IPRJ - Métodos Numéricos para Equações Diferenciais

Trabalho 1

Professor: Grazione Souza

Nome do aluno: Pedro Henrique Couto Silva
Matrícula: 202020466311

Nome do aluno : Vinicius Carvalho Monnerat Bandeira
Matrícula: 202020466711

RESUMO

Este trabalho apresenta a implementação e avaliação de um sistema de equações diferenciais ordinárias que descrevem a dinâmica de concentração de um reagente e temperatura em um reator químico. Utilizou-se o método Runge-Kutta de 4ª ordem (RK4) para a solução numérica, validando seus resultados por meio do método *solve_ivp* da biblioteca SciPy. A implementação foi realizada em Python, utilizando pacotes como NumPy, SciPy e Matplotlib para cálculos e visualização dos resultados. Além da validação, foram conduzidas simulações com diferentes passos de tempo para avaliar o impacto na precisão e no custo computacional para definir um equilíbrio ideal. O trabalho também explorou variações nas condições iniciais de temperatura e concentração, analisando como essas alterações afetam o sistema.

SUMÁRIO

INTRODUÇÃO

Este trabalho tem como objetivo implementar e avaliar a solução de um sistema de equações diferenciais que descrevem a concentração e a temperatura em um reator químico, utilizando o método de Runge-Kutta clássico de 4ª ordem (RK4). A modelagem segue as equações propostas pelo problema em estudo, abordando o comportamento dinâmico do sistema reacional. Além da implementação do método RK4, os resultados são validados por meio da função *solve_ivp* da biblioteca SciPy, que oferece uma solução numérica eficiente para equações diferenciais ordinárias. A análise inclui não apenas a comparação entre os dois métodos, mas também uma investigação detalhada para diferentes condições iniciais e resoluções temporais, permitindo uma avaliação do impacto dessas variáveis na precisão e na estabilidade das soluções.

1 ENTENDENDO O PROBLEMA

1.1 Equações do reator

Antes de modelar a solução, é necessário entender o que foi proposto para o trabalho. Generalizando, o trabalho propõe o estudo de um reator visando a concentração de um reagente e a temperatura.

As seguintes equações foram propostas:

$$\dot{C} = -e^{-\frac{10}{T+273}} \cdot C, \tag{1}$$

$$\dot{T} = 1000 \cdot e^{-\frac{10}{T+273}} \cdot C - 10 \cdot (T - 20), \tag{2}$$

Assim, pode-se deduzir a necessidade de resolver um sistema visto que (2) depende de (1).

1.2 **Runge-Kutta clássico de 4ª ordem**

Para que o método seja generalizado, ou seja, possa ser repetido para outros problemas, é necessário criar uma função geral do método. Como visto em sala, tem-se a formulação geral para f^{n+1}, k_1, k_2, k_3, k_4.

2 MODELANDO O PROBLEMA

Tendo entendido as necessidades do trabalho, pode-se iniciar a modelagem da solução. A equipe optou por desenvolver a solução utilizando a Python, uma linguagem de código aberto, fácil utilização e grande comunidade. Junto a ele, utilizou-se a ferramenta jupyter notebook para uma visualização e desenvolvimento mais direto. Os pacotes NumPy, SciPy e Matplotlib foram usados como auxiliares.

2.1 **Escrevendo o sistema de equações**

Como visto em 1.1, o reator possui duas equações que descrevem seu funcionamento, de tal forma que a temperatura do reator depende da concentração do reagente, de maneira geral, o sistema ficou como o seguinte:

Figura 1 - Sistema de equações

```
1  def sistema(x, t):
2          f1, f2 = x
3          dx1dt = -np.exp((-10)/(f2+273))*f1
4          dx2dt = (1000*np.exp((-10)/(f2+273))*f1)-(10*(f2-20))
5          return np.array([dx1dt, dx2dt])
```

Fonte: O autor

Onde x é uma lista com os valores de concentração (f_1) e temperatura (f_2), $dx1dt$ e $dx2dt$ são as equações $\dot{C}$ e $\dot{T}$ respectivamente, o retorno da função é uma lista com os valores de $\dot{C}$ e $\dot{T}$.

2.2 **Escrevendo o método RK4**

Como visto em 1.2, o objetivo é criar uma função generalizada para solucionar problemas diversos, logo, é necessário informar a equação ou sistema de equações que se espera solucionar, a condição inicial e o passo de tempo da solução.

Figura 2 - Função de Runge-Kutta

```
1   def rk4(f, x0, t):
2       x = np.zeros((len(t), len(x0)))
3       x[0] = x0
4       for i in range(1, len(t)):
5           dt = t[i] - t[i-1]
6           k1 = f(x[i-1], t[i-1])
7           k2 = f(x[i-1] + 0.5*dt*k1, t[i-1] + 0.5*dt)
8           k3 = f(x[i-1] + 0.5*dt*k2, t[i-1] + 0.5*dt)
9           k4 = f(x[i-1] + dt*k3, t[i])
10          x[i] = x[i-1] + (dt/6)*(k1 + 2*k2 + 2*k3 + k4)
11      return x
```

Fonte: O autor

Onde f é a função que se deseja obter a solução, nesse caso, f é a função definida pelo sistema de equações em 2.1, $x0$ é a condição inicial para o problema neste caso a concentração e temperatura inicial, e por fim, t é o vetor de tempo discretizado sobre o passo de tempo Δt.

2.3 **Obtendo os primeiros resultados**

Dadas as funções de 2.1 e 2.3, basta definir as condições iniciais, e o vetor de tempo de simulação.

Figura 3 - Primeiras condições iniciais

```
1   Tini = 35 # temperatura inicial em graus Celsius
2   Cini = 5 # concentração inicial em gmol/L
3
4   x0 = [Cini, Tini] # vetor de condições iniciais
5   ts = 10 # tempo de simulação em segundos
6   t = np.linspace(0, ts, 1001) # vetor de tempo de 0 a 10 segundos
    com 1001 pontos, ou seja, com um passo de 0.01 segundos
```

Fonte: O autor

Onde f é a função que se deseja obter a solução, nesse caso, f é a função definida pelo sistema de equações em 2.1, $x0$ é a condição inicial para o problema neste caso a concentração e temperatura inicial, e por fim, t é o vetor de tempo discretizado sobre o passo de tempo Δt.

A solução será um vetor com as soluções para cada valor do vetor de tempo e para obtê-la basta executar a função 2.2 com os parâmetros definidos.

