FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D
COIMBRA

Luís Januário
António França Oliveira

Defesa Oral
Física Computacional

Este documento serve como preparação e apoio para a defesa oral das fichas realizadas ao longo do segundo semestre para a cadeira de *Física Computacional*

abril 2024

Conteúdo

Ficha 1

1.1 Exercício 1

```
>>> x=1
>>> e=1e-20
>>> x+e
1.0
>>> x+e-x
0.0
>>> e+x-x
0.0
>>> x-x+e
1e-20
```

(a) A expressão "$\mathbf{x - x + e}$" retorna o valor correto de **e** porque, ao realizar operações aritméticas, adição e subtração têm a mesma precedência e são avaliadas da esquerda para a direita. Neste caso, "$\mathbf{x - x}$" anula-se, resultando em **0**, e a adição de **e** a **0** retorna o valor correto de **e**.

No entanto, as expressões "$\mathbf{x + e - x}$" e "$\mathbf{e + x - x}$" não retornam o valor correto de **e** devido às limitações da aritmética de ponto flutuante. Em Python, números de ponto flutuante são representados usando um número finito de bits, o que leva a erros de arredondamento. Ao somar ou subtrair dois números com magnitudes muito diferentes, como **x** e **e** neste caso, o valor menor (**e**) pode ser "perdido"na precisão do valor maior (**x**). Como resultado, a subtração pode não resultar no valor exato de **e**.

(b) Se usarmos $\mathbf{e = 0.01}$ em vez de $\mathbf{e = 1e - 20}$, as expressões retornam resultados diferentes. A expressão "$\mathbf{x + e - x}$" e "$\mathbf{e + x - x}$" ambas irão avaliar para **0.01** porque a magnitude de **e** não é pequena o suficiente para ser "perdida"na precisão de **x**. Erros de aritmética de ponto flutuante tornam-se mais aparentes ao lidar com valores extremamente pequenos como $\mathbf{1e^{-20}}$.

Em resumo, a precisão e as limitações da aritmética de ponto flutuante em Python podem afetar os resultados das operações aritméticas, especialmente ao combinar valores com magnitudes significativamente diferentes. É importante estar ciente dessas limitações ao trabalhar com números de ponto flutuante.

1.2 Exercício 2

```
d = 0.1
x = 0.0
while x != 1.0:
    print (x)
    x += d
```

O programa não para quando **x** atinge o valor de **1.0** devido à forma como os números de ponto flutuante são representados e comparados nos sistemas de computação. Devido a limitações de precisão, o valor exato de **1.0** nem sempre pode ser representado com exatidão.

Para modificar o programa de forma a que pare assim que **x** atingir o valor de **1.0**, você pode alterar a condição $\mathbf{x} \neq \mathbf{1.0}$ para $\mathbf{x} < \mathbf{1.0}$. Dessa forma, o loop continuará até que **x** se torne maior que ou igual a **1.0**. Aqui está o programa modificado com a secção de comentário solicitada:

```
# Programa para imprimir n meros de 0.0 a 1.0 em incrementos de 0.1

# Inicializar o tamanho do passo e x
d = 0.1
x = 0.0

# Loop at  que x atinja 1.0
while x < 1.0:
    # Imprimir o valor atual de x
    print(x)

    # Incrementar x pelo tamanho do passo
    x += d
```

Neste programa modificado, a condição $\mathbf{x} < \mathbf{1.0}$ garante que o loop continue até que **x** se torne maior ou igual a **1.0**, momento em que ele irá parar.

Note-se que, devido a limitações de precisão dos números de ponto flutuante, mesmo com essa modificação, o loop pode não terminar exatamente em **1.0** em alguns casos, mas ele irá parar quando **x** atingir um valor muito próximo de **1.0**.

1.3 Exercício 3

```
n = 0
a = 1

while a > 0:
    n = n + 1
    print(n)
    a = (1.0 + 2.0**(-n)) - 1.0
```

(a) O programa para porque a variável **a** atinge um valor tão pequeno que é considerado insignificante em termos de precisão dos números de ponto flutuante. Apesar da expressão matemática indicar que **a** deveria ser maior que **0**, devido à limitação de precisão dos números de ponto flutuante, quando **n** se torna suficientemente grande, a operação $(\mathbf{1.0} + \mathbf{2.0^{-n}}) - \mathbf{1.0}$ resulta em um valor muito próximo de zero, mas não exatamente zero. Portanto, a condição $\mathbf{a} > \mathbf{0}$ não é mais satisfeita e o loop é interrompido.

O último valor de **n** antes de o programa parar representa o número de iterações necessárias para atingir o ponto em que a condição $\mathbf{a} > \mathbf{0}$ não é mais satisfeita.

(b) Quando alteramos a linha para $\mathbf{a} = \mathbf{2.0^{-n}}$, a expressão $\mathbf{1.0} + \mathbf{2.0^{-n}} - \mathbf{1.0}$ é simplificada para apenas $\mathbf{2.0^{-n}}$. Nesse caso, a variável **a** assume diretamente o valor de **2** elevado a $-\mathbf{n}$. Conforme **n** aumenta, o valor de **a** torna-se cada vez menor, aproximando-se de zero. Como a condição do loop é $\mathbf{a} > \mathbf{0}$, assim que a se torna menor ou igual a zero, o loop é interrompido. Portanto, o programa ainda termina, mas o critério para a interrupção é diferente.

O último valor de **n** agora representa o número de iterações necessárias para que **a** se torne menor ou igual a zero.

(c)

```
def find_sigfigs(a):
    '''Returns the number of significant digits in a number. This takes into
    account
       strings formatted in 1.23e+3 format and even strings such as 123.450'''
    # change all the 'E' to 'e'
    a = a.lower()
    if 'e' in a:
        # return the length of the numbers before the 'e'
        myStr = a.split('e')
        return len(myStr[0]) - 1  # to compensate for the decimal point
    else:
        # put it in e format and return the result of that
        m = ('%.*e' % (20, float(a))).split('e')
        # remove and count the number of removed user added zeroes. (these are
    sig figs)
        if '.' in a:
            s = a.replace('.', '')
            # number of zeroes to add back in
            l = len(s) - len(s.rstrip('0'))
            # strip off the Python added zeroes and add back in the ones the
    user added
            m[0] = m[0].rstrip('0') + ''.join(['0' for _ in range(l)])
        else:
            # the user had no trailing zeroes, so just strip them all
            m[0] = m[0].rstrip('0')
        # pass it back to the beginning to be parsed
    return find_sigfigs('e'.join(m))


n = 0
```

```
28 a = 1
29 while a > 0:
30     n = n + 1
31     print('Number of significant numbers:', find_sigfigs(str(a)))
32     print(a)
33     a = (1.0 + 2.0**(-n)) - 1.0
```

A função **'find_sigfigs(a)'** é definida para calcular o número de dígitos significativos em um número. Essa função leva em consideração a notação científica e trata dos zeros finais.

O código começa por verificar se o carácter **'e'** está presente em **'a'**. Se estiver, isso indica que **'a'** está no formato de notação científica, como **'1,23e+3'**. Nesse caso, o código divide **'a'** em duas partes: a parte antes do **'e'** e a parte depois do **'e'**. A função retorna o comprimento da parte antes do **'e'**, subtraindo 1 para compensar o ponto decimal.

Se **'e'** não estiver presente em **'a'**, isso indica que **'a'** está no formato decimal, como **'123,450'**. Nesse caso, o código converte **'a'** em notação científica usando **'%.*e' % (20, float(a))'**. Isso garante que **'a'** seja representado com até 20 dígitos significativos. Em seguida, a notação científica é dividida em duas partes: o coeficiente e o expoente, separados pelo **'e'**.

O código verifica se **'a'** contém uma vírgula decimal. Se contiver, isso indica que existem zeros finais que podem ser significativos. O código remove a vírgula decimal de **'a'** e conta o número de zeros removidos pelo utilizador. Esses zeros são considerados dígitos significativos. Em seguida, o código ajusta o coeficiente, removendo quaisquer zeros finais adicionados automaticamente pelo Python e adicionando de volta os zeros finais que o utilizador adicionou.

Se **'a'** não contiver uma vírgula decimal, isso significa que não há zeros finais a serem considerados. Nesse caso, o código remove todos os zeros finais do coeficiente.

Em seguida, a função chama recursivamente **'find_sigfigs('e'.join(m))'** para calcular o número de dígitos significativos na notação científica ajustada. A repetição é usada para lidar com casos em que a notação científica ajustada também contém uma vírgula decimal.

O programa principal inicia duas variáveis: **'n'** é inicializada como 0 e **'a'** é inicializada como 1. Em seguida, entra em um loop **'while'** que continua enquanto **'a'** for maior que 0. Dentro do loop, a variável **'n'** sofre incrementos de 1 para acompanhar o número de iterações. O programa imprime o valor atual de **'a'** usando o comando **'print(a)'**. Em seguida, o programa chama a função **'find_sigfigs(str(a))'** para calcular o número de dígitos significativos em **'a'**. Esse valor é impresso usando **'print('Número de dígitos significativos:', find_sigfigs(str(a)))'**. A variável **'a'** é atualizada usando a fórmula **'a = (1.0 + 2.0**(-n)) - 1.0'**. Isso é feito para gerar uma sequência de valores de **'a'** que se aproxima cada vez mais de 1. O programa executa o loop até que **'a'** seja menor ou igual a 0. Nesse ponto, o loop é interrompido e o programa termina.

Em resumo, o programa calcula uma sequência de valores de **'a'** usando uma fórmula específica e, em cada iteração, imprime o valor atual de **'a'** e o número de dígitos significativos nesse valor. O objetivo é observar como o número de dígitos significativos muda à medida que a sequência se aproxima de 1.

1.4 Exercício 4

```python
def binomial_coeff(n, k):
    if not isinstance(n, int) or not isinstance(k, int) or n < 0 or k < 0 or k > n:
        return None

    # Compute using the smaller value of (n-k)
    if k > n-k:
        k = n-k

    # Compute the binomial coefficient using the formula
    res = 1
    for i in range(1, k+1):
        res = res * (n - (k - i)) // i

    return res
```

A função **'binomial_coeff(n, k)'** é definida para calcular o coeficiente binomial (ou coeficiente de combinação) **"n choose k"**, que representa o número de maneiras de escolher **k** elementos de um conjunto de **n** elementos. O coeficiente binomial é calculado usando a fórmula matemática:

$$C(n,k) = \frac{n!}{k! \times (n-k)!}$$

A função começa por verificar se os argumentos **n** e **k** são inteiros não negativos. Caso contrário, ela retorna **None**. Em seguida, é determinado qual é o menor valor entre **k** e **(n - k)**, pois o coeficiente binomial é simétrico em relação a esses dois valores.

Em seguida, o coeficiente binomial é calculado usando um loop. A variável **res** é inicializada como 1 e, em cada iteração, o valor de **res** é atualizado multiplicando-o pelo próximo termo do coeficiente binomial e dividindo pelo índice do loop. Essa técnica é usada para evitar a ocorrência de erros de ponto flutuante durante os cálculos, pois o operador // realiza a divisão inteira.

A função retorna o valor do coeficiente binomial.

```python
def binom(n, m):
    """
    Computes the probability of obtaining m heads in n coin tosses, where the probability of heads is 1/2.
    """
    if not isinstance(n, int) or not isinstance(m, int) or n < 0 or m < 0 or m > n:
        return None

    # Compute the probability using the binomial coefficient function
    coeff = binomial_coeff(n, m)
```

```
    p = coeff * (1/2)**n

    return p
```

A função **binom(n, m)** calcula a probabilidade de obter **m** vezes a face *"heads"* em **n** lançamentos de uma moeda, onde a probabilidade de obter *"heads"* é de 1/2.

Assim como na função anterior, a função verifica se **n** e **m** são inteiros não negativos. Caso contrário, ela retorna **None**.

Em seguida, a função calcula o coeficiente binomial chamando a função **binomial_coeff(n, m)**. Esse coeficiente representa o número de combinações possíveis de obter **m** *"heads"* em **n** lançamentos.

A probabilidade é calculada multiplicando o coeficiente binomial pelo valor $(1/2)^n$, onde $(1/2)$ é a probabilidade de obter *"heads"* em um único lançamento da moeda.

A função retorna o valor da probabilidade.

```
n = int(input("Enter the number of coin tosses: "))
m = int(input("Enter the number of heads: "))

prob = binom(n, m)

print("The probability of obtaining", m, "heads in", n, "coin tosses is", prob)
```

As últimas linhas do código solicitam ao usuário que insira o número de lançamentos da moeda **(n)** e o número de *"heads"* desejados **(m)**. Esses valores são convertidos em inteiros usando a função **int()**.

Em seguida, a função **binom(n, m)** é chamada para calcular a probabilidade de obter **m** *"heads"* em **n** lançamentos. O resultado é armazenado na variável **prob**.

Por fim, a função **print()** é usada para exibir a probabilidade calculada na forma de uma frase que descreve o número de *"heads"* e o número de lançamentos.

1.5 Exercício 5

```
def factorial(n):
    """
    Computes the factorial of a non-negative integer n.
    """
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

A função **factorial(n)** é definida para calcular o fatorial de um número inteiro não negativo **n**. O fatorial de **n** é o produto de todos os números inteiros positivos menores ou iguais a **n**. A

função é implementada de forma recursiva. Começando por verificar se **n** é igual a 0. Se for, retorna 1, pois o fatorial de 0 é definido como 1. Caso contrário, a função retorna o valor de **n** multiplicado pelo fatorial de **n-1**. Essa repetição continua até que **n** seja igual a 0, onde a função retorna 1 e encerra a repetição.

