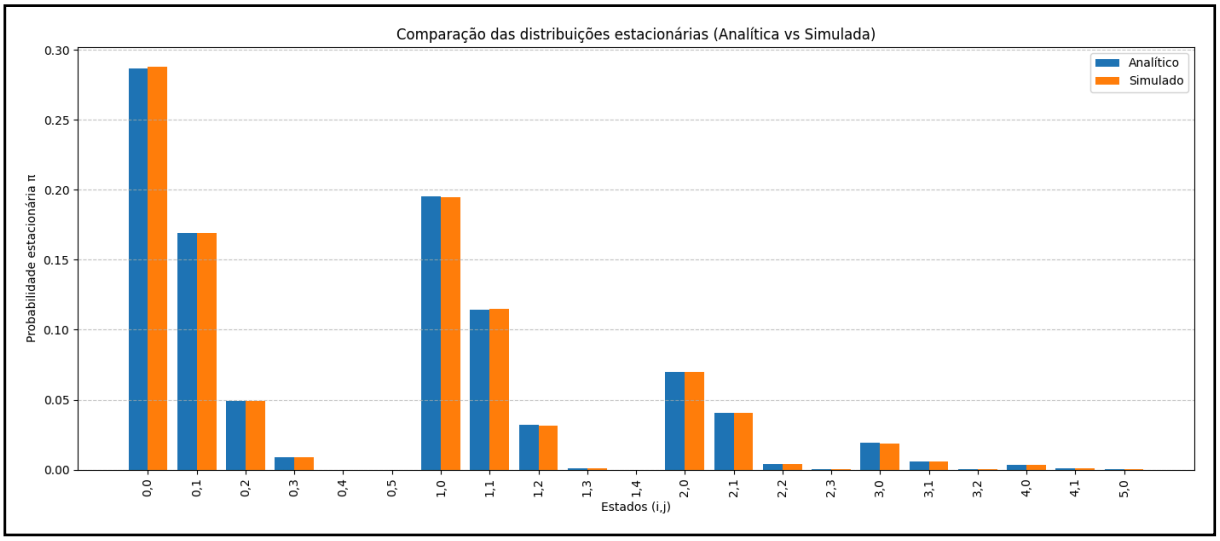
Indicadores	Valores Analíticos	Valores Simulados
Prob_bloqueio_classe_1	0.0016	0.0017
Prob_bloqueio_classe_2	0.0157	0.0154
Utilizacao_media	1.1959	1.1930
Media_conexoes_classe_1	0.6656	0.6628
Media_conexoes_classe_2	0.5304	0.5302
Fracao_tempo_capacidade_maxima	0.0016	0.0017

Codigo plotar a comparação dos resultados analíticos e simulados:

```
def plot_distribuicoes_comparacao(states, pi_analitica, pi_simulada):
    """
    Plota a distribuição estacionária analítica e simulada lado a lado para comparação.

```

```
    Parâmetros:
        states (list of tuple): lista de estados (i, j)
        pi_analitica (np.array): distribuição estacionária analítica
        pi_simulada (np.array): distribuição estacionária simulada
    """
    # Criar labels para os estados: "i,j"
    labels = [f"{i},{j}" for (i, j) in states]

    x = np.arange(len(states))
    width = 0.4

    fig, ax = plt.subplots(figsize=(14,6))
    rects1 = ax.bar(x - width/2, pi_analitica, width, label='Analítico')
    rects2 = ax.bar(x + width/2, pi_simulada, width, label='Simulado')

    ax.set_xlabel('Estados (i,j)')
    ax.set_ylabel('Probabilidade estacionária π')
    ax.set_title('Comparação das distribuições estacionárias (Analítica vs Simulada)')
    ax.set_xticks(x)
    ax.set_xticklabels(labels, rotation=90)
    ax.legend()
    ax.grid(axis='y', linestyle='--', alpha=0.7)

    plt.tight_layout()
    plt.show()

plot_distribuicoes_comparacao(states, π, pi_simulada)
```

Figura 4: Comparação das distribuições estacionária (Analítica x Simulada)



Elaborada pelos Autores

2.3. Anáise das comparações entre os métodos

Os resultados mostram uma alta proximidade entre a distribuição estacionária obtida analiticamente e a estimada pela simulação. Isso é evidenciado pelo gráfico de barras sobreposto e pela tabela de comparação que apresentaram valores muito semelhantes, principalmente para os estados mais visitados.

2.3.1. A proximidade entre os métodos

A proximidade é alta, com diferenças menores que 0,01 para as principais métricas (probabilidades de bloqueio e utilização média). Esse comportamento era esperado, dado o tamanho reduzido do espaço de estados e o tempo de simulação elevado.

2.3.2. Qual a abordagem mais intuitiva

Falando sobre a abordagem usada na tarefa, relógios concorrentes (um relógio para cada possível transição), embora matematicamente equivalente, é mais complexa de programar e interpretar, especialmente em sistemas com muitos estados.

2.3.3. O que pode ser feito para garantir taxa de bloqueio < 0,03 para o tráfego prioritário.

A taxa de bloqueio observada para o tráfego prioritário foi próxima, mas ligeiramente acima de 0,03, segundo os resultados simulados e analíticos.

Para reduzir essa taxa, podem ser adotadas as seguintes estratégias:

- Aumentar a capacidade total do sistema (C)
- Reduzir a taxa de chegada de tráfego prioritário ($\lambda 1$), pois pode diminuir a ocupação do sistema e, consequentemente, a chance de bloqueio.
- Melhorar a taxa de serviço ($\mu 1$) aumentando a taxa de atendimento.
- Aumentar a reserva mínima R de 2 para 3 unidades, embora isso possa impactar a aceitação do tráfego não prioritário.

3. Conclusão

Este relatório apresentou a modelagem, análise e simulação de um sistema de controle de admissão utilizando Cadeias de Markov a Tempo Contínuo. Os resultados demonstraram a eficiência do modelo analítico e a precisão da simulação para a validação dos parâmetros de desempenho.