Figura 4 - Obtendo a primeira solução

```
1    solucao = rk4(sistema, x0, t)
```

Fonte: O autor

2.3.1 Entendendo o vetor de tempo de simulação

A simulação e resoluções em geral utilizam um tempo de objetivo e o passo de tempo para chegar nesse objetivo. Intuitivamente, cada valor da solução será dado para um tempo $t + \Delta t$. Na função vista em 2.2, o vetor de tempo é utilizado em dois momentos, o primeiro deles para determinar quantas iterações da função irão ocorrer, dependente do tamanho do vetor e o segundo momento é para calcular o Δt a ser utilizado, ou seja, seria possível passar para a função um vetor onde o passo é variável (porém não será trabalhado neste projeto). Pensando nessas necessidades, o vetor de tempo deve obrigatoriamente ter um valor inicial, um valor objetivo final e esses valores devem seguir alguma lógica de espaçamento. Utilizando o método *linspace* do pacote NumPy, é possível obter o vetor baseado nesses 3 pontos. Segundo a documentação oficial do NumPy, o método *linspace* é um método que "*Retorna um número de amostras uniformemente espaçadas, calculadas no intervalo [início , fim]*", logo basta informar o intervalo desejado e a quantidade de pontos dentro do intervalo. A quantidade de pontos está diretamente ligada ao passo de tempo, usando o exemplo de um intervalo de 0 até 10 segundos, para se obter um passo de 0.1 segundos, pode-se seguir a seguinte equação:

$$Passo = \frac{(Final - Inicial)}{Número\ de\ pontos}, \tag{3}$$

$$0.1 = \frac{(10 - 0)}{Número\ de\ pontos}, \tag{4}$$

$$Número\ de\ pontos\ =\ 100 \quad (5)$$

Obtido o número de pontos, soma-se 1 ao total para que o valor inicial e final estejam adequadamente incluídos e igualmente espaçados com o passo de tempo objetivo. O output para o vetor de tempo definido em (3)-(4)-(5) seria:

Figura 5 - Vetor [0,10] com passo 0.1

```
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9,  1. ,
        1.1,  1.2,  1.3,  1.4,  1.5,  1.6,  1.7,  1.8,  1.9,  2. ,  2.1,
        2.2,  2.3,  2.4,  2.5,  2.6,  2.7,  2.8,  2.9,  3. ,  3.1,  3.2,
        3.3,  3.4,  3.5,  3.6,  3.7,  3.8,  3.9,  4. ,  4.1,  4.2,  4.3,
        4.4,  4.5,  4.6,  4.7,  4.8,  4.9,  5. ,  5.1,  5.2,  5.3,  5.4,
        5.5,  5.6,  5.7,  5.8,  5.9,  6. ,  6.1,  6.2,  6.3,  6.4,  6.5,
        6.6,  6.7,  6.8,  6.9,  7. ,  7.1,  7.2,  7.3,  7.4,  7.5,  7.6,
        7.7,  7.8,  7.9,  8. ,  8.1,  8.2,  8.3,  8.4,  8.5,  8.6,  8.7,
        8.8,  8.9,  9. ,  9.1,  9.2,  9.3,  9.4,  9.5,  9.6,  9.7,  9.8,
        9.9, 10. ])
```

Fonte: O autor

O passo de tempo é um parâmetro crítico para as soluções. Um passo mal ajustado pode introduzir erros significativos no método, causando instabilidade e perdendo detalhes importantes. Por outro lado, um passo muito pequeno pode resultar em uma solução precisa, mas com alto custo computacional, logo, o passo ideal é aquele que melhor se aproxima de um resultado consistente sob menor custo computacional. Nas seções 4.1 e 4.2 serão aprofundados como o passo de tempo influencia na solução para o trabalho proposto.

2.3.2 Visualizando os resultados

Para a visualização dos resultados, utiliza-se o pacote Matplotlib, que permite criar gráficos de maneira simples e efetiva. A solução do sistema será um vetor de pares ordenados de concentração e temperatura para cada valor do vetor de tempo. Assim para plotar esses valores basta a seguinte lógica:

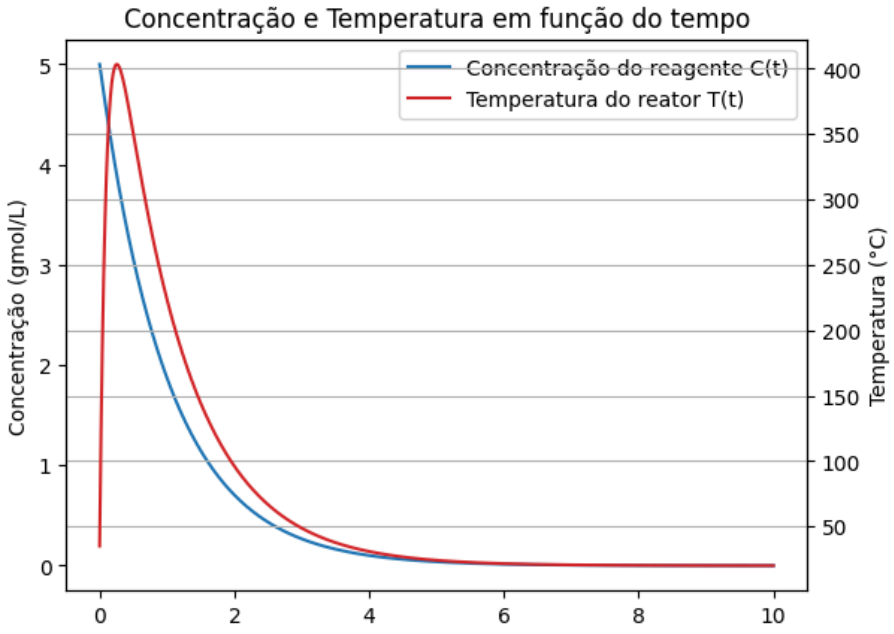
Figura 6 - Lógica para plotar os resultados

```
fig, ax = plt.subplots()
ax2 = ax.twinx() # cria um eixo secundário


plotC = ax.plot(t, solucao[:, 0],label='Concentração do reagente C(t)', color='tab:blue') # plota o gráfico da concentração
plotT = ax2.plot(t, solucao[:, 1], label='Temperatura do reator T(t)', color='tab:red') # plota o gráfico da temperatura

ax.set_ylabel('Concentração (gmol/L)')
ax2.set_ylabel('Temperatura (°C)')
plt.xlabel('Tempo (s)')
titulo = 'Concentração e Temperatura em função do tempo'
plt.title(titulo)
plt.grid()
ax.legend(handles=[plotC[0], plotT[0]])
plt.show()
```

Fonte: O autor

E o resultado será:

Figura 7 - Primeiros resultados



Fonte: O autor

3 VALIDAÇÃO DO MÉTODO

Para garantir que o método desenvolvido é coerente e correto, pode-se comparar com soluções de outros métodos. No Python, o pacote ScyPy fornece diferentes métodos para solução de equações diferenciais, o método *solve_ivp* é um deles, segundo a documentação oficial "*Esta função integra numericamente um sistema de equações diferenciais ordinárias dado um valor inicial:*

$dy / dt = f(t, y)$

$y(t0) = y0$

Aqui t é uma variável independente 1-D (tempo), y(t) é uma função ND com valor vetorial (estado), e uma função ND com valor vetorial f(t, y) determina as equações diferenciais. O objetivo é encontrar y(t) satisfazendo aproximadamente as equações diferenciais, dado um valor inicial y(t0)=y0".

O método utilizado como padrão para solução é "*RK45' (padrão): Método Runge-Kutta explícito de ordem 5*".