```
def binom(n, m):
    """
    Computes the probability of obtaining m heads in n coin tosses, where the
    probability of heads is 1/2.
    """
    if not isinstance(n, int) or not isinstance(m, int) or n < 0 or m < 0 or m >
     n:
        return None
    else:
        return factorial(n) // (factorial(m) * factorial(n-m)) * 2**(-n)
```

A função **binom(n, m)** calcula a probabilidade de obter**m** vezes a face *"heads"* em **n** lançamentos de uma moeda, onde a probabilidade de obter *"heads"* é de 1/2.

Assim como nas explicações anteriores, a função verifica se **n** e **m** são inteiros não negativos e se **m** não é maior do que **n**. Caso contrário, retorna **None**.

Em seguida, a função usa a função **factorial()** para calcular o fatorial de **n**, **m** e **n-m**. O coeficiente binomial é calculado dividindo-se o fatorial de **n** pelo produto do fatorial de **m** e do fatorial de **n-m**. O resultado é multiplicado por dois elevado a menos **n**.

Por fim, a função retorna o resultado dessa fórmula, representando a probabilidade de obter **m** vezes *"heads"* em **n** lançamentos de uma moeda.

```
n = int(input("Enter the number of coin tosses: "))

# Initialize the sum of probabilities to zero
total_prob = 0

# Iterate over all possible values of m and compute the probability for each
    value
for m in range(n+1):
    prob = binom(n, m)
    if prob is not None:
        print("The probability of obtaining", m, "heads in", n, "coin tosses is"
    , prob)
        total_prob += prob
    else:
        print("Invalid input.")

# Print the sum of all probabilities
print("The sum of probabilities is", total_prob)
```

O código solicita ao usuário que insira o número de lançamentos da moeda **(n)** por meio da

função **input()**. Em seguida, é inicializada uma variável **total_prob** para armazenar a soma de todas as probabilidades calculadas.

Um loop *for* é usado para iterar sobre todos os valores possíveis de **m**, variando de 0 a **n**. Para cada valor de **m**, a função **binom(n, m)** é chamada para calcular a probabilidade de obter **m** vezes *"heads"* em **n** lançamentos.

Se a probabilidade calculada for diferente de **None**, ou seja, se a entrada for válida, o programa imprime a probabilidade no formato **"A probabilidade de obter *m* heads em *n* coin tosses is *prob*"** e adiciona essa probabilidade à variável **total_prob**.

Caso a probabilidade seja **None**, o programa imprime **"Invalid input."** indicando que a entrada não é válida.

Por fim, o programa imprime a soma de todas as probabilidades calculadas usando a mensagem **"The sum of probabilities is *total_prob*"**. Isso representa a soma de todas as probabilidades de obter um número específico de *"heads"* em **n** lançamentos da moeda.

1.6 Exercício 6

```
def factorial(n):
    """
    Calcula o fatorial de um n mero inteiro n o negativo n.
    """
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

A função **factorial(n)** calcula o fatorial de um número inteiro não negativo **n**. Ela é implementada usando uma recursão. Se o valor de **n** for igual a zero, a função retorna 1. Caso contrário, a função retorna **n** multiplicado pelo fatorial de **n-1**, ou seja, **n * factorial(n-1)**.

```
def binom(n, m):
    """
    Calcula a probabilidade de obter m "heads" em n lancamentos de moeda, onde a
     probabilidade de "heads"    1/2.
    """
    if not isinstance(n, int) or not isinstance(m, int) or n < 0 or m < 0 or m >
     n:
        return None
    else:
        return factorial(n) // (factorial(m) * factorial(n-m)) * 2**(-n)
```

A função **binom(n, m)** calcula a probabilidade de obter **m** *"heads"]* em **n** lançamentos de moeda, onde a probabilidade de *"heads"* é 1/2. Ela verifica se **n** e **m** são inteiros não negativos e se **m** é menor ou igual a **n**. Caso contrário, ela retorna **None** para indicar uma entrada inválida.

Se os valores de entrada forem válidos, a função calcula a probabilidade utilizando a fórmula do coeficiente binomial: **factorial(n) // (factorial(m) * factorial(n-m))** multiplicado por **2**(-n)**.

```
n = int(input("Enter the number of coin tosses: "))
m = int(input("Enter the number of heads: "))

prob = binom(n, m)

if prob is not None:
    print("The probability of obtaining", m, "heads in", n, "coin tosses is",
    prob)
else:
    print("Invalid input.")
```

No programa principal, o usuário é solicitado a inserir o número de lançamentos de moeda **(n)** e o número de *"heads"* **(m)** através das funções **input()**. Em seguida, a função **binom(n, m)** é chamada para calcular a probabilidade.

Se a probabilidade calculada for diferente de **None**, o programa imprime a probabilidade no formato **"A probabilidade de obter *m* heads em *n* coin tosses is *prob*"**. Caso contrário, o programa imprime **"Invalid input."** para indicar que a entrada fornecida é inválida.

1.6.1 Correção exercício 6

O código não cumpriu com o que foi pedido neste exercício. Para que tivesse correto deveria-se retirar a função **factorial(n)** e substituir a função **binom(n, m)** por:

```
def binom(n, m):
    """
    Calcula a probabilidade de obter m "heads" em n lan amentos de moeda, onde
    a probabilidade de "heads"    1/2.
    """
    if not isinstance(n, int) or not isinstance(m, int) or n < 0 or m < 0 or m >
     n:
        return None

    numerator = 1
    denominator = 1

    for i in range(m):
        numerator *= (n - i)
        denominator *= (i + 1)

    return numerator // denominator * 2**(-n)
```

Para melhorar a eficiência do cálculo e lidar com valores grandes de **n**, podemos fazer uso da propriedade do coeficiente binomial conhecida como *"cancelamento parcial"*. Essa propriedade

permite reduzir o número de cálculos de fatoriais necessários.

Nesta nova implementação, substituímos os cálculos de fatoriais pelos loops *for*. O loop *for* itera de 0 a **m-1** e atualiza o **numerator** multiplicando por **(n - i)** a cada iteração, enquanto atualiza o **denominator** multiplicando por **(i + 1)**.

Esta abordagem reduz significativamente o número de cálculos necessários e melhora a eficiência do programa ao lidar com valores grandes de **n**, uma vez que não precisamos calcular fatoriais completos.

No entanto, é importante ressaltar que mesmo com essa melhoria, a computação de coeficientes binomiais para valores muito grandes de **n** ainda pode ser desafiadora devido ao crescimento exponencial do número de iterações e ao potencial estouro de inteiro. Portanto, para valores extremamente grandes de **n**, podem ser necessárias técnicas adicionais, como a utilização de algoritmos mais avançados, aritmética de precisão arbitrária ou métodos de aproximação.

1.7 Exercício 7

```
1  def factorial(n):
2      """
3      Computes the factorial of a non-negative integer n.
4      """
5      if n == 0:
6          return 1
7      else:
8          return n * factorial(n-1)
9
10 def binom(n, m):
11     """
12     Computes the probability of obtaining m heads in n coin tosses, where the
       probability of heads is 1/2.
13     """
14     if not isinstance(n, int) or not isinstance(m, int) or n < 0 or m < 0 or m >
        n:
15         return None
16
17     # Compute using the smaller value of (n-m)
18     if m > n-m:
19         m = n-m
20
21     # Compute the probability using the formula
22     p = 1
23     for i in range(1, m+1):
24         p = (n - (m - i)) / i
25         p= 0.5**(n)
26
27     return p
28
29
```

```
30 n = int(input("Enter the number of coin tosses: "))
31 m = int(input("Enter the number of heads: "))
32
33 prob = binom(n, m)
34
35 print("The probability of obtaining", m, "heads in", n, "coin tosses is", prob)
```

Este código calcula a probabilidade de obter um número específico de caras em um número específico de lançamentos. Ele define duas funções: **factorial** e **binom**. A função **factorial** calcula o fatorial de um número inteiro não negativo **n**. A função **binom** calcula a probabilidade de obter **m** caras em **n** lançamentos de moeda, onde a probabilidade de cara é 1/2.

O problema com o código é que ele não calcula corretamente a probabilidade na função **binom**. A linha **p = (n - (m - i)) / i** deve ser substituída por **p *= (n - (m - i)) / i**. Além disso, a linha **p= 0.5**(n)** deve ser movida para fora do loop e alterada para **p *= 0.5**(n)**.

O código acima não é capaz de calcular a probabilidade para números grandes de **n** e **m** devido a problemas de precisão numérica. Isso ocorre porque o cálculo é realizado usando a fórmula direta, o que resulta em operações com números muito grandes e pequenos, levando a erros de arredondamento.

1.7.1 Correção exercício 7

O código correto seria o seguinte:

```
1  def factorial(n):
2      if n == 0:
3          return 1
4      prod = 1
5      for i in range(1, n+1):
6          prod *= i
7      return prod
8
9  def factorial2(n, m):
10     if n >= m:
11         s = n
12         p = 1
13         while s > m:
14             p *= s
15             s -= 1
16         return p
17
18 def binom(n, m):
19     P = factorial2(n, m) / (factorial(n - m) * (2 ** n))
20     return P
21
22 print("Probabilidade de lan ar 100,0000 moedas e obter 50,000 caras:", binom
       (100000, 50000))
```

```
23 print("Probabilidade de lan ar 10,000 moedas e obter 5,000 caras:", binom
    (10000, 5000))
```

Neste código, há algumas alterações que ajudam a lidar com números grandes de forma mais eficiente.

A função **factorial** foi modificada para usar um loop em vez de uma chamada recursiva. Embora essa alteração não resolva diretamente o problema de números grandes, ela melhora o desempenho do cálculo do fatorial.

A função **factorial2** foi adicionada para calcular o fatorial parcial necessário para o cálculo da probabilidade binomial. Ela calcula o fatorial apenas para os valores relevantes (de **n** a **m+1**), reduzindo a quantidade de operações envolvidas.

A fórmula para o cálculo da probabilidade binomial foi reescrita de forma mais eficiente. Em vez de calcular o fatorial de **(n - m)** e o expoente de 2 separadamente, a função **factorial2** é usada para calcular diretamente o fatorial parcial necessário.

Essas alterações ajudam a evitar problemas de precisão numérica e melhoram o desempenho geral para números grandes.

Ficha 2

2.1 Exercício 1

```
def f(x):
    f=25*x**(4)-x**(2)/2-2
    return f

def df(x):
    df=100*x**(3)-x
    return df
```

Definimos a função **f(x)** e a sua derivada em relação de **x**.

$$f(x) = 25x^4 - \frac{x^2}{2} - 2$$

$$\frac{df}{dx} = 100x^3 - x$$

Estas funções são usadas nos métodos numéricos para calcular os valores da função e da sua derivada em pontos específicos.

```
# M todo da bissec  o
def bisec(a, b, k_max, eps_1, eps_2):
    k = 1
    c = (1/2)*(a + b)

    while k < k_max:
        k += 1
        c = (1/2)*(a + b)

        if abs(f(c)) < eps_1 and b - a < eps_2:
            print(f"Converge em {k} itera  es.")
            return c

        if f(a) * f(c) < 0:
            b = c
```

```
        else:
            a = c

    print("N o foi poss vel encontrar a raiz ap s", k_max, "itera  es.")
    return c
```

Em seguida, temos a implementação do método da bissecção, que é utilizado para encontrar a raiz de uma função. Ele recebe cinco parâmetros: **a** e **b**, que são os limites do intervalo onde a raiz será procurada; **k_max**, que é o número máximo de iterações permitidas; e **eps_1** e **eps_2**, que são os critérios de parada do método.

```
# M todo da Secante
def sec(a, b, k_max, eps_1, eps_2):
    k = 1
    c = a + f(a) * ((a - b) / (f(b) - f(a)))

    while k < k_max:
        a = b
        b = c
        c = a + f(a) * ((a - b) / (f(b) - f(a)))

        if abs(f(c)) < eps_1 and b - a < eps_2:
            print(f"Converge em {k} itera  es.")
            return c

        k += 1

    if k == k_max:
        print("N o foi poss vel encontrar a raiz ap s", k_max, "itera  es."
    )
    else:
        print("A raiz   :", c)

    return c
```

O próximo método é a implementação do método da secante. Assim como o método da bissecção, recebe os mesmos cinco parâmetros e também é usado para encontrar a raiz da função.

```
# M todo de Newton
def newton(f, df, x0, eps_1, eps_2, k_max):
    x = x0
    k = 1

    while k < k_max:
        x_prev = x
        x = x - f(x) / df(x)

```

```
        if abs(f(x)) < eps_1 and abs(x - x_prev) < eps_2:
            print(f"Converge em {k} itera  es.")
            return x

        k += 1

    print(f"Falhou em convergir em {k_max} itera  es.")
    return x
```

O último método é o método de Newton, que também é utilizado para encontrar a raiz de uma função. Ele recebe os mesmos parâmetros que os outros métodos.

```
# Escolha do M todo Pretendido
method = input("Qual o m todo que deseja utilizar?\n1. Bisse  o \n2. Secante
    \n3. Newton's \nDigite o n mero correspondente: ")

while method not in ['1', '2', '3']:
    method = input("M todo inv lido. Tente novamente.\nQual o m todo que
    deseja utilizar?\n1. Bise  o \n2. Secante \n3. Newton's \nDigite o n mero
     correspondente: ")
```

Aqui o usuário é solicitado a escolher um dos três métodos para encontrar a raiz da função. Se a opção digitada não for válida (1, 2 ou 3), o usuário será solicitado novamente a fazer a escolha.

Depois, dependendo do método escolhido, o usuário é solicitado a inserir os parâmetros específicos para cada método e, finalmente, o resultado é exibido na tela.