Dada a definição geral, o uso do método *solve_ivp* é similar ao definido em 2.2, tal que ambos dependem de valores vetoriais para o tempo e equações. Seu uso se dá da seguinte forma:

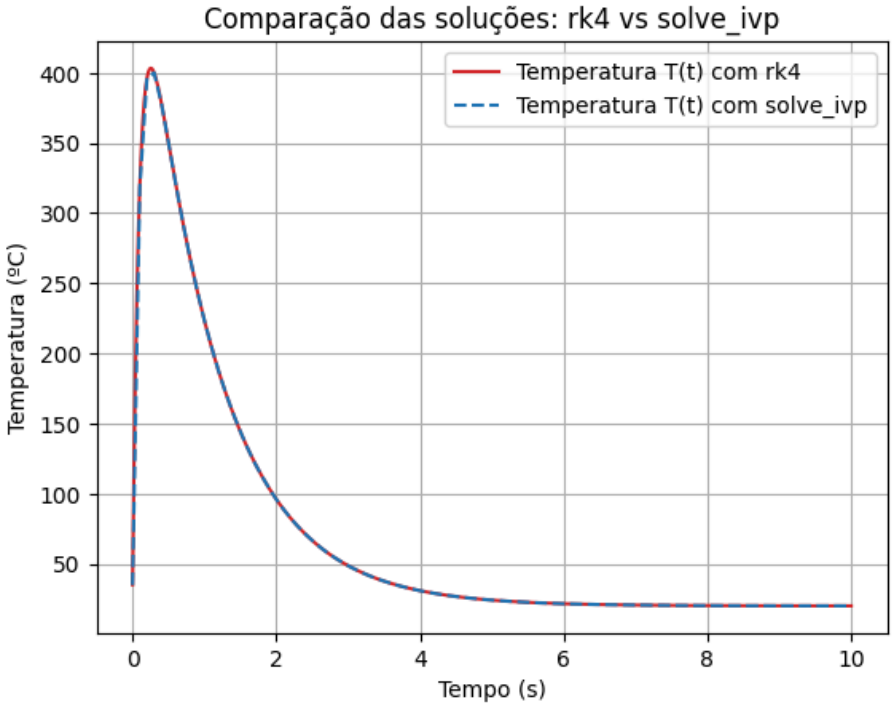
Figura 8 - Uso do solve_ivp

```
from scipy.integrate import solve_ivp

def sistema_ivp(t, x):
    f1, f2 = x
    dx1dt = -np.exp((-10)/(f2+273))*f1
    dx2dt = (1000*np.exp((-10)/(f2+273))*f1)-(10*(f2-20))
    return [dx1dt, dx2dt]

sol_ivp = solve_ivp(sistema_ivp, [0, 10], x0, t_eval=np.linspace(0, 10, 101))
```

Fonte: O autor

Sobrepondo as soluções do método construído e do método *solve_ivp:*

Figura 9 - Validação do método



Fonte: O autor

É visto então que os métodos sobrepostos se aproximam e por isso o método construído em 2.2 pode ser considerado válido para a solução de equações diferenciais.

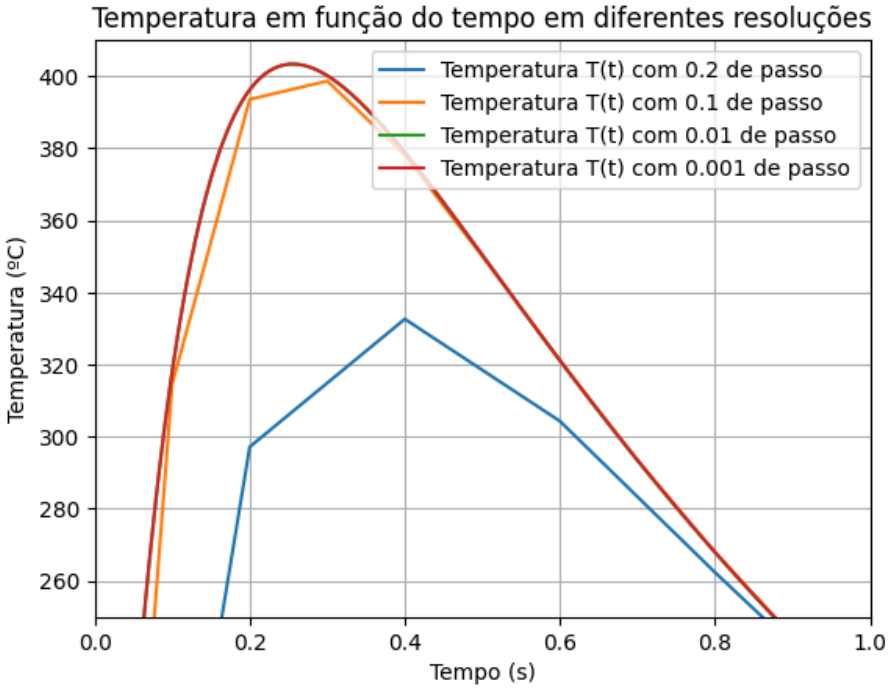
4 ANÁLISES DAS SIMULAÇÕES

Nesta seção, será estudado os efeitos das mudanças do passo de tempo na solução e do comportamento para diferentes condições iniciais.

4.1 Diferentes passos de tempo para a temperatura

Como visto em 2.3.1, a escolha do passo causa influência direta na precisão da solução e no custo computacional da execução, logo, é necessário escolher qual passo melhor se adequa ao problema proposto para o reator. Isso também significa que o passo de tempo ideal para a temperatura pode não ser o mesmo para a concentração, sendo necessário escolher separadamente qual o melhor passo para as equações. Pensando primeiro na temperatura, foi simulado condições de passo de 0.2, 0.1, 0.01 e 0.001 segundos e o resultado gráfico foi o seguinte utilizando uma temperatura inicial de $35°C$ e concentração de $5 gmol/L$:

Figura 10 - Diferentes passos para a temperatura



Fonte: O autor

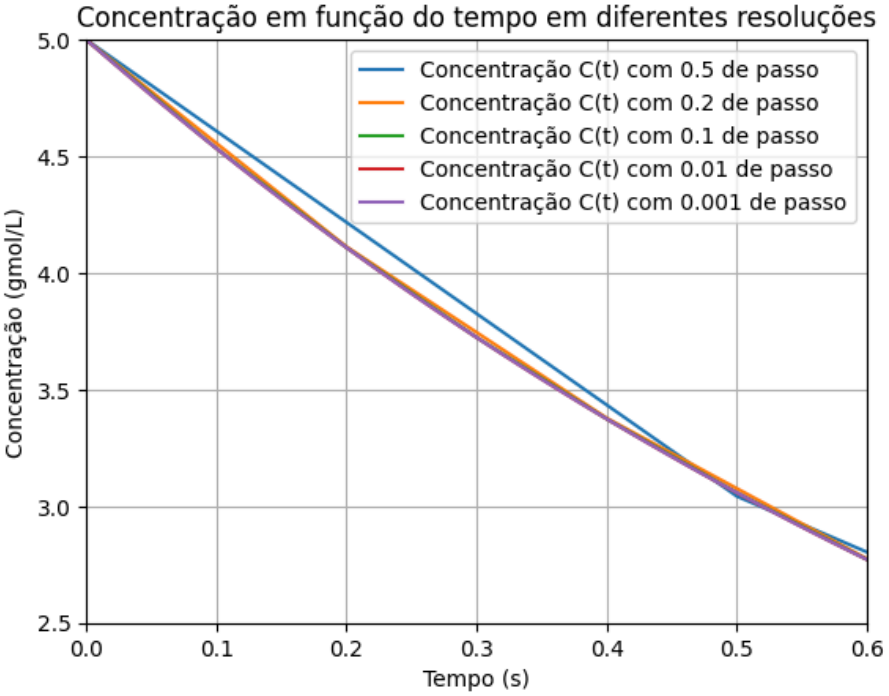
No plano destacado, nota-se que diminuindo o passo, a solução se torna mais precisa, estável e detalhada. Porém, nota-se que a curva de passo 0.01 foi perfeitamente sobreposta pela curva de passo 0.001, logo a curva de passo 0.01 obtém a mesma solução que uma curva