Este código permite que o usuário escolha entre três métodos numéricos diferentes para encontrar a raiz de uma função e, com base na escolha, inserir os parâmetros necessários para executar o método selecionado.

2.1.1 Correção exercício 1

```
def f(x):
    f = 25 * x ** 4 - x ** 2 / 2 - 2
    return f

def df(x):
    df = 100 * x ** 3 - x
    return df

def bisec(a, b, k_max, eps_1, eps_2):
    k = 1
    c = (a + b) / 2

    while k <= k_max:
        print(f"Iteration {k}: a={a}, b={b}, c={c}, f(c)={f(c)}")

```

```
16          if abs(f(c)) < eps_1 and b - a < eps_2:
17              print(f"Converged after {k} iterations.")
18              return c
19
20          if f(a) * f(c) < 0:
21              b = c
22          else:
23              a = c
24
25          c = (a + b) / 2
26          k += 1
27
28      print("Failed to find the root after", k_max, "iterations.")
29      return c
30
31  def sec(a, b, k_max, eps_1, eps_2):
32      k = 1
33      c = a + f(a) * ((a - b) / (f(b) - f(a)))
34
35      while k <= k_max:
36          print(f"Iteration {k}: a={a}, b={b}, c={c}, f(c)={f(c)}")
37
38          if abs(f(c)) < eps_1 and b - a < eps_2:
39              print(f"Converged after {k} iterations.")
40              return c
41
42          a = b
43          b = c
44          c = a + f(a) * ((a - b) / (f(b) - f(a)))
45          k += 1
46
47      print("Failed to find the root after", k_max, "iterations.")
48      return c
49
50  def newton(f, df, x0, eps_1, eps_2, k_max):
51      x = x0
52      k = 1
53
54      while k <= k_max:
55          print(f"Iteration {k}: x={x}, f(x)={f(x)}")
56
57          x_prev = x
58          x = x - f(x) / df(x)
59
60          if abs(f(x)) < eps_1 and abs(x - x_prev) < eps_2:
61              print(f"Converged after {k} iterations.")
62              return x
63
64          k += 1
65
```

```
66      print(f"Failed to converge after {k_max} iterations.")
67      return x
68
69
70  method = input("Which method do you want to use?\n1. Bisection\n2. Secant\n3.
        Newton\nEnter the corresponding number: ")
71
72  while method not in ['1', '2', '3']:
73      method = input("Invalid method. Try again.\nWhich method do you want to use
        ?\n1. Bisection\n2. Secant\n3. Newton\nEnter the corresponding number: ")
74
75  if method == '1':
76      a = float(input("Enter the value of a: "))
77      b = float(input("Enter the value of b: "))
78      k_max = int(input("Enter the maximum number of iterations: "))
79      eps_1 = float(input("Enter the value of eps_1: "))
80      eps_2 = float(input("Enter the value of eps_2: "))
81
82      result = bisec(a, b, k_max, eps_1, eps_2)
83      print(f"The root of the function is: {result}")
84
85  elif method == '2':
86      a = float(input("Enter the value of a: "))
87      b = float(input("Enter the value of b: "))
88      k_max = int(input("Enter the maximum number of iterations: "))
89      eps_1 = float(input("Enter the value of eps_1: "))
90      eps_2 = float(input("Enter the value of eps_2: "))
91
92      result = sec(a, b, k_max, eps_1, eps_2)
93      print(f"The root of the function is: {result}")
94
95  else:
96      x0 = float(input("Enter the value of x0: "))
97      k_max = int(input("Enter the maximum number of iterations: "))
98      eps_1 = float(input("Enter the value of eps_1: "))
99      eps_2 = float(input("Enter the value of eps_2: "))
100
101     result = newton(f, df, x0, eps_1, eps_2, k_max)
102     print(f"The root of the function is: {result}")
```

Alterações realizadas:

- Nas funções **bisec** e **sec**, foram adicionadas instruções de impressão para mostrar os valores atuais dos parâmetros em cada iteração, tal como nas tabelas da apresentação da aula.

- Foram removidas as instruções de impressão desnecessárias relacionadas à convergência na função **newton**, uma vez que a convergência já é indicada em cada iteração.

2.2 Exercício 2

```
""" Fun   o e respetiva derivada  """

def g(x):
    g = np.log(x) + (1 / x ** 2) - 1
    return g

def dg(x):
    dg = (1 / x) - (2 / x ** 3)
    return dg
```

Aqui definimos duas funções: g(x) e dg(x). A função g(x) representa uma função matemática que é a soma do logaritmo natural de x, $1/x^2$ e **-1**. A função **dg(x)** representa a derivada da função **g(x)**. Ambas as funções foram definidas usando o **numpy** para permitir operações matemáticas eficientes.

```
#M todo de Newton
def newton(g, dg, x0, eps_1, eps_2, k_max):
    x = x0
    k = 1
    while k < k_max:
        x_prev = x
        x = x - g(x) / dg(x)
        if abs(g(x)) < eps_1 and abs(x - x_prev) < eps_2:
            print(f"Converge em {k} itera  es.")
            return x
        k += 1
    print(f"Falhou em convergir em {k_max} itera  es.")
    return x
```

Aqui definimos a função **newton(g, dg, x0, eps_1, eps_2, k_max)** que implementa o método de Newton para encontrar a raiz de uma função **g(x)**. O método de Newton é uma técnica iterativa que utiliza a derivada da função para encontrar a raiz.

Dentro desta função, **x** representa o valor inicial, e o método iterativamente atualiza o valor de **x** utilizando a fórmula **x = x - g(x) / dg(x)**. O loop while continua até que o número máximo de iterações (**k_max**) seja alcançado ou até que a convergência seja alcançada. A convergência é verificada pelas condições **abs(g(x))** $<$ **eps_1** (o valor da função é suficientemente próximo de zero) e **abs(x - x_prev)** $<$ **eps_2** (o valor de x mudou pouco entre iterações).

```
""" Introdu  o dos parametros """

x0 = float(input("Digite o valor de x0: "))
k_max = int(input("Digite o n mero m ximo de itera  es: "))
eps_1 = float(input("Digite o valor de eps_1: "))
eps_2 = float(input("Digite o valor de eps_2: "))
```

```
7
8 result = newton(g, dg, x0, eps_1, eps_2, k_max)
9 print(f"A raiz da fun   o    : {result}")
```

Aqui pede-se ao utilizador para introduzir os valores necessários para o método de Newton: o valor inicial **x0**, o número máximo de iterações **k_max**, a tolerância **eps_1** para a função ser considerada próxima de zero, e a tolerância **eps_2** para verificar a convergência do valor de **x**. Em seguida, chamamos a função **newton(g, dg, x0, eps_1, eps_2, k_max)** com os valores fornecidos e apresentamos o resultado, ou seja, a raiz da função, na última linha de código.

2.2.1 Porque dá erro com $x_0 = 1.4$

Ao introduzir o valor 1.4 no programa, o mesmo só irá funcionar para a primeira "volta"do ciclo while, isto porque na segunda volta o valor de **x** já irá ser negativo e deste modo a função **g(x)** irá dar erro, pois não é possível calcular o **ln** de um **x** negativo.

2.3 Exercício 3

```
1 """ Fun   o e respetiva derivada  """
2
3 def g(x):
4     g = np.log(x) + (1 / x ** 2) - 1
5     return g
6
7 def dg(x):
8     dg = (1 / x) - (2 / x ** 3)
9     return dg
```

Nesta parte, definimos duas funções: **g(x)** e **dg(x)**. A função **g(x)** representa uma função matemática que é a soma do logaritmo natural de **x**, $1/x^2$ e **-1**. A função **dg(x)** representa a derivada da função **g(x)**. Ambas as funções foram definidas usando o **numpy** para permitir operações matemáticas eficientes.

```
1 """ M todos para determinar a raiz da fun   o """
2
3 # M todo da bissec   o
4 def bisec(a, b, k_max, eps_1, eps_2):
5     k = 1
6     c = (1/2) * (a + b)
7
8     while k < 5:
9         k += 1
10        c = (1/2) * (a + b)
11
12        if abs(g(c)) < eps_1 and b - a < eps_2:
13            print(f"Converge em {k} itera   es.")
```

```
            return c

        if g(a) * g(c) < 0:
            b = c
        else:
            a = c

    return c
```

Nesta parte, implementamos o método da bissecção para encontrar a raiz da função **g(x)**. O método da bissecção é uma técnica de procura binária que divide o intervalo [a, b] por metade em cada iteração até encontrar uma raiz aproximada. Neste caso, o método está configurado para realizar no máximo **5** iterações.

```
# M todo de Newton
def newton(g, dg, c, eps_1, eps_2, k_max):
    x = c
    k = 5
    while k < k_max:
        x_prev = x
        x = x - g(x) / dg(x)
        if abs(g(x)) < eps_1 and abs(x - x_prev) < eps_2:
            print(f"Converge em {k} itera  es.")
            return x
        k += 1
    print(f"Falhou em convergir em {k_max} itera  es.")
    return x
```

Nesta parte, implementamos o método de Newton para encontrar a raiz da função **g(x)**. O método de Newton é uma técnica iterativa que utiliza a derivada da função para encontrar a raiz. Neste caso, o método está configurado para realizar no máximo **k_max** iterações.

```
""" Introdu  o dos parametros """

a = float(input("Digite o valor de a: "))
b = float(input("Digite o valor de b: "))
k_max = int(input("Digite o n mero m ximo de itera  es: "))
eps_1 = float(input("Digite o valor de eps_1: "))
eps_2 = float(input("Digite o valor de eps_2: "))

result = newton(g, dg, bisec(a, b, k_max, eps_1, eps_2), eps_1, eps_2, k_max)
print(f"A raiz da fun  o  : {result}")
```

Nesta parte, pedimos ao utilizador para introduzir os valores necessários para os métodos: o valor de **a**, o valor de **b**, o número máximo de iterações **k_max**, a tolerância **eps_1** para a função ser considerada próxima de zero e a tolerância **eps_2** para verificar a convergência do valor de **x**. Em seguida, chamamos a função **bisec(a, b, k_max, eps_1, eps_2)** para encontrar

uma aproximação inicial da raiz utilizando o método da bissecção e, em seguida, chamamos a função **newton(g, dg, c, eps_1, eps_2, k_max)** para refiná-la utilizando o método de Newton. Finalmente, apresentamos o resultado, ou seja, a raiz da função, na última linha de código. Note que o método de Newton começa a partir da aproximação inicial encontrada pelo método da bissecção.

Ficha 3

3.1 Exercicio 1

```
from math import sqrt
```

Esta linha importa a função **sqrt** (raiz quadrada) da biblioteca **math**. Isto é necessário para calcular a raiz quadrada mais tarde no código.

```
def f(x,y):
    return x-y
```

Esta linha define uma função chamada **f(x, y)** que calcula a diferença entre **x** e **y** e retorna o resultado.

```
def dfx(x):
    return 1
```

Esta função **dfx(x)** retorna o valor 1. É uma função auxiliar usada para calcular a derivada parcial de **f** em relação a **x**.

```
def dfy(y):
    return -1
```

Esta função **dfy(y)** retorna o valor -1. É uma função auxiliar usada para calcular a derivada parcial de **f** em relação a **y**.

```
def gauss(x,y,dx,dy):
    erro = sqrt((dfx(x)*dx)**2+(dfy(y)*dy)**2)
    print(f"x-y = {f(x,y):.4f}     +-      {erro:.4f}")
    if abs(f(x,y)) > erro:
        return "Os numeros sao diferentes"
    else:
        return "Os numeros sao iguais"
```

A função **gauss(x, y, dx, dy)** calcula o erro usando o método de gauss. Ela recebe quatro parâmetros: **x**, **y**, **dx** e **dy**. O erro é calculado como a raiz quadrada da soma dos quadrados dos produtos das derivadas parciais de **f** com os erros **dx** e **dy**. Em seguida, imprime a diferença entre **x** e **y** juntamente com o erro calculado. Finalmente, compara o valor absoluto de **f(x, y)** com o erro e retorna uma mensagem indicando se os números são diferentes ou iguais.

```
x = float(input("Enter the value of x: "))
print("==========================")
y = float(input("Enter the value of y: "))
print("==========================")
dx = float(input("Enter the error in x: "))
print("==========================")
dy = float(input("Enter the error in y: "))
print("==========================")
print(gauss(x, y, dx, dy))
```

Essas linhas solicitam ao usuário que insira os valores de **x**, **y**, **dx** e **dy** e armazenam esses valores nas variáveis correspondentes. Em seguida, chama a função **gauss(x, y, dx, dy)** com os valores fornecidos e imprime o resultado retornado pela função.

3.2 Exercicio 2

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

Essas linhas importam as bibliotecas necessárias para o código: **matplotlib.pyplot** para os gráficos, **numpy** para operações numéricas e **pandas** para manipulação dos dados nas tabelas.

```
url = 'https://trixi.coimbra.lip.pt/data/fc/fc03/data-2019244037.xlsx'
df = pd.read_excel(url)
```

Essas linhas acessam um arquivo Excel a partir de uma URL fornecida e leem o conteúdo do arquivo em um objeto **DataFrame** chamado **df** usando a função **read_excel** do pandas.

```
g = 9.81  # m/s^2
df['Expected Period'] = 2 * np.pi * np.sqrt(df['Length'] / g)
```

Essas linhas calculam o período esperado utilizando a fórmula **T = 2*pi*sqrt(L/g)**, onde **L** é o comprimento e **g** é a aceleração gravitacional. O resultado é armazenado em uma nova coluna chamada **'Expected Period'** no **'DataFrame' 'df'**.

```
df['Error in Expected Period'] = np.sqrt((np.pi**2/g*np.sqrt(df['Length']))*df['
    dLength']**2)
```

Essa linha calcula o erro no período esperado utilizando uma fórmula específica e armazena o resultado em uma nova coluna chamada **'Error in Expected Period'** no **'DataFrame' 'df'**.

```
def f(x, y):
    return x - y

def dfx(x):
    return 1

def dfy(y):
    return -1
```

Essas linhas definem três funções: **f(x, y)**, **dfx(x)**, **dfy(y)**. Essas funções serão usadas posteriormente no cálculo do erro em relação ao período observado e esperado.

```
def gauss(x, y, dx, dy):
    erro = np.sqrt((dfx(x) * dx) ** 2 + (dfy(y) * dy) ** 2)
    if abs(f(x, y)) > erro:
        return "no"
    else:
        return "yes"
```

Essa função **gauss(x, y, dx, dy)** calcula o erro utilizando as funções **dfx(x)**, **dfy(y)**, **f(x, y)** definidas anteriormente. Se o valor absoluto de **f(x, y)** for maior que o erro, a função retorna "**no**", caso contrário, retorna "**yes**".

```
df['Equal?'] = df.apply(lambda row: gauss(row['Period'], row['Expected Period'],
    row['dPeriod'], row['Error in Expected Period']), axis=1)
```

Essa linha aplica a função **gauss** a cada linha do **'DataFrame' 'df'** usando a função **apply** do **pandas**. O resultado é armazenado em uma nova coluna chamada **'Equal?'** no **DataFrame**, indicando se o período observado é igual ao período esperado dentro do erro.

```
dx = df['dPeriod']
dy = df['Error in Expected Period']
x = df['Period']
y = df['Expected Period']
```

Essas linhas atribuem os valores das colunas relevantes do **'DataFrame' 'df'** a variáveis para serem usadas posteriormente no gráfico.

```
num_yes = df['Equal?'].value_counts()['yes']
print(f"Numero de potos pontos com per odos iguais: {num_yes}")
```

Essas linhas contam o número de ocorrências do valor **'yes'** na coluna **'Equal?'** do **'DataFrame' 'df'** e armazena na variável **num_yes**. O resultado é impresso no ecrã.

```
plt.errorbar(df['Length'], df['Period'], yerr=df['dPeriod'], marker='o',
    linestyle='none')
plt.plot(df['Length'], df['Period'], 'o', label='Periodo observado')
plt.plot(df['Length'], df['Expected Period'], '-', label='Periodo Esperado')
plt.xlabel('Comprimento (m)')
plt.ylabel('Periodo (s)')
plt.title('Periodo em fun   o do comprimento')
plt.legend()
plt.show()
```

Essas linhas criam um gráfico utilizando as colunas **'Length'**, **'Period'** e **'dPeriod'** do **'DataFrame' 'df'**. São trçados pontos com barras de erro, o período observado e o período esperado em função do comprimento. São adicionados rótulos aos eixos e um título ao gráfico. Em seguida, o gráfico é exibido no ecrã.

```
filename_excel = "updated_data.xlsx"
filename_json = "updated_data.json"
df.to_excel(filename_excel, index=False)
df.to_json(filename_json, orient='records')
```

Essas linhas definem os nomes de arquivo para o arquivo *Excel* e o arquivo *JSON* que serão criados. Em seguida, o **'DataFrame' 'df'** é salvo em formato *Excel* e *JSON* nos arquivos correspondentes.

Ficha 4

4.1 Exercício 1

```
def f(t):
    return (10*np.exp(-t)*np.sin(2*np.pi*t))**2
```

Nesta parte, defini-se uma função chamada **f(t)**. A função é uma expressão matemática que envolve funções trigonométricas e exponenciais.

```
def rectangulo(f, a, b, n):
    h = (b - a) / n
    integral = 0.0
    for i in range(n):
        xi = a + h/2 + i*h
        integral += f(xi)
    return h * integral
```

- **rectangulo(f, a, b, n)**: Esta função recebe como argumentos a função **f(x)** a ser integrada, os limites de integração inferior (**a**) e superior (**b**), e o número de subdivisões (**n**) para calcular o integral.
- **h = (b - a) / n**: É calculado o tamanho do intervalo (**h**) entre as subdivisões, que será usado para o cálculo da integral.
- **integral = 0.0**: É inicializada a variável integral com o valor zero, onde armazenaremos a soma das contribuições de cada subdivisão para o cálculo da integral.
- **for i in range(n):**: Inicia-se um loop que percorrerá as subdivisões, de **0** até **n-1**.
- **xi = a + h/2 + i*h**: Calcula-se o ponto médio (**xi**) de cada subdivisão, que será usado como o ponto de avaliação da função **f(x)** para o método do retângulo.
- **integral += f(xi)**: Adiciona-se à variável integral o valor da função **f(xi)** avaliada no ponto médio da subdivisão, representando a contribuição daquela subdivisão para o cálculo da integral.

- **return h * integral**: Retorna o resultado final da integral, que é a soma das contribuições de todas as subdivisões multiplicada pelo tamanho do intervalo (**h**).

```
def trapezio(f, a, b, n):
    h = (b - a) / n
    integral2 = f(a) + f(b)
    for i in range(1, n):
        xi2 = a + i * h
        integral2 += 2 * f(xi2)
    integral2 *= h / 2
    return integral2
```

- **trapezio(f, a, b, n)**: Esta função recebe os mesmos argumentos da função anterior e também calcula a integral utilizando o método do trapézio.
- **h = (b - a) / n**: É calculado o tamanho do intervalo (**h**) entre as subdivisões.
- **integral2 = f(a) + f(b)**: É inicializada a variável **integral2** com a soma dos valores da função **f(x)** nos limites **a** e **b**, que são as contribuições dos trapézios formados pelos extremos da integral.
- **for i in range(1, n):**: Inicia-se um loop que percorrerá as subdivisões, de **1** até **n-1**.
- **xi2 = a + i * h**: Calcula-se o valor de **xi2**, que é o ponto de avaliação da função **f(x)** no início de cada subdivisão, onde será utilizada no cálculo dos trapézios.
- **integral2 += 2 * f(xi2)**: Adiciona-se à variável **integral2** o valor da função **f(xi2)** multiplicado por 2, representando a contribuição de cada trapézio para o cálculo da integral.
- **integral2 *= h / 2**: Multiplica-se a soma das contribuições de todos os trapézios por **h/2**, que é o fator comum no método do trapézio.
- **return integral2**: Retorna o resultado final da integral, que é a soma das contribuições de todos os trapézios multiplicada pelo fator **h/2**.

```
def simpson(f, a, b, n):
    h = (b - a) / n
    integral3 = f(a) + f(b)
    for i in range(1, n):
        xi = a + i * h
        if i % 2 == 0:
            integral3 += 2 * f(xi)
        else:
            integral3 += 4 * f(xi)
    integral3 *= h / 3
    return integral3
```

- **simpson(f, a, b, n)**: Esta função também recebe os mesmos argumentos das funções anteriores e calcula a integral utilizando o método de Simpson.
- **h = (b - a) / n**: É calculado o tamanho do intervalo (**h**) entre as subdivisões.
- **integral3 = f(a) + f(b)**: É inicializada a variável **integral3** com a soma dos valores da função **f(x)** nos limites **a** e **b**, que são as contribuições dos retângulos externos do método de Simpson.
- **for i in range(1, n):**: Inicia-se um loop que percorrerá as subdivisões, de **1** até **n-1**.
- **xi = a + i * h**: Calcula-se o valor de **xi**, que é o ponto de avaliação da função **f(x)** no início de cada subdivisão, onde será utilizado no cálculo dos retângulos internos do método de Simpson.
- **if i % 2 == 0:**: Verifica se **i** é um número par (se **i** é divisível por **2**).
- **integral3 += 2 * f(xi)**: Se **i** é par, adiciona-se à variável **integral3** o valor da função **f(xi)** multiplicado por **2**, representando a contribuição dos retângulos internos pares para o cálculo da integral.
- **else:**: Caso contrário, ou seja, se **i** for ímpar.
- **integral3 += 4 * f(xi)**: Adiciona-se à variável **integral3** o valor da função **f(xi)** multiplicado por **4**, representando a contribuição dos retângulos internos ímpares para o cálculo da integral.
- **integral3 *= h / 3**: Multiplica-se a soma das contribuições de todos os retângulos internos por **h/3**, que é o fator comum no método de Simpson.
- **return integral3**: Retorna o resultado final da integral, que é a soma das contribuições de todos os retângulos internos multiplicada pelo fator **h/3**.

```
if method == '1':
    n = int(input("Digite o n mero de itera  es: "))
    result = rectangulo(f, a, b, n)
    n_rect = 1
    erro = 1 + erro_desejado
    while erro > erro_desejado:
        n_rect += 1
        erro = M * (b - a) ** 3 / (24 * n_rect ** 2)
    print("Integral =", result)
    print(f"Para a regra do ret ngulo , n = {n_rect} para se obter o erro de {
    erro_desejado}")
    show_graph = input("Deseja visualizar o gr fico? (s/n): ").lower()
    while show_graph not in ['s', 'n']:
        show_graph = input("Deseja visualizar o gr fico? (s/n): ").lower()
    if show_graph.lower() == "s":
```

```
15          x = np.linspace(a, b)
16          y = f(x)
17          plt.plot(x, y, label='f(t)')
18          dx = (b - a) / n
19          for i in range(n):
20              xi = a + i * dx + dx / 2
21              plt.fill_between([xi - dx / 2, xi + dx / 2], [0, 0], [f(xi), f(xi)],
    color='r', alpha=0.3)
22          plt.title('Regra do Ret ngulo')
23          plt.legend()
24          plt.show()
25      else:
26          print("Se desejar ver o gr fico , volte a correr o programa.")
```

Nesta parte, se o utilizador escolher o método do retângulo (**method == '1'**), pedimos o número de iterações (**n**) e calculamos o integral utilizando a função **rectangulo**. Além disso, calculamos o número mínimo de iterações (**n_rect**) para alcançar o erro desejado usando uma fórmula específica. Em seguida, o programa verifica se o utilizador deseja visualizar o gráfico do método do retângulo.

```
1  elif method == '2':
2      n = int(input("Digite o n mero de itera  es: "))
3      n_trap = 1
4      erro = 1 + erro_desejado
5      while erro > erro_desejado:
6          n_trap += 1
7          erro = M * (b - a) ** 3 / (12 * n_trap ** 2)
8      result2 = trapezio(f, a, b, n)
9      print("Integral =", result2)
10     print(f"Para a regra do trap zio , n = {n_trap} para se obter o erro de {
       erro_desejado}")
11     show_graph = input("Deseja visualizar o gr fico? (s/n): ").lower()
12     while show_graph not in ['s', 'n']:
13         show_graph = input("Deseja visualizar o gr fico? (s/n): ").lower()
14     if show_graph.lower() == "s":
15         x = np.linspace(a, b)
16         y = f(x)
17         plt.plot(x, y, label='f(t)')
18         dx = (b - a) / n
19         for i in range(n):
20             xi = a + i * dx
21             plt.fill_between([xi, xi + dx], [0, 0], [f(xi), f(xi + dx)], color='
       r', alpha=0.3)
22         plt.title('Regra do Trap zio')
23         plt.legend()
24         plt.show()
25     else:
26         print("Se desejar ver o gr fico , volte a correr o programa.")
```

Nesta parte, se o utilizador escolher o método do trapézio (**method == '2'**), pedimos o número de iterações (**n**) e calculamos o integral utilizando a função **trapezio**. Assim como antes, também calculamos o número mínimo de iterações (**n_trap**) para alcançar o erro desejado. Em seguida, o programa verifica se o utilizador deseja visualizar o gráfico do método do trapézio.

```
else:
    n = int(input("Digite o n mero de itera  es: "))
    n_sim = 1
    erro = 1 + erro_desejado
    while erro > erro_desejado:
        n_sim += 1
        erro = M * (b - a) ** 5 / (180 * n_sim ** 4)
    result3 = simpson(f, a, b, n)
    print("Integral =", result3)
    print(f"Para a regra de Simpson, n = {n_sim} para se obter o erro de {
    erro_desejado}")
    show_graph = input("Deseja visualizar o gr fico? (s/n): ").lower()
    while show_graph not in ['s', 'n']:
        show_graph = input("Deseja visualizar o gr fico? (s/n): ").lower()
    if show_graph.lower() == "s":
        x = np.linspace(a, b)
        y = f(x)
        plt.plot(x, y, label='f(t)')
        dx = (b - a) / n
        for i in range(n):
            xi = a + i * dx
            if i % 2 == 0:
                plt.fill_between([xi, xi + dx], [0, 0], [f(xi), f(xi + dx)],
    color='r', alpha=0.3)
            else:
                plt.fill_between([xi, xi + dx], [0, 0], [f(xi), f(xi + dx)],
    color='g', alpha=0.3)
        plt.title('Regra de Simpson')
        plt.legend()
        plt.show()
    else:
        print("Se desejar ver o gr fico, volte a correr o programa.")
```

Nesta parte, se o utilizador escolher o método de Simpson (**method == '3'**), pedimos o número de iterações (**n**) e calculamos o integral utilizando a função **simpson**. Mais uma vez, calculamos o número mínimo de iterações (**n_sim**) para alcançar o erro desejado. Em seguida, o programa verifica se o utilizador deseja visualizar o gráfico do método de Simpson.