de passo maior porém com menos custo computacional, sendo assim, esse é o passo ideal para a temperatura.

4.2 **Diferentes passos de tempo para a concentração**

De maneira similar ao visto em 4.1, o passo ideal para a concentração pode ser definido da mesma forma, resultado gráfico foi o seguinte:

Figura 11 - Diferentes passos para a concentração



Fonte: O autor

No plano destacado, nota-se que a solução de passo 0.1 já satisfaz corretamente a precisão, detalhe e estabilidade das soluções de 0.01 e 0.001 de passo. Entretanto, como as soluções são obtidas em conjunto, tanto concentração quanto temperatura, o passo ideal é aquele que melhor serve à ambas curvas, neste caso 0.01 de passo.

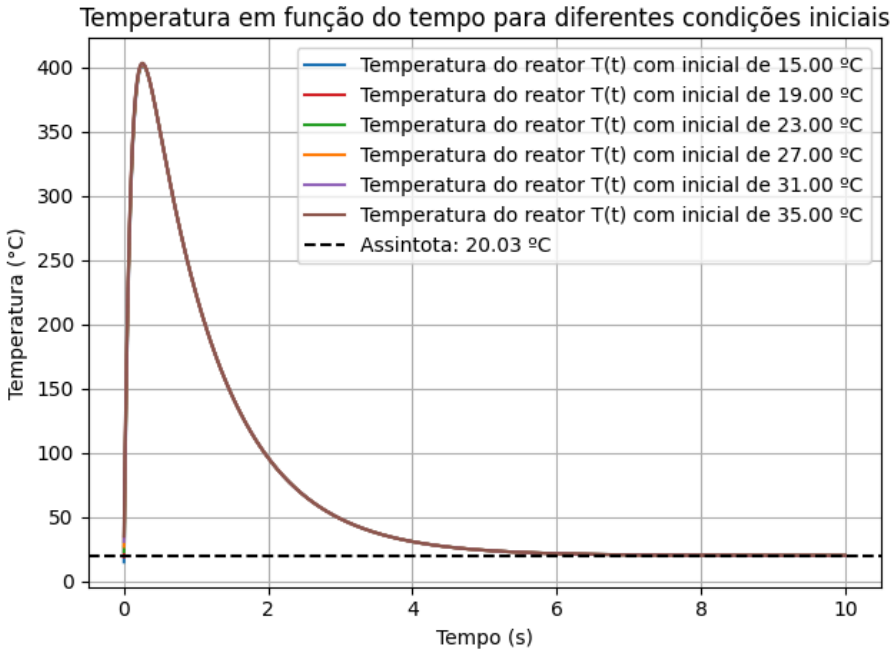
Dessa forma, foi definido então o melhor passo para as simulações pensando no custo computacional e no detalhe da solução.

4.3 **Diferentes condições iniciais de temperatura**

Nesta seção o objetivo é avaliar como diferentes condições de temperatura inicial influenciam no comportamento geral do reator. Para as simulações, foi mantido constante o

valor para a concentração de $5gmol/L$ e a temperatura inicial foi variada em 6 diferentes temperaturas entre $15°C$ e $35°C$.

Figura 12 - Diferentes temperatura iniciais



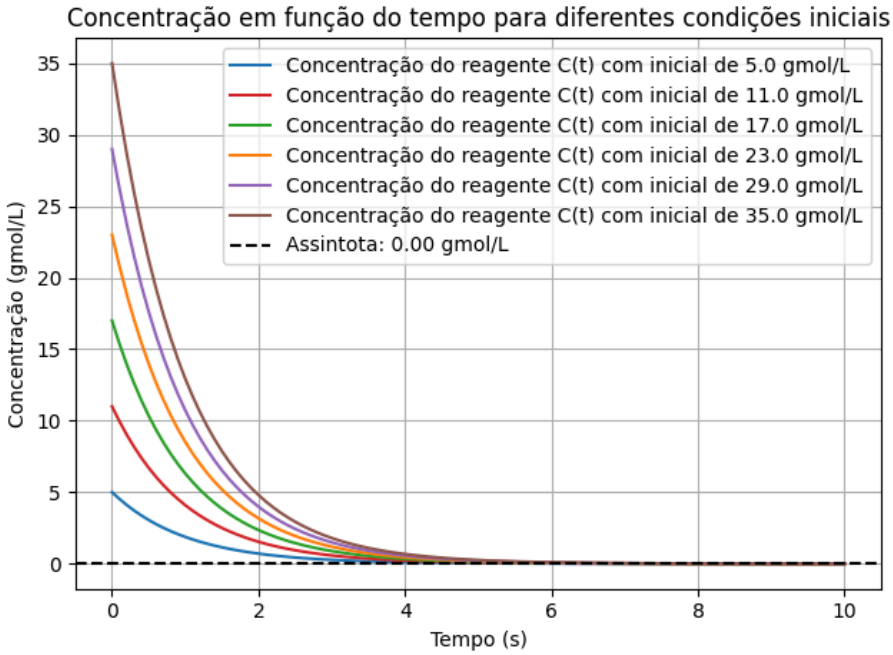
Fonte: O autor

Observa-se que todas as curvas tiveram o mesmo comportamento, variando apenas seu ponto inicial. O valor médio para o máximo de cada curva foi $402.57°C$ com desvio padrão de $0.52°C$, atingidos em 0.26 segundos de todas as curvas. Também observa-se uma redução de temperatura limitada à $20.03°C$ de todas as curvas também.

4.4 **Diferentes condições iniciais de concentração**

Nesta seção o objetivo é avaliar como diferentes condições de concentração inicial influenciam no comportamento geral do reagente. Para as simulações, foi mantido constante o valor para a temperatura de $15°C$ e a concentração inicial foi variada em 6 diferentes concentrações entre $5gmol/L$ e $35gmol/L$.

Figura 13 - Diferentes concentrações iniciais



Fonte: O autor

Novamente todas as curvas tiveram o mesmo comportamento, variando apenas seu ponto inicial. O valor limite de $0 gmol/L$ foi observado e era esperado levando em consideração a equação (1).