4.2 Exercício 2

```
def f(t):
    return  (t/(t**2 + 1)*np.cos(10*t**2))
```

É definida a **função f(t)**, que representa a função a ser integrada. Essa é a principal diferença em relação ao primeiro código, pois agora estamos a usar uma função diferente para o cálculo da integral.

```
a = 0
b = np.pi
valor_exato = 0.0003156
M = 408.1987
erro_desejado = 0.01 * valor_exato
```

Aqui são definidos os parâmetros **a** e **b** para os limites de integração, o **valor_exato** do integral exato e o **M**, que é usado para calcular o erro máximo permitido (**erro_desejado**) de 1% do valor exato.

```
if method == '1':
    n = int(input("Digite o n mero de itera  es: "))
    result = rectangulo(f, a, b, n)
    n_rect = 1
    erro = 1 + erro_desejado
    while erro > erro_desejado:
        n_rect += 1
        erro = M * (b - a)**3 / (24 * n_rect**2)
    print("Integral =", result)
    print(f"Para a regra do retangulo, n = {n_rect} para se obter o erro de {
    erro_desejado}")
    # Codigo para tra ado do grafico, se desejado pelo usu rio.
```

Se o método escolhido for o retângulo (**código '1'**), o usuário deve informar o número de iterações (**n**). Em seguida, o programa calcula a integral usando o método do retângulo e determina o número necessário de iterações (**n_rect**) para alcançar o erro desejado.

O mesmo sucede para as outras funções, sendo a implementação do código de forma semelhante.

Ficha 5

5.1 Simulação Monte Carlo

Note-se que nesta ficha seria para simular o decaimento radioativo pela simulação de Monte Carlo, no entanto usei outros métodos, que embora mais simples não são o pedido pois não realiza simulação de Monte Carlo.

5.2 Exercício 1

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

As linhas acima importam as bibliotecas necessárias para o código: **numpy** para operações numéricas e **matplotlib.pyplot** para traçado de gráficos.

```
N1 = 100000
N2 = 0
N3 = 0

dt = 1

l1 = 0.1
l2 = 0.3
l3 = 0
```

Estas linhas definem valores para as variáveis **N1**, **N2** e **N3**, que representam o número inicial de núcleos de cada isótopo. A variável **dt** define o intervalo de tempo para cada iteração. Por último **l1**, **l2** e **l3** são as taxas de decaimento para cada tipo de núcleo.

```
t = np.arange(0, 100, dt)

N1_array = np.zeros(len(t))
N2_array = np.zeros(len(t))
```

```
5 N3_array = np.zeros(len(t))
```

Estas linhas criam um **array t** que representa o intervalo de tempo em que os cálculos serão realizados. Os arrays **N1_array**, **N2_array** e **N3_array** são inicializados com zeros e serão usados para armazenar os valores do número de núcleos de cada tipo em cada instante de tempo.

```
1 N1_array[0] = N1
2 N2_array[0] = N2
3 N3_array[0] = N3
```

Estas linhas definem os valores iniciais dos arrays **N1_array**, **N2_array** e **N3_array** com os valores iniciais de **N1**, **N2** e **N3**, respetivamente.

```
1  for i in range(1, len(t)):
2      decay1 = np.random.rand(N1_array[i-1].astype(int)) < l1*dt
3      decay2 = np.random.rand(N2_array[i-1].astype(int)) < l2*dt
4
5      N1 = N1 - np.sum(decay1)
6      N2 = N2 + np.sum(decay1) - np.sum(decay2)
7      N3 = N3 + np.sum(decay2)
8
9      N1_array[i] = N1
10     N2_array[i] = N2
11     N3_array[i] = N3
```

Estas linhas representam o loop principal do código. Para cada instante de tempo, começando do segundo (i = 1), o código realiza o decaimento de núcleos. As variáveis **decay1** e **decay2** são **boolean arrays** (Um boolean array é um tipo de array que contém valores boolean, ou seja, valores verdadeiros (True) ou falsos (False).) gerados aleatoriamente com base nas taxas de decaimento **l1** e **l2**, respetivamente. O número de núcleos de cada tipo é atualizado de acordo com o decaimento ocorrido. Os valores atualizados são armazenados nos arrays **N1_array**, **N2_array** e **N3_array**.

Dentro do loop, as seguintes etapas são executadas:

1. **decay1 = np.random.rand(N1_array[i-1].astype(int)) ¡ l1*dt**: É gerado um boolean array chamado **decay1**. O tamanho desse **array** é determinado pelo valor armazenado em **N1_array[i-1]**, que representa o número de núcleos do tipo 1 no momento anterior. A função **np.random.rand()** gera um **array** com números aleatórios entre 0 e 1 do mesmo tamanho. A comparação **¡ l1*dt** é aplicada a cada elemento do **array** gerado, resultando em **True** para os elementos menores que **l1*dt** e **False** caso contrário. Isso determina quais núcleos do tipo 1 irão decair no intervalo de tempo atual.

2. **decay2 = np.random.rand(N2_array[i-1].astype(int)) ¡ l2*dt**: É gerado um boolean array chamado **decay2**. O tamanho desse **array** é determinado pelo valor armazenado em **N2_array[i-1]**, que representa o número de núcleos do tipo 2 no momento anterior. Da mesma forma que em **decay1**, os elementos de **decay2** são definidos com base na comparação entre os valores gerados aleatoriamente e **l2*dt**. Isso determina quais núcleos do tipo 2 irão decair no intervalo de tempo atual.

3. **N1 = N1 - np.sum(decay1)**: O número de núcleos do tipo 1 é atualizado subtraindo a soma dos elementos **True** em **decay1** do valor atual de **N1**. Isso representa a diminuição do número de núcleos do tipo 1 devido ao decaimento.

4. **N2 = N2 + np.sum(decay1) - np.sum(decay2)**: O número de núcleos do tipo 2 é atualizado somando a soma dos elementos **True** em **decay1** e subtraindo a soma dos elementos **True** em **decay2** do valor atual de **N2**. Isso considera tanto o aumento no número de núcleos do tipo 2 devido ao decaimento do tipo 1 quanto a diminuição devido ao decaimento do próprio tipo 2.

5. **N3 = N3 + np.sum(decay2)**: O número de núcleos do tipo 3 é atualizado somando a soma dos elementos **True** em **decay2** ao valor atual de **N3**. Isso representa o aumento no número de núcleos do tipo 3 devido ao decaimento do tipo 2.

6. **N1_array[i] = N1**, **N2_array[i] = N2** e **N3_array[i] = N3**: Os valores atualizados de **N1**, **N2** e **N3** são armazenados nos **arrays N1_array**, **N2_array** e **N3_array**, respetivamente, para o tempo atual **t[i]**.

Estas etapas são repetidas para cada intervalo de tempo no loop, atualizando os números de núcleos e armazenando-os nos **arrays** correspondentes.

No final desse bocado de código, teremos os **arrays N1_array**, **N2_array** e **N3_array** contendo os números de núcleos dos tipos 1, 2 e 3, respetivamente, para cada intervalo de tempo **t**.

```
plt.plot(t, N1_array, label='N1 λ1 = 0.1/s')
plt.plot(t, N2_array, label='N2 λ2 = 0.3/s')
plt.plot(t, N3_array, label='N3 λ3 = 0')
plt.xlabel('t [s]')
plt.ylabel('n (t)')
plt.title('Decaimento Δt=1')
plt.legend()
plt.grid(True)
plt.show()
```

Estas linhas criam um gráfico para exibir os resultados. Os **arrays t**, **N1_array**, **N2_array** e **N3_array** são usados para traçar as curvas de decaimento de cada tipo de núcleo ao longo do tempo. Os rótulos dos eixos e o título do gráfico são definidos. A função **legend()** exibe uma

legenda com as informações dos diferentes tipos de núcleos. Por fim, **grid(True)** adiciona uma grelha ao gráfico e **show()** exibe o gráfico no ecrã.

```
plt.plot(t[:21], N1_array[:21], label='N1 λ1 = 0.1/s')
plt.plot(t[:21], N2_array[:21], label='N2 λ2 = 0.3/s')
plt.plot(t[:21], N3_array[:21], label='N3 λ3 = 0')
plt.xlabel('t [s]')
plt.ylabel('n (t)')
plt.title('Decaimento Δt=1 para 0 ≤ t ≤ 20s')
plt.legend()
plt.grid(True)
plt.show()
```

Estas linhas criam um segundo gráfico que mostra apenas os primeiros 21 pontos no tempo. Isso permite uma visualização mais detalhada do decaimento inicial. O restante do código é semelhante ao anterior, com a diferença de que o intervalo de tempo é limitado a 20 segundos.

Uma fonte de erro sistemático na simulação de Monte Carlo do decaimento nuclear é o uso de um intervalo de tempo fixo Δt. Isso pode levar a imprecisões na simulação, especialmente se Δt não for suficientemente pequeno em comparação com as constantes de decaimento. Se Δt for muito grande, a simulação pode não capturar com precisão o comportamento do sistema, resultando em erros nos resultados.

Uma maneira de reduzir esse erro é usar um valor menor de Δt. Isso aumentará a precisão da simulação, permitindo que ela acompanhe de perto o comportamento do sistema. No entanto, usar um valor menor de Δt também aumentará o custo computacional da simulação, pois serão necessárias mais etapas de tempo para cobrir o mesmo intervalo de tempo.

Outra abordagem para reduzir esse erro é usar um método de passo de tempo adaptativo, em que o tamanho de Δt é ajustado dinamicamente com base no comportamento do sistema. Isso permite que a simulação use passos de tempo maiores quando o sistema muda lentamente e passos de tempo menores quando muda rapidamente. Isso pode melhorar a precisão da simulação e reduzir o custo computacional.

5.3 Exercício 2

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

N1 = 100000
N2 = 0
N3 = 0

dt = 0.1

l1 = 0.1
l2 = 0.3
```

```
l3 = 0

t = np.arange(0, 20 + dt, dt)

N1_array = np.zeros(len(t))
N2_array = np.zeros(len(t))
N3_array = np.zeros(len(t))


N1_array[0] = N1
N2_array[0] = N2
N3_array[0] = N3

for i in range(1, len(t)):
    # Determina pelo m todo de Monte Carlo para cada nucleo N1 e N2 se estes
    v o decair.
    decay1 = np.random.rand(N1_array[i-1].astype(int)) < l1*dt
    decay2 = np.random.rand(N2_array[i-1].astype(int)) < l2*dt

    # Determina o n mero de nucleos N1, N2, N3 depois de cada intrevalo Δt.
    N1 = N1 - np.sum(decay1)
    N2 = N2 + np.sum(decay1) - np.sum(decay2)
    N3 = N3 + np.sum(decay2)

    # Armazena o n mero de n cleos N1, N2 e N3 numa matriz.
    N1_array[i] = N1
    N2_array[i] = N2
    N3_array[i] = N3

# Graph the number of nuclei per time for the three nuclei in the same diagram
    for 0 ≤ t ≤ 100s and 0 ≤ t ≤ 20s.
plt.plot(t, N1_array, label='N1 λ1 = 0.1/s')
plt.plot(t, N2_array, label='N2 λ2 = 0.3/s')
plt.plot(t, N3_array, label='N3 λ3 = 0')
plt.xlabel('t [s]')
plt.ylabel('n (t)')
plt.title('Decaimento Δt=0.1')
plt.legend()
plt.grid(True)
plt.show()
```

As constantes de decaimento para **N1** e **N2** não foram alteradas. Em vez disso, a probabilidade de decaimento para cada núcleo em cada intervalo de tempo foi calculada multiplicando a constante de decaimento pelo intervalo de tempo. Isso é equivalente a aplicar um fator de correção à constante de decaimento igual ao intervalo de tempo.

5.3.1 Explain which correction you applied to the decay constant

A razão para aplicar essa correção é que a probabilidade de decaimento para cada núcleo em um determinado intervalo de tempo é dada pelo produto da constante de decaimento pelo intervalo de tempo. Como o intervalo de tempo no código modificado foi reduzido de 1 segundo para $0,1$ segundos, a probabilidade de decaimento para cada núcleo em cada intervalo de tempo é reduzida por um fator de 10. Multiplicar a constante de decaimento pelo intervalo de tempo ao calcular a probabilidade de decaimento garante que a probabilidade de decaimento seja corretamente dimensionada para o menor intervalo de tempo.

Note-se que estas *"correções"* já foram realizadas no código do **exercício 1**.

5.4 Exercício 3

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

Estas linhas importam as bibliotecas **numpy** e **matplotlib.pyplot**, que serão utilizadas no código.

```
l1 = 0.9  # constante de decaimento para N1
l2 = 0.3  # constante de decaimento para N2
l3 = 0.0  # constante de decaimento para N3
```

Estas linhas definem as constantes de decaimento para cada tipo de núcleo. Neste caso, apenas **N1** fo alterado em relação aos códigos anteriores.

```
t_max = 3.0  # tempo m ximo para a simula  o
dt_values = [1.0, 0.1, 0.01]  # intervalos de tempo para a simula  o
```

Estas linhas definem o tempo máximo da simulação (3.0 segundos) e os intervalos de tempo (1.0 segundo, 0.1 segundos e 0.01 segundos) que serão utilizados.

```
fig, axs = plt.subplots(3, 1, figsize=(8, 12))  # cria  o dos subplots
```

Esta linha cria uma figura com três **subplots** dispostos verticalmente. Cada **subplot** será utilizado para traçar os resultados da simulação com um intervalo de tempo diferente.

```
for i, dt in enumerate(dt_values):
    t = np.arange(0, t_max+dt, dt)  # array de tempo para a simulacao

    N1 = np.zeros(len(t))
    N2 = np.zeros(len(t))
    N3 = np.zeros(len(t))

```

```
    N1[0] = N1_0
    N2[0] = N2_0
    N3[0] = N3_0
```

Este bocado do código é um loop que irá executar a simulação para cada intervalo de tempo definido em **dt_values**. Para cada iteração, ele cria um **array** de tempo **t** que vai de 0 até **t_max** com um passo **dt**. Também são criados **arrays N1**, **N2** e **N3**, preenchidos com zeros, que serão usados para armazenar a quantidade de núcleos de cada tipo ao longo do tempo. As quantidades iniciais definidas anteriormente são atribuídas aos primeiros elementos desses **arrays**.

```
    for j in range(1, len(t)):
        decay1 = np.random.rand(N1[j-1].astype(int)) < l1*dt
        decay2 = np.random.rand(N2[j-1].astype(int)) < l2*dt

        N1[j] = N1[j-1] - np.sum(decay1)
        N2[j] = N2[j-1] + np.sum(decay1) - np.sum(decay2)
        N3[j] = N3[j-1] + np.sum(decay2)
```

Este é o loop interno que realiza a simulação do decaimento radioativo para cada instante de tempo. Para cada valor de **j**, ele gera números aleatórios entre 0 e 1 para **N1[j-1]** e **N2[j-1]**. A partir desses números, ele determina quais núcleos de cada tipo decaem de acordo com as constantes de decaimento **l1** e **l2** multiplicadas pelo intervalo de tempo **dt**. Os núcleos que decaem são subtraídos de **N1[j-1]** e **N2[j-1]** e adicionados a **N2[j-1]** e **N3[j-1]**, respetivamente.

```
    axs[i].plot(t, N1, label='N1 λ1 = 0.9/s')
    axs[i].plot(t, N2, label='N2 λ2 = 0.3/s')
    axs[i].plot(t, N3, label='N3 λ3 = 0')
    axs[i].set_xlabel('t [s]')
    axs[i].set_ylabel('n (t)')
    axs[i].set_title(f'Decay Δt={dt}s')
    axs[i].legend()
    axs[i].grid(True)
```

A primeira linha cria um gráfico no **subplot** atual (**axs[i]**) com o tempo **t** no eixo **x** e a quantidade de núcleos **N1** no eixo **y**. A legenda do gráfico é definida como **"N1 1 = 0.9/s"**, onde **"N1"** representa o tipo de núcleo, e **"1 = 0.9/s"** indica a constante de decaimento associada ao tipo de núcleo **N1**.

A segunda linha cria outro gráfico no mesmo **subplot** atual, mas agora com a quantidade de núcleos **N2**. A legenda do gráfico é definida como **"N2 2 = 0.3/s"**, onde **"N2"** representa o tipo de núcleo e **"2 = 0.3/s"** indica a constante de decaimento associada ao tipo de núcleo **N2**.

Da mesma forma que as linhas anteriores, a terceira linha cria um gráfico no **subplot** atual com a quantidade de núcleos **N3**. A legenda do gráfico é definida como **"N3 3 = 0"**, onde **"N3"**

representa o tipo de núcleo e **"3 = 0"** indica que o tipo de núcleo N3 não sofre decaimento, pois sua constante de decaimento é zero.

A quarta e quinta linha definem os rótulos dos eixos do gráfico. O eixo **x** é rotulado como **"t [s]"**, indicando o tempo em segundos, e o eixo **y** é rotulado como **"n (t)"**, indicando a quantidade de núcleos em função do tempo.

A sexta linha define o título do **subplot**. O título é formatado com o valor do intervalo de tempo **dt** utilizado na simulação, indicando a taxa de decaimento.

A sétima linha exibe a legenda no gráfico, mostrando as informações sobre os diferentes tipos de núcleos e suas constantes de decaimento.

A oitava linha adiciona uma grelha ao gráfico, facilitando a visualização dos pontos no plano.

5.5 Exercício 4

```
lambda_N1_N2 = 0.5   # Constante de decaimento para N1 → N2
lambda_N2_N4 = 0.1   # Constante de decaimento para N2 → N4
lambda_N1_N3 = 0.4   # Constante de decaimento para N1 → N3
lambda_N3_N4 = 0.1   # Constante de decaimento para N3 → N4
```

Estas linhas definem as constantes de decaimento para os diferentes processos nucleares. Por exemplo, **lambda_N1_N2** é a constante de decaimento para o processo em que os núcleos do tipo N1 se transformam em N2.

```
N1_0 = 100000   # N mero inicial de n cleos N1
N2_0 = 0       # N mero inicial de n cleos N2
N3_0 = 0       # N mero inicial de n cleos N3
N4_0 = 0       # N mero inicial de n cleos N4
```

Estas linhas definem o número inicial de núcleos para cada tipo de núcleo.

```
dt = 0.01          # Passo de tempo (em segundos)
t = np.arange(0, 50, dt)    # Array de tempo
```

Estas linhas definem o intervalo de tempo (**dt**) utilizado na simulação e criam um **array** de tempo **t** que varia de 0 a 50 segundos com intervalos de **dt**.

```
P_N1_N2 = 1 - np.exp(-lambda_N1_N2*dt)    # Probabilidade de decaimento N1 → N2
P_N2_N4 = 1 - np.exp(-lambda_N2_N4*dt)    # Probabilidade de decaimento N2 → N4
P_N1_N3 = 1 - np.exp(-lambda_N1_N3*dt)    # Probabilidade de decaimento N1 → N3
P_N3_N4 = 1 - np.exp(-lambda_N3_N4*dt)    # Probabilidade de decaimento N3 → N4
```

Estas linhas calculam as probabilidades de decaimento para cada processo nuclear, com base nas constantes de decaimento e no passo de tempo. A fórmula utilizada é **1 - exp(-lambda*dt)**, onde lambda é a constante de decaimento e **dt** é o intervalo de tempo.

```
N1 = np.zeros_like(t)
N2 = np.zeros_like(t)
N3 = np.zeros_like(t)
N4 = np.zeros_like(t)
```

Estas linhas inicializam **arrays** (vetores) **N1**, **N2**, **N3** e **N4** com zeros, com a mesma forma (tamanho) do **array** de tempo **t**.

```
N1[0] = N1_0
N2[0] = N2_0
N3[0] = N3_0
N4[0] = N4_0
```

Estas linhas definem os valores iniciais dos **arrays N1**, **N2**, **N3** e **N4** como os números iniciais de núcleos.

```
for i in range(len(t)-1):
    # N1 → N2
    N1[i+1] = N1[i] - np.random.binomial(N1[i], P_N1_N2)
    N2[i+1] = N2[i] + np.random.binomial(N1[i], P_N1_N2) - np.random.binomial(N2
    [i], P_N2_N4)
    # N1 → N3
    N1[i+1] = N1[i+1] - np.random.binomial(N1[i+1], P_N1_N3)
    N3[i+1] = N3[i] + np.random.binomial(N1[i+1], P_N1_N3) - np.random.binomial(
    N3[i], P_N3_N4)
    # N2 → N4 e N3 → N4
    N4[i+1] = N4[i] + np.random.binomial(N2[i], P_N2_N4) + np.random.binomial(N3
    [i], P_N3_N4)
```

Esta parte do código realiza a simulação do processo de decaimento nuclear ao longo do tempo. O loop **for** itera sobre os índices de tempo, exceto o último. Dentro do loop, são realizados os cálculos para atualizar os números de núcleos em cada intervalo de tempo, com base nas probabilidades de decaimento.

```
plt.grid(True)
plt.plot(t, N1, label='N1')
plt.plot(t, N2, label='N2')
plt.plot(t, N3, label='N3')
plt.plot(t, N4, label='N4')
plt.legend()
plt.xlabel('t [s]')
plt.ylabel('n (t)')
plt.title('Simulacao de Cadeia de Decaimento')
plt.show()
```

Estas linhas criam um gráfico usando a biblioteca **matplotlib.pyplot** para exibir os resultados da simulação. Os **arrays** de tempo **t** e os números de núcleos **N1**, **N2**, **N3** e **N4** são

traçados no gráfico. Os rótulos dos eixos **x** e **y**, a legenda e o título do gráfico também são definidos. Finalmente, o gráfico é exibido usando **plt.show()**.

Ficha 6

6.1 Exercício 1

```
1  def monte_carlo(f, a, b, n):
2      subsets = np.arange(0, n + 1, n / 10)
3      u = np.zeros(n)
4      for i in range(10):
5          start = int(subsets[i])
6          end = int(subsets[i + 1])
7          u[start:end] = np.random.uniform(low=i / 10, high=(i + 1) / 10, size=end
       - start)
8      np.random.shuffle(u)
9      u_func = f(a + (b - a) * u)
10     s = ((b - a) / n) * u_func.sum()
11     return s
```

É definida a função **monte_carlo** para calcular o integral de uma função usando o método de Monte Carlo.

- Da linha 2 até à linha 8:
  - Neste secção do código, a amostragem aleatória para o método de Monte Carlo é realizada: **subsets** é um vetor com **11** valores igualmente espaçados entre **0** e **n**, representando os pontos iniciais e finais de **10** subconjuntos do tamanho **n/10**.
  - **u** é inicializado como um vetor de **zeros** com tamanho **n**.
  - Num loop de 10 iterações, é criado um vetor de tamanho **n/10** com números aleatórios uniforme-mente distribuídos no intervalo [0, 0.1], [0.1, 0.2], ..., [0.9, 1.0] e inseridos nos locais corretos em **u**.
  - Após a inserção, os elementos de **u** são baralhados aleatoriamente.
- Na linha 9, **u** é usado para calcular os valores da função **f** no intervalo [a, b] e armazenado em **u_func**.

- Na linha 10, o integral é calculada usando a fórmula básica do método de Monte Carlo, onde o integral é aproximadamente igual à média da função **f** multiplicado pela largura do intervalo **(b - a) / n**.
- Na linha 11, o resultado aproximado do integral é retornado.

```
1 def f(t):
2     return (10 * np.exp(-t) * np.sin(2 * np.pi * t)) ** 2
```

A função **f(t)** é definida. Esta é a função que será integrada usando o método de Monte Carlo. É semelhante à função do primeiro código, mas agora para uma nova função.

```
1  def simpson(f, a, b, n):
2      h = (b - a) / n
3      integral3 = f(a) + f(b)
4      for i in range(1, n):
5          xi = a + i * h
6          if i % 2 == 0:
7              integral3 += 2 * f(xi)
8          else:
9              integral3 += 4 * f(xi)
10     integral3 *= h / 3
11     return integral3
```

A função simpson é definida novamente, semelhante à função do código da *ficha 4*, mas agora para a nova função **f(t)**.

```
1 a = 0
2 b = 0.5
3 valor_exato = 15.41260804810169
4
5 n = int(input("Digite o n mero de itera  es: "))
```

Nesta parte, são definidos os parâmetros **a** e **b** para os limites de integração, e também é definido o valor exato do integral **valor_exato**. O usuário é solicitado a inserir o número de iterações **n** para o método de Monte Carlo.

```
1 start_time = time.time()
2 result3 = simpson(f, a, b, n)
3 simpson_time = time.time() - start_time
4 print(f"Integral pelo m todo de Simpson: {result3:.5f}, tempo: {simpson_time:.5
      f}s")
```

O método de Simpson é executado com o número de iterações **n** definido pelo usuário, e o tempo de execução é medido usando a biblioteca **time**. O resultado aproximado do integral e o tempo de execução são exibidos na saída.

```
start_time = time.time()
result = monte_carlo(f, a, b, n)
monte_carlo_time = time.time() - start_time
print(f"Integral pelo m todo de Monte Carlo: {result:.5f}, tempo: {
    monte_carlo_time:.5f}s")
```

O método de Monte Carlo é executado com o número de iterações **n** definido pelo usuário, e o tempo de execução é medido novamente usando a biblioteca **time**. O resultado aproximado do integral e o tempo de execução são exibidos na saída.

6.1.1 Método de Monte Carlo melhor

```
def monte_carlo(f, a, b, n):
    # Gera n pontos aleat rios distribu dos uniformemente entre a e b
    u = np.random.uniform(low=a, high=b, size=n)

    # Calcula os valores da fun   o f para cada ponto aleat rio gerado
    u_func = f(u)

    # Calcula a aproxima   o da integral multiplicando a m dia dos valores
    pela largura do intervalo (b - a)
    s = ((b - a) / n) * u_func.sum()

    return s
```

1. **def monte**$_carlo(f, a, b, n)$ :: $Essa linha de fine uma fun$**ção** $chamada$ **monte_carlo** $que recebe quatro parâ$
   $a fun$**ção** **f**, $os limites de integra$**ção** **a** **e** **b**, $e o número de pontos aleatórios$ **n**.

2. **u = np.random.uniform(low=a, high=b, size=n)**: Aqui, a função **np.random.uniform** é usada para gerar **n** pontos aleatórios (variáveis u) distribuídos uniformemente entre **a** e **b**. Esses pontos são amostras aleatórias que serão utilizadas para calcular a aproximação do integral.

3. **u_func = f(u)**: A função **f** é aplicada a cada um dos pontos aleatórios **u** gerados. Isso cria um novo conjunto de valores **u_func**, que corresponde aos valores de **f(u)** para cada **u**.

4. **s = ((b - a) / n) * u_func.sum()**: Aqui, é calculada a estimativa do integral usando o método de Monte Carlo. A média dos valores **u_func** é multiplicada pela largura do intervalo **(b - a)**, representando uma aproximação da área sob a curva da função entre **a** e **b**.

5. **return s**: A função retorna o resultado da estimativa do integral.

6.2 Exercício 2