CONCLUSÃO

Com a implementação do método de Runge-Kutta clássico de 4ª ordem (RK4), foi possível resolver o sistema de equações diferenciais que descreve a concentração de um reagente e a temperatura em um reator químico, obtendo-se resultados coerentes e precisos. A validação utilizando a função *solve_ivp* da biblioteca SciPy, com o método RK45, comprovou que o modelo desenvolvido é consistente e adequado para representar o comportamento do sistema.

Os testes com diferentes passos de tempo mostraram que passos menores oferecem maior precisão, mas aumentam o custo computacional. Foi identificado que um passo de 0,01 s é suficiente para garantir uma solução precisa sem sobrecarregar a execução. Já os testes com condições iniciais variadas demonstraram como a escolha dos valores iniciais de temperatura e concentração afeta a dinâmica do sistema, alterando os tempos de pico e valores máximos alcançados. No entanto, todas as curvas convergem para valores de equilíbrio próximos, reforçando a estabilidade do modelo.

Trabalho 1

O estudo aqui proposto pode ser encontrado em https://github.com/ViniciusCMB/Metodos_Num.git

Bibliotecas

In [179…
```
import numpy as np
import matplotlib.pyplot as plt
```

$$\dot{C} = -e^{-\frac{10}{T+273}} * C$$

$$\dot{T} = 1000 * e^{-\frac{10}{T+273}} * C - 10 * (T - 20)$$

Sistema de equações

In [180…
```
def sistema(x, t):
        f1, f2 = x
        dx1dt = -np.exp((-10)/(f2+273))*f1
        dx2dt = (1000*np.exp((-10)/(f2+273))*f1)-(10*(f2-20))
        return np.array([dx1dt, dx2dt])
```

Função para solucionar EDO usando Runge-Kutta clássico de 4ª ordem

In [181…
```
def rk4(f, x0, t):
    x = np.zeros((len(t), len(x0)))
    x[0] = x0
    for i in range(1, len(t)):
        dt = t[i] - t[i-1]
        k1 = f(x[i-1], t[i-1])
        k2 = f(x[i-1] + 0.5*dt*k1, t[i-1] + 0.5*dt)
        k3 = f(x[i-1] + 0.5*dt*k2, t[i-1] + 0.5*dt)
        k4 = f(x[i-1] + dt*k3, t[i])
        x[i] = x[i-1] + (dt/6)*(k1 + 2*k2 + 2*k3 + k4)
    return x
```

Condições iniciais:

In [182…
```
Tini = 35 # temperatura inicial em graus Celsius
Cini = 5 # concentração inicial em gmol/L

x0 = [Cini, Tini] # vetor de condições iniciais
ts = 10 # tempo de simulação em segundos
t = np.linspace(0, ts, 1001) # vetor de tempo de 0 a 10 segundos com 1001 pontos, ou s

t
```

Out[182… `array([ 0.  ,  0.01,  0.02, ...,  9.98,  9.99, 10.  ])`

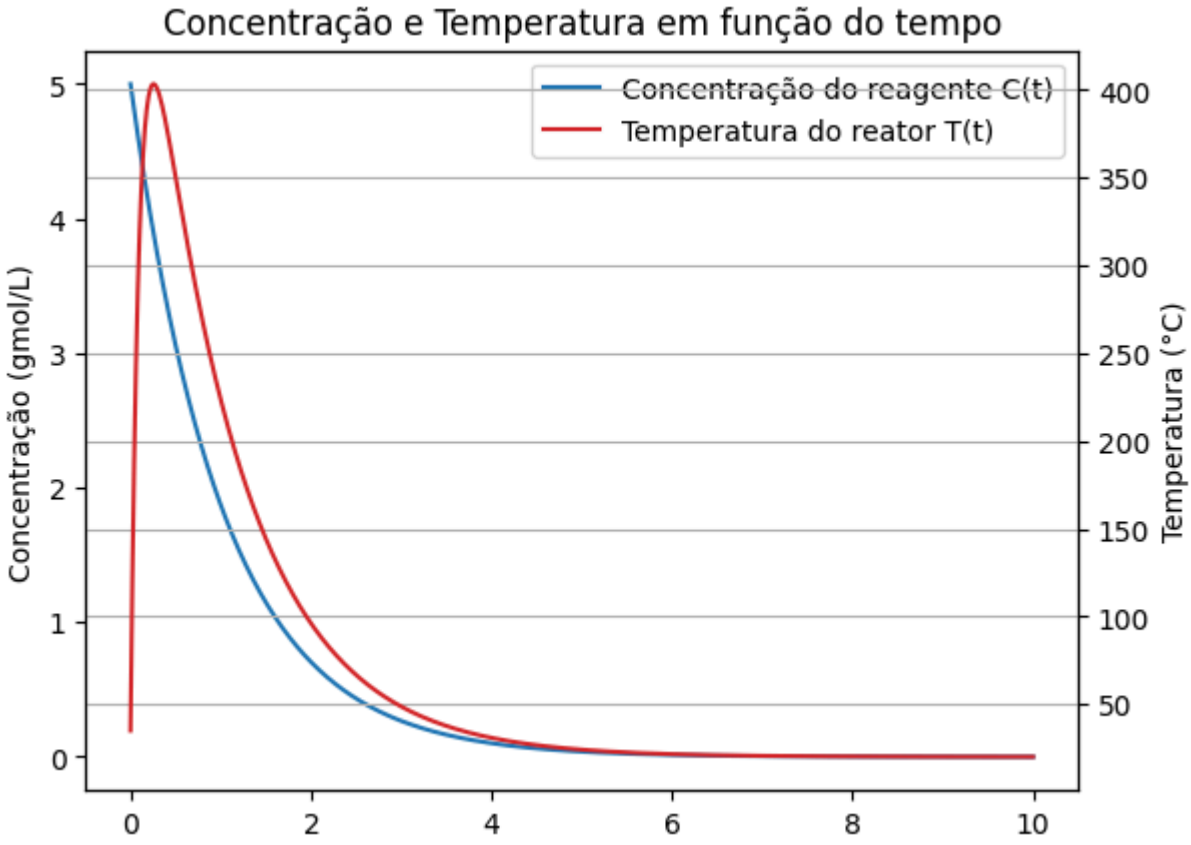
Obtem a solução:

In [183…
```
solucao = rk4(sistema, x0, t)

solucao
```

Out[183…
```
array([[5.00000000e+00, 3.50000000e+01],
       [4.95172630e+00, 7.95088805e+01],
       [4.90375453e+00, 1.19494392e+02],
       ...,
       [3.12818404e-04, 2.00334667e+01],
       [3.09809733e-04, 2.00331449e+01],
       [3.06829999e-04, 2.00328261e+01]])
```

Plota o resultado:

In [184…
```
fig, ax = plt.subplots()
ax2 = ax.twinx() # cria um eixo secundário

plotC = ax.plot(t, solucao[:, 0],label='Concentração do reagente C(t)', color='tab:blu
plotT = ax2.plot(t, solucao[:, 1], label='Temperatura do reator T(t)', color='tab:red

ax.set_ylabel('Concentração (gmol/L)')
ax2.set_ylabel('Temperatura (°C)')
plt.xlabel('Tempo (s)')
titulo = 'Concentração e Temperatura em função do tempo'
plt.title(titulo)
plt.grid()
ax.legend(handles=[plotC[0], plotT[0]])
plt.show()
```

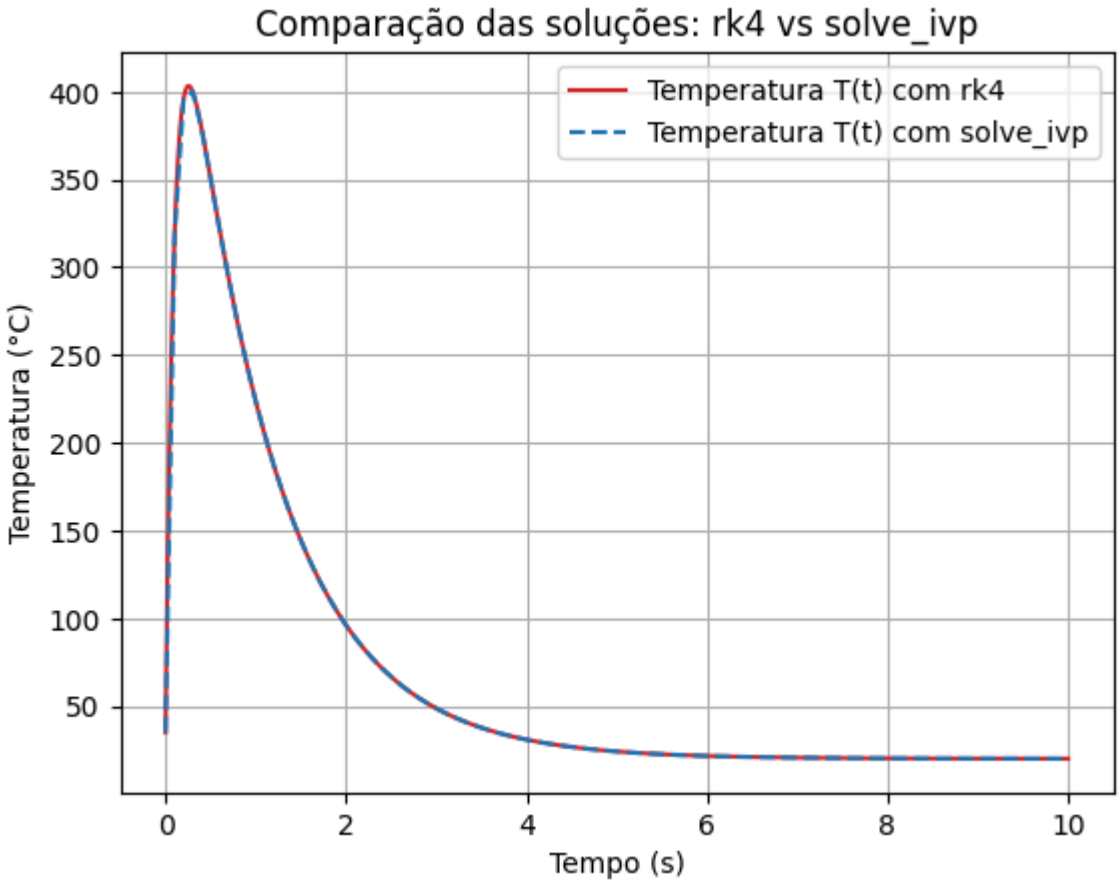


Validando o resultado:

Pacotes conhecidos do python como o sympy.solve_ivp utilizando o Método Runge-Kutta explícito de ordem 5

In [185…
```
from scipy.integrate import solve_ivp

def sistema_ivp(t, x):
    f1, f2 = x
    dx1dt = -np.exp((-10)/(f2+273))*f1
    dx2dt = (1000*np.exp((-10)/(f2+273))*f1)-(10*(f2-20))
    return [dx1dt, dx2dt]
```

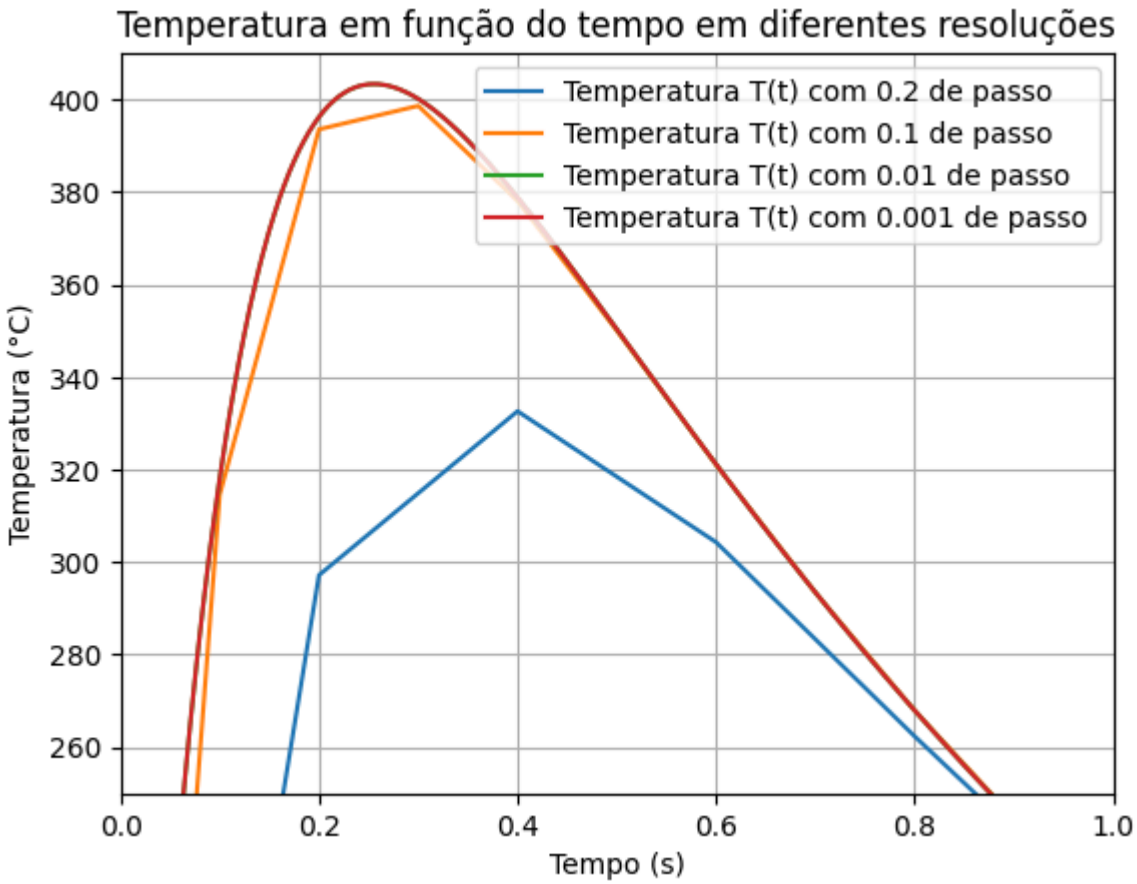
```
sol_ivp = solve_ivp(sistema_ivp, [0, 10], x0, t_eval=np.linspace(0, 10, 101))

plt.plot(t, solucao[:, 1], label='Temperatura T(t) com rk4', color='tab:red')
plt.plot(sol_ivp.t, sol_ivp.y[1], '--', label='Temperatura T(t) com solve_ivp', color=
plt.xlabel('Tempo (s)')
plt.ylabel('Temperatura (°C)')
titulo = 'Comparação das soluções: rk4 vs solve_ivp'
plt.title(titulo)
plt.grid()
plt.legend()
plt.savefig('docs/img/trab1/'+titulo+'.png')
```