```
1 def monte_carlo(f, a, b, n):
2     x = np.random.uniform(low=a, high=b, size=n)
3     fx = f(x)
4     s = (b - a) * np.mean(fx)
5     return s
```

A função **monte_carlo** implementa o método de Monte Carlo para calcular uma aproximação numérica do integral definida de uma função $f(x)$ no intervalo $[a, b]$.

Parâmetros:

- **f**: É o primeiro parâmetro da função e representa a função que queremos integrar. Nesse contexto, f é uma função que aceita um array x como entrada e retorna um array com os valores da função aplicada a cada elemento de x.
- **a**: É o segundo parâmetro e representa o limite inferior do intervalo de integração.
- **b**: É o terceiro parâmetro e representa o limite superior do intervalo de integração.
- **n**: É o quarto parâmetro e indica o número de pontos aleatórios que serão gerados no intervalo $[a, b]$ para realizar a aproximação.

Algoritmo:

O algoritmo da função **monte_carlo** é composto pelos seguintes passos:

1. Gera n pontos aleatórios no intervalo $[a, b]$ usando a função **np.random.uniform()** e armazena-os em um array chamado x.
2. Aplica a função $f()$ a cada um dos pontos gerados (x), resultando em um novo array chamado fx, contendo os valores de $f(x)$ para cada valor de x gerado aleatoriamente.
3. Calcula a média ponderada dos valores obtidos em fx multiplicando essa média ponderada pela diferença entre os limites do intervalo $[a, b]$ (ou seja, $(b - a)$).
4. Retorna o valor calculado, que é a aproximação numérica do integral de f no intervalo $[a, b]$.

O método de Monte Carlo baseia-se na ideia de que, quanto mais pontos aleatórios forem gerados no intervalo de integração, mais precisa será a aproximação da integral. Esse método é útil quando a função a ser integrada não possui uma forma analítica simples ou quando o integral é de difícil cálculo por outros métodos tradicionais. Ele é amplamente usado em simulações e problemas de alta dimensionalidade. A principal vantagem é a simplicidade de implementação e a capacidade de lidar com funções complexas. No entanto, a precisão da aproximação pode depender do número de pontos n gerados aleatoriamente, sendo necessário aumentar n para obter resultados mais precisos.

```
def f(t):
    return 10 * np.exp(-t) * np.sin(2 * np.pi * t)
```

Define **f(t)** como:

$$f(t) = 10\, e^{(-t)}\, \sin(2\pi\, t)$$

```
def simpson(f, a, b, n):
    h = (b - a) / n
    integral3 = f(a) + f(b)
    for i in range(1, n):
        xi = a + i * h
        if i % 2 == 0:
            integral3 += 2 * f(xi)
        else:
            integral3 += 4 * f(xi)
    integral3 *= h / 3
    return integral3
```

Aqui, temos outra função chamada simpson, que implementa o método de integração numérica de Simpson. Esse método é baseado na aproximação da função integrada por segmentos de parábolas. Ele divide o intervalo $[a, b]$ em **n** sub-intervalos e aproxima o integral em cada sub-intervalo usando uma fórmula específica. Ao final, somam-se as contribuições de todos os sub-intervalos para obter a aproximação do integral.

```
a = 0
b = 1
valor_exato = 0.98120
```

Nesta parte, são definidas algumas variáveis: **a** e **b** representam os limites do intervalo de integração, e **valor_exato** contém o valor exato do integral da função **f** no intervalo $[a, b]$.

```
n = int(input("Digite o n mero de itera  es: "))
```

Aqui, o código solicita ao usuário que digite o número de iterações (**n**). Esse valor será usado para calcular a aproximação numérica do integral usando ambos os métodos, Monte Carlo e Simpson.

```
start_time = time.time()
result_mc = monte_carlo(f, a, b, n)
mc_time = time.time() - start_time

start_time = time.time()
result_simpson = simpson(f, a, b, n)
simpson_time = time.time() - start_time
```

Estas linhas medem o tempo de execução para cada método. O código chama as funções **monte_carlo** e **simpson** com os parâmetros fornecidos (**f**, **a**, **b** e **n**) e armazena os resultados em **result_mc** e **result_simpson**, respetivamente. O tempo de execução para cada método é calculado e armazenado em **mc_time** e **simpson_time**.

```
print(f"Integral pelo m todo de Monte Carlo: {result_mc:.5f}, tempo: {mc_time
    :.5f}s")
print(f"Integral pelo m todo de Simpson: {result_simpson:.5f}, tempo: {
    simpson_time:.5f}s")
print(f"Valor exato: {valor_exato:.5f}")
```

Estas linhas imprimem no ecrã os resultados obtidos com ambos os métodos, juntamente com o valor exato do integral da função **f** no intervalo $[a, b]$.

```
erro_mc = abs(result_mc - valor_exato)
erro_simpson = abs(result_simpson - valor_exato)
print(f"Erro do m todo de Monte Carlo: {erro_mc:.5f}")
print(f"Erro do m todo de Simpson: {erro_simpson:.5f}")
```

Por fim, o código calcula os erros das aproximações feitas por ambos os métodos, comparando os resultados obtidos com o valor exato. Os erros são impressos no ecrã.

6.3 Exercício 3

A única alteração realizada no código foi:

```
def f_traco(f, a, b, n):
    h = (b - a) / n
    som = 0.0
    for i in range(n):
        xi = a + h/2 + i*h
        som += f(xi)
    return som/n
```

- Define a função **f_traco(f, a, b, n)**, que calcula o valor médio de **f(x)** no intervalo $[a, b]$. Essa função é usada para determinar a média dos valores da função **f(x)** para os pontos intermediários entre os sub-intervalos.
- **h = (b - a) / n**: Calcula o tamanho do sub-intervalo **h** com base no número de sub-intervalos **n**.
- O loop for percorre todos os sub-intervalos e atualiza a variável **som** acumulando os valores da função **f()** nos pontos intermediários (**xi = a + h/2 + i*h**).

- **return som/n**: Retorna a média dos valores da função **f(x)** calculada a partir dos pontos intermediários.

Ao executar o programa para o integral do problema *2*, observamos que o método de Monte Carlo fornece um resultado muito mais rápido do que o método de Simpson. Isso acontece porque o método de Monte Carlo não requer que a função seja diferenciável e, portanto, pode lidar com funções mais complexas. Além disso, o método de Monte Carlo usa amostras aleatórias para aproximar ao integral, o que pode ser mais eficiente para integrais de alta dimensão.

No entanto, quando aplicamos o mesmo programa ao integral do problema 1, observamos que o método de Monte Carlo apresenta um erro muito maior em comparação com o método de Simpson. Isso ocorre porque a função do problema *1* tem uma singularidade em $\mathbf{x = 0}$, e o método de Monte Carlo pode não capturá-la com precisão.

Para corrigir esse problema, podemos modificar o método de Monte Carlo amostrando a função perto da singularidade com mais pontos para obter uma melhor estimativa do integral.

Em relação à questão de saber se contar os pontos sob a curva ou usar a média dos valores da função ($\overline{f}$) fornece uma melhor aproximação, isso depende da natureza da função e do método de integração utilizado. Em geral, usar a média dos valores da função pode ser mais preciso se a função for suave e bem comportada, enquanto contar os pontos sob a curva pode ser mais eficiente para funções complexas ou integrais de alta dimensão.

Ficha 7

7.1 Exercício 1

```
import numpy as np
from numpy.lib._datasource import open as open_url
```

Estas linhas importam a biblioteca **NumPy** e a função **open_url** da sub-biblioteca **_datasource** dentro do pacote **NumPy**. Isso será usado posteriormente para carregar dados de um URL. ***Atenção***, ao importar o **NumPy**, não é necessário importar a função **open** com o nome **open_url** da sub-biblioteca **_datasource**. A linha **from numpy.lib._datasource import open as open_url** é desnecessária e pode ser removida. O **NumPy** por si só é suficiente para utilizar as funcionalidades da biblioteca.

```
def gaussian_elimination(A, b, pivoting=True):
    n = A.shape[0]
    Ab = np.concatenate((A, b.reshape(n, 1)), axis=1)
    for i in range(n-1):
        if pivoting:
            pivot_row = np.argmax(np.abs(Ab[i:, i])) + i
            Ab[[i, pivot_row]] = Ab[[pivot_row, i]]
        pivot = Ab[i, i]
        if pivot == 0:
            raise ValueError('Zero pivot encountered, unable to proceed with elimination.')
        Ab[i, :] /= pivot
        for j in range(i+1, n):
            multiplier = Ab[j, i]
            Ab[j, :] -= multiplier * Ab[i, :]
    if Ab[-1, -2] == 0:
        raise ValueError('Zero pivot encountered in the last row, unable to solve the system uniquely.')
    x = np.zeros(n)
    x[-1] = Ab[-1, -1] / Ab[-1, -2]
    for i in range(n-2, -1, -1):
        x[i] = (Ab[i, -1] - Ab[i, i+1:n] @ x[i+1:]) / Ab[i, i]
```

```
21     return x
```

Esta função **gaussian_elimination** implementa a eliminação de Gauss para resolver um sistema linear. Ela recebe três argumentos: **A** (uma matriz), **b** (um vetor) e **pivoting** (um valor boolean que indica se o pivô deve ser usado ou não). Aqui está o que o código faz dentro da função:

- **n = A.shape[0]** obtém o número de linhas da matriz **A** (assumindo que **A** é uma matriz quadrada).
- **Ab = np.concatenate((A, b.reshape(n, 1)), axis=1)** conjuga a matriz **A** e o vetor **b** horizontalmente para formar a matriz aumentada **Ab**.
- O loop **for i in range(n-1)**: itera sobre as linhas de **Ab** (exceto a última) para executar a eliminação de Gauss.
- Se pivoting for verdadeiro, a seguinte sequência de comandos é executada para realizar o pivoteamento parcial:
  - **pivot_row = np.argmax(np.abs(Ab[i:, i])) + i** encontra o índice da linha com o maior valor absoluto na coluna atual (**i**) a partir da linha **i** em diante.
  - **Ab[[i, pivot_row]] = Ab[[pivot_row, i]]** troca as linhas **i** e **pivot_row** em **Ab**, realizando o pivoteamento parcial.
- **pivot = Ab[i, i]** obtém o valor do pivô na linha atual.
- Se o pivô for zero, uma exceção **ValueError** é lançada, indicando que não é possível prosseguir com a eliminação.
- **Ab[i, :] /= pivot** divide a linha atual pelo pivô para normalizá-la.
- O loop **for j in range(i+1, n)**: itera sobre as linhas abaixo da linha atual para executar a eliminação por subtração.
  - **multiplier = Ab[j, i]** obtém o multiplicador para a eliminação da linha **j**.
  - **Ab[j, :] multiplier * Ab[i, :]** subtrai **multiplier** vezes a linha atual (**i**) da linha **j** em **Ab**.
- Se o pivô da última linha for zero, uma exceção **ValueError** é lançada, indicando que o sistema não pode ser resolvido de forma única.
- **x = np.zeros(n)** cria um vetor de zeros de tamanho **n**.
- **x[-1] = Ab[-1, -1] / Ab[-1, -2]** calcula o valor da última variável desconhecida **x[n-1]**.
- O loop **for i in range(n-2, -1, -1)**: itera reversamente pelas linhas restantes de **Ab** para calcular os valores das variáveis desconhecidas restantes.

  - **x[i] = (Ab[i, -1] - Ab[i, i+1:n] @ x[i+1:]) / Ab[i, i]** calcula o valor de **x[i]** com base nos valores anteriores de **x** e nos coeficientes da linha **i** em **Ab**. Nota: O operador **@** em Python é utilizado para realizar a multiplicação matricial ou o produto interno entre **arrays**. Ele é conhecido como o operador "**matmul**"(multiplicação de matrizes).

- Retorna o vetor **x** com as soluções do sistema linear.

```
url = "https://trixi.coimbra.lip.pt/data/fc/fc07/f07-p1a.npz"
with open_url(url, "rb") as file:
    npzfile = np.load(file)
    A = npzfile['A']
    b = npzfile['b']
```

Estas linhas definem um URL de onde os dados serão carregados e, em seguida, carregam o arquivo **.npz** correspondente. O arquivo contém dois **arrays**, **A** e **b**, que são extraídos do arquivo carregado.

```
pivoting = input('Do you want to use pivoting? (y/n): ').lower() == 'y'
x = gaussian_elimination(A, b, pivoting)
print('Solution: ', x)
```

Esta parte do código solicita ao usuário se o pivoteamento deve ser usado ou não. O valor de entrada é convertido para minúsculas e comparado com **'y'**. O resultado é armazenado na variável **pivoting**. Em seguida, a função **gaussian_elimination** é chamada com os argumentos **A**, **b** e **pivoting**, o resultado é armazenado em **x**. Por fim, a solução **x** é apresentado no ecrã.