Comparando os passos de tempo para solução:

In [186…

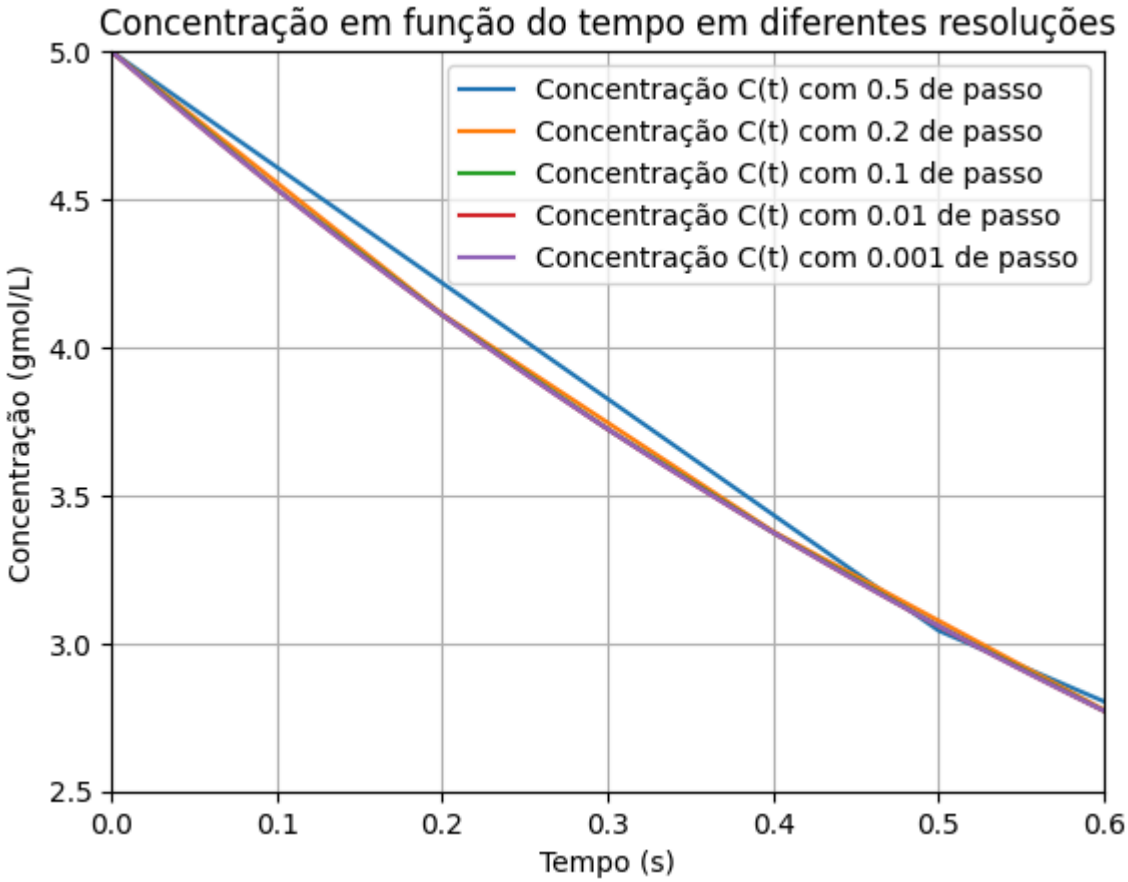
```
Tini = 35  # temperatura inicial em graus Celsius
Cini = 5  # concentração inicial em gmol/L

x0 = [Cini, Tini]  # vetor de condições iniciais
t = [np.linspace(0, 10, 51), np.linspace(0, 10, 101), np.linspace(0, 10, 1001), np.lin

solucao = [rk4(sistema, x0, tempo) for tempo in t]

for tempo, sol in zip(t, solucao):
    plt.plot(tempo, sol[:, 1], label=f'Temperatura T(t) com {tempo[-1]/(len(tempo)-1)}

titulo = 'Temperatura em função do tempo em diferentes resoluções'
plt.title(titulo)
plt.axis([0, 1, 250, 410])
plt.xlabel('Tempo (s)')
plt.ylabel('Temperatura (°C)')
plt.legend(loc='upper right')
plt.grid()
plt.savefig('docs/img/trab1/'+titulo+'.png')
```



O mesmo para a concentração:

In [187…

```
Tini = 35  # temperatura inicial em graus Celsius
Cini = 5  # concentração inicial em gmol/L

x0 = [Cini, Tini]  # vetor de condições iniciais
t = [np.linspace(0, 10, 21), np.linspace(0, 10, 51), np.linspace(0, 10, 101), np.linsp

solucao = [rk4(sistema, x0, tempo) for tempo in t]

for tempo, sol in zip(t, solucao):
    plt.plot(tempo, sol[:, 0], label=f'Concentração C(t) com {tempo[-1]/(len(tempo)-1)

titulo = 'Concentração em função do tempo em diferentes resoluções'
plt.title(titulo)
plt.axis([0, 0.6, 2.5, 5])
plt.xlabel('Tempo (s)')
plt.ylabel('Concentração (gmol/L)')
plt.legend(loc='upper right')
plt.grid()
plt.savefig('docs/img/trab1/'+titulo+'.png')
```



Plotando diferentes condições iniciais

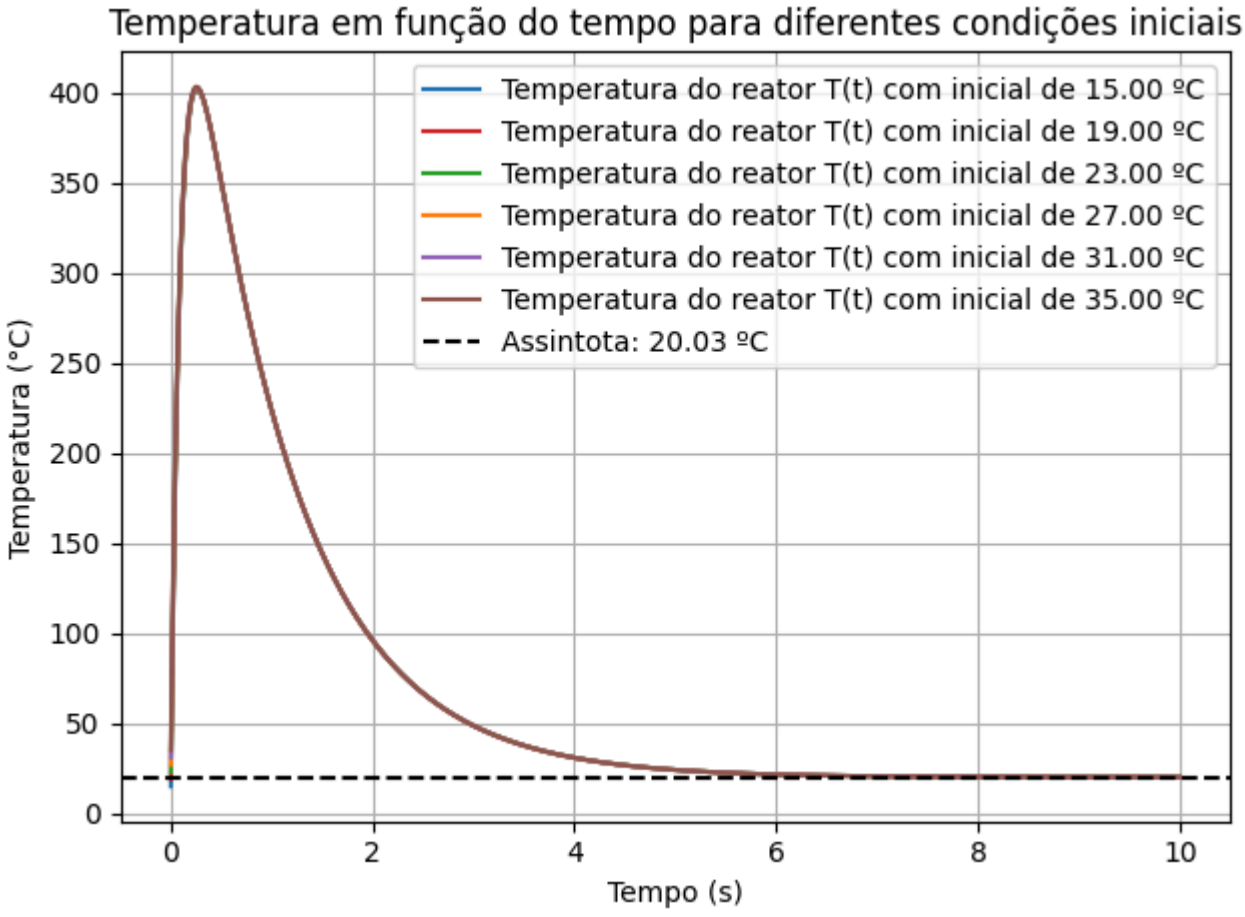
Comparando variações nas condições iniciais de temperatura:

In [190…

```
Tini = np.arange(15, 36, 4)  # temperatura inicial em graus Celsius
Cini = 5  # concentração inicial em gmol/L

x0 = [Cini, Tini]  # vetor de condições iniciais
t = np.linspace(0, 10, 1001)  # vetor de tempo de 0 a 10 segundos com resolução de 0.(

solucao = np.array([rk4(sistema, [Cini, T], t) for T in Tini])

fig, ax = plt.subplots()

plotT = []
cor = ['tab:blue', 'tab:red', 'tab:green', 'tab:orange', 'tab:purple', 'tab:brown']

for i, T in enumerate(Tini):
    plotT.append(ax.plot(t, solucao[i][:, 1], label=f'Temperatura do reator T(t) com :

ax.set_ylabel('Temperatura (°C)')
plt.xlabel('Tempo (s)')
titulo = 'Temperatura em função do tempo para diferentes condições iniciais'
plt.title(titulo)
plt.grid()

# Calcula a média dos últimos valores de cada solução
media_finais = np.mean(solucao[:, -1, 1])

# Plota uma linha preta horizontal na média dos últimos valores
ax.axhline(media_finais, color='black', linestyle='--', label=f'Assintota: {media_fina

# Calcula o valor máximo de cada curva e o tempo em que ocorre
valores_maximos = np.max(solucao[:, :, 1], axis=1)
tempos_maximos = t[np.argmax(solucao[:, :, 1], axis=1)]
# Calcula o desvio padrão da média dos máximos valores
desvio_padrao = np.std(valores_maximos)
```

```
# Printe os pontos máximos
for i, T in enumerate(Tini):
    print(f'Máximo T={valores_maximos[i]:.2f} °C em t={tempos_maximos[i]:.2f} s')

print(f'Média dos últimos valores: {media_finais:.10f} °C')
print(f'Média dos máximos valores: {np.mean(valores_maximos):.2f} °C, desvio padrão: 

# Ajustando as legendas para garantir que todas as linhas sejam identificáveis
handles, labels = ax.get_legend_handles_labels()
ax.legend(handles, labels)

fig.tight_layout()
# plt.show()
plt.savefig('docs/img/trab1/'+titulo+'.png')
```

```
Máximo T=401.81 °C em t=0.26 s
Máximo T=402.12 °C em t=0.26 s
Máximo T=402.42 °C em t=0.26 s
Máximo T=402.72 °C em t=0.26 s
Máximo T=403.03 °C em t=0.26 s
Máximo T=403.33 °C em t=0.26 s
Média dos últimos valores: 20.0328274794 °C
Média dos máximos valores: 402.57 °C, desvio padrão: 0.52 °C
```



Continuando similar com a concentração:

In [194…

```
Cini = np.arange(5, 36, 6)  # concentração inicial em gmol/L
Tini = 15  # temperatura inicial em graus Celsius

x0 = [Cini, Tini]  # vetor de condições iniciais
t = np.linspace(0, 10, 101)  # vetor de tempo de 0 a 10 segundos com resolução de 0.1

solucao = np.array([rk4(sistema, [C, Tini], t) for C in Cini])

fig, ax = plt.subplots()
```

```
plotC = []
cor = ['tab:blue', 'tab:red', 'tab:green', 'tab:orange', 'tab:purple', 'tab:brown']

for j, C in enumerate(Cini):
    plotC.append(ax.plot(t, solucao[j][:, 0], label=f'Concentração do reagente C(t) co

titulo = f'Temperatura inicial {Tini} °C e concentração inicial {Cini} gmol/L'
ax.set_ylabel('Concentração (gmol/L)')
plt.xlabel('Tempo (s)')
titulo = 'Concentração em função do tempo para diferentes condições iniciais'
plt.title(titulo)
plt.grid()

# Calcula a média dos últimos valores de cada solução
media_finais = np.mean(solucao[:, -1, 0])

# Plota uma linha preta horizontal na média dos últimos valores
ax.axhline(media_finais, color='black', linestyle='--', label=f'Assintota: {media_fina

print(f'Média dos últimos valores: {media_finais:.10f} gmol/L')

handles, labels = ax.get_legend_handles_labels()
ax.legend(handles, labels)

fig.tight_layout()
# plt.show()
plt.savefig('docs/img/trab1/'+titulo+'.png')
```

```
Média dos últimos valores: 0.0011775915 gmol/L
```