```
residuals = A @ x - b
largest_error = np.max(np.abs(residuals))
sum_of_errors = np.sum(np.abs(residuals))
print('Largest absolute error: ', largest_error)
print('Sum of absolute errors: ', sum_of_errors)
```

Estas linhas calculam os resíduos (diferença entre **A @ x e b)**, o maior erro absoluto e a soma dos erros absolutos. Os resultados são apresentados no ecrã.

7.2 Exercício 2

Se não for usada pivotagem, obtemos um erros superior que está de acordo com o esperado.

7.3 Exercício 3

```
def inverse(A):
    n = A.shape[0]
    A_inv = np.zeros((n, n))
    for i in range(n):
        b = np.zeros(n)
        b[i] = 1
        x = gaussian_elimination(A.copy(), b)
        A_inv[:, i] = x
    return A_inv
```

Esta função **inverse** calcula a matriz inversa de uma matriz **A**. Ela recebe um argumento **A** (uma matriz) e retorna a matriz inversa **A_inv**. Aqui está o que o código faz dentro da função:

- **n = A.shape[0]** obtém o número de linhas da matriz **A**.
- **A_inv = np.zeros((n, n))** cria uma matriz de zeros de tamanho **(n, n)** para armazenar a matriz inversa **A_inv**.
- O loop **for i in range(n)**: itera sobre as colunas de **A_inv**.
  - **b = np.zeros(n)** cria um vetor de zeros **b** de tamanho **n**.
  - **b[i] = 1** atribui o valor 1 ao elemento **i** do vetor **b**. Isso cria um vetor de base, onde apenas o **i-ésimo** elemento é não nulo.
  - **x = gaussian_elimination(A.copy(), b)** chama a função **gaussian_elimination** para resolver o sistema linear usando a eliminação de Gauss, passando a matriz **A** e o vetor **b**.
  - **A_inv[:, i] = x** atribui o vetor **x** como a **i-ésima** coluna da matriz inversa **A_inv**.
- Retorna a matriz inversa **A_inv**.

```
def gaussian_elimination(A, b):
    # C digo da fun   o gaussian_elimination
    # igual ao c digo anterior
```

Esta função **gaussian_elimination** é exatamente igual à função **gaussian_elimination** do código anterior. Ela realiza a eliminação de Gauss para resolver um sistema linear, recebendo uma matriz **A** e um vetor **b** como argumentos.

```
url = "https://trixi.coimbra.lip.pt/data/fc/fc07/f07-p1c.npz"
with open_url(url, "rb") as file:
    npzfile = np.load(file)
    A = npzfile['A']
```

Estas linhas definem um URL de onde os dados serão carregados e, em seguida, carregam o arquivo **.npz** correspondente. O arquivo contém a matriz **A**, que é extraída do arquivo carregado.

```
A_inv = inverse(A)

print('Inverse of A: ')
print(A_inv)
```

Estas linhas calculam a matriz inversa de **A** chamando a função **inverse** e armazenam o resultado em **A_inv**. Em seguida, a matriz inversa **A_inv** é exibida no ecrã.

Portanto, as principais diferenças em relação ao código do *exercício 1* são a adição da função **inverse** para calcular a matriz inversa e a alteração do arquivo **.npz** carregado a partir do URL.

7.4 Exercício 4

Destacando as diferenças em relação ao código anterior:

```
url = "https://trixi.coimbra.lip.pt/data/fc/fc07/f07-p3a.npz"
with open_url(url, "rb") as file:
    npzfile = np.load(file)
    A = npzfile['A']
    b = npzfile['b']
```

Estas linhas definem um URL de onde os dados serão carregados e, em seguida, carregam o arquivo **.npz** correspondente. O arquivo contém a matriz **A** e o vetor **b**, que são extraídos do arquivo carregado.

```
print('Without pivoting:')
x = gaussian_elimination(A, b, False)
print('Solution: ', x)
```

Estas linhas resolvem o sistema de equações usando a eliminação de Gauss sem pivotagem. A função **gaussian_elimination** é chamada com a matriz **A**, o vetor **b** e o argumento **pivoting=False**. A solução **x** é exibida no ecrã.

```
residuals = A @ x - b
largest_error = np.max(np.abs(residuals))
sum_of_errors = np.sum(np.abs(residuals))
print('Largest absolute error: ', largest_error)
print('Sum of absolute errors: ', sum_of_errors)
print('-------------------------------------------------------')
```

Estas linhas calculam os erros residuais do sistema resolvido comparando a multiplicação da matriz **A** pela solução **x** com o vetor **b**. O maior erro absoluto e a soma dos erros absolutos são calculados e exibidos no ecrã.

```
url2 = "https://trixi.coimbra.lip.pt/data/fc/fc07/f07-p3d.npz"
with open_url(url2, "rb") as file:
    npzfile = np.load(file)
    B = npzfile['A']
    d = npzfile['b']
```

Estas linhas definem um novo URL de onde os dados serão carregados e, em seguida, carregam o arquivo **.npz** correspondente. O arquivo contém a matriz **B** e o vetor **d**, que são extraídos do arquivo carregado.

```
print('Without pivoting: ')
x2 = gaussian_elimination(B, d, False)
print('Solution: ', x2)
```

Estas linhas resolvem o sistema de equações usando a eliminação de Gauss sem pivotagem para a matriz **B** e o vetor **d**. A solução **x2** é exibida no ecrã.

```
residuals2 = B @ x - d
largest_error2 = np.max(np.abs(residuals2))
sum_of_errors2 = np.sum(np.abs(residuals2))
print('Largest absolute error: ', largest_error2)
print('Sum of absolute errors: ', sum_of_errors2)
```

Estas linhas calculam os erros residuais do sistema resolvido comparando a multiplicação da matriz **B** pela solução **x** com o vetor **d**. O maior erro absoluto e a soma dos erros absolutos são calculados e exibidos no ecrã.

```
print('With pivoting:')
x = gaussian_elimination(A, b, True)
print('Solution: ', x)
```

Estas linhas resolvem o sistema de equações usando a eliminação de Gauss com pivotagem para a matriz **A** e o vetor **b**. A solução **x** é exibida no ecrã.

```
residuals = A @ x - b
largest_error = np.max(np.abs(residuals))
sum_of_errors = np.sum(np.abs(residuals))
print('Largest absolute error: ', largest_error)
print('Sum of absolute errors: ', sum_of_errors)
print('-------------------------------------------------------')
```

Estas linhas calculam os erros residuais do sistema resolvido comparando a multiplicação da matriz **A** pela solução **x** com o vetor **b**. O maior erro absoluto e a soma dos erros absolutos são calculados e exibidos no ecrã.

```
print('Without pivoting: ')
x2 = gaussian_elimination(B, d, True)
print('Solution: ', x2)
```

Estas linhas resolvem o sistema de equações usando a eliminação de Gauss com pivotagem para a matriz **B** e o vetor **d**. A solução **x2** é exibida no ecrã.

```
residuals2 = B @ x - d
largest_error2 = np.max(np.abs(residuals2))
sum_of_errors2 = np.sum(np.abs(residuals2))
print('Largest absolute error: ', largest_error2)
print('Sum of absolute errors: ', sum_of_errors2)
```

Estas linhas calculam os erros residuais do sistema resolvido comparando a multiplicação da matriz **B** pela solução **x** com o vetor **d**. O maior erro absoluto e a soma dos erros absolutos são calculados e exibidos no ecrã. ***Nota***, na linha 1 acima devia ser **x2** e não **x**.

No final, o código compara os resultados obtidos com e sem pivotagem para os sistemas de equações dos arquivos **f07-p3a.npz** e **f07-p3d.npz**.

7.4.1 Correção exercício 4

Embora o código imprima o pedido, é difícil perceber qual é qual, abaixo encontra-se essa secção do código corrigida.

```
print('Without pivoting(A):')
x = gaussian_elimination(A, b, False)
print('Solution: ', x)

# Check the result
residuals = A @ x - b
largest_error = np.max(np.abs(residuals))
sum_of_errors = np.sum(np.abs(residuals))
print('Largest absolute error: ', largest_error)
print('Sum of absolute errors: ', sum_of_errors)
print('-------------------------------------------------------')

print('Without pivoting(B): ')
x2 = gaussian_elimination(B, d, False)
print('Solution: ', x2)

# Check the result
residuals2 = B @ x2 - d
largest_error2 = np.max(np.abs(residuals2))
```

```
sum_of_errors2 = np.sum(np.abs(residuals2))
print('Largest absolute error: ', largest_error2)
print('Sum of absolute errors: ', sum_of_errors2)

print('=====================================================')

print('With pivoting (A):')
x = gaussian_elimination(A, b, True)
print('Solution: ', x)

# Check the result
residuals = A @ x - b
largest_error = np.max(np.abs(residuals))
sum_of_errors = np.sum(np.abs(residuals))
print('Largest absolute error: ', largest_error)
print('Sum of absolute errors: ', sum_of_errors)
print('-----------------------------------------------------')


print('Without pivoting (B): ')
x2 = gaussian_elimination(B, d, True)
print('Solution: ', x2)

# Check the result
residuals2 = B @ x2 - d
largest_error2 = np.max(np.abs(residuals2))
sum_of_errors2 = np.sum(np.abs(residuals2))
print('Largest absolute error: ', largest_error2)
print('Sum of absolute errors: ', sum_of_errors2)
```

Ficha 8

8.1 Exercício 1

```
def three_point_rule(f, x, h):
    return (f(x + h) - f(x - h)) / (2 * h)
```

Aqui, é definida uma função chamada **three_point_rule**, que calcula a derivada de primeira ordem de uma função **f** em um ponto **x** usando a regra de três pontos. O parâmetro **h** é a diferença entre os pontos de **x** para calcular a derivada.

```
def five_point_rule(f, x, h):
    return (-f(x + 2 * h) + 8 * f(x + h) - 8 * f(x - h) + f(x - 2 * h)) / (12 *
    h)
```

Esta função **five_point_rule** calcula a derivada de primeira ordem usando a regra de cinco pontos. Ela é mais precisa do que a regra de três pontos, pois utiliza mais pontos em torno de **x**.

```
def three_point_rule_second_derivative(f, x, h):
    return (f(x + h) - 2 * f(x) + f(x - h)) / (h ** 2)
```

Aqui, é definida a função **three_point_rule_second_derivative**, que calcula a segunda derivada de uma função **f** em um ponto **x** usando a regra de três pontos.

```
def five_point_rule_second_derivative(f, x, h):
    return (-f(x + 2 * h) + 16 * f(x + h) - 30 * f(x) + 16 * f(x - h) - f(x - 2
    * h)) / (12 * (h ** 2))
```

A função **five_point_rule_second_derivative** calcula a segunda derivada usando a regra de cinco pontos.

```
f = lambda x: np.exp(x)
```

Aqui, uma função **f(x) = exp(x)** é definida usando uma expressão lambda.

Uma função lambda, também conhecida como função anónima, é uma forma concisa de definir uma função em Python. É chamada de "anónima"porque não recebe um nome como as funções definidas usando a declaração 'def'. Em vez disso, é criada usando a palavra-chave 'lambda'.

A sintaxe geral de uma função lambda é a seguinte:

```
lambda argumentos: express o
```

Aqui está a explicação da função lambda utilizada no código:

```
f = lambda x: np.exp(x)
```

Neste caso, a função lambda tem um único argumento '**x**', e a expressão é '**np.exp(x)**'. Isso significa que, quando você passar um valor para '**x**', a função lambda irá calcular o valor de '**np.exp(x)**' e retorná-lo.

'**np.exp(x)**' é uma função da biblioteca **NumPy** que calcula o exponencial de '**x**', ou seja, '**e**' elevado à potência de '**x**'.

Com a declaração acima, pode-se usar '**f**' como uma função normal em qualquer lugar do código. Por exemplo, '**f(2)**' retornaria o valor de '**e**' elevado à potência de **2**.

```
x = np.linspace(0, 12, num=1000)
y = f(x)
```

Aqui, é criado um array **x** contendo 1000 pontos espaçados uniformemente entre 0 e 12. Em seguida, a função **f(x)** é avaliada nesses pontos para obter os valores correspondentes de **y**.

```
dydx_3 = three_point_rule(f, x, 0.001)
dydx_5 = five_point_rule(f, x, 0.001)
d2ydx2_3 = three_point_rule_second_derivative(f, x, 0.001)
d2ydx2_5 = five_point_rule_second_derivative(f, x, 0.001)
```

Aqui, as derivadas de primeira e segunda ordem são calculadas nos pontos do array **x** usando as funções definidas anteriormente, com um valor de **h** igual a 0.001. Isso significa que a diferença entre os pontos usados para o cálculo das derivadas será de 0.001.

```
plt.plot(x, y, label='f(x)')
plt.plot(x, dydx_3, label="f'(x) (3-point rule)")
plt.plot(x, dydx_5, label="f'(x) (5-point rule)")
plt.plot(x, d2ydx2_3, label="f''(x) (3-point rule)")
plt.plot(x, d2ydx2_5, label="f''(x) (5-point rule)")
plt.legend()
plt.show()
```

Nestas linhas, o gráfico é traçado. O gráfico contém cinco curvas: a função original **f(x)**, a derivada de primeira ordem calculada com a regra de três pontos, a derivada de primeira ordem calculada com a regra de cinco pontos, a segunda derivada calculada com a regra de três pontos e a segunda derivada calculada com a regra de cinco pontos. As legendas são adicionadas para facilitar a identificação das curvas. A função **plt.show()** exibe o gráfico no ecrã.

O gráfico resultante mostrará as curvas correspondentes à função exponencial original e as suas derivadas de primeira e segunda ordem estimadas usando as regras de diferenças finitas. Isso pode ser útil para visualizar como as derivadas se comportam ao longo do intervalo de 0 a 12.

Quando o tamanho do passo h é sucessivamente reduzido de 10^{-1} para 10^{-14}, os erros relativos nas primeiras e segundas derivadas diminuem, teoricamente. Isso ocorre porque à medida que **h** diminui, a aproximação da derivada usando o método de diferenças finitas torna-se mais preciso. No entanto, após um certo ponto ($h < 10^{-8}$), os erros relativos começam a aumentar novamente devido a erros de arredondamento. Isso significa que existe um valor ótimo de **h** que minimiza o erro relativo. Para este exemplo e função específicos (e^x), parece que um valor ótimo de **h** é em torno de 10^{-8}.

8.2 Exercício 2

```
def newton(g, dydx_3, x0, eps_1, eps_2, k_max):
    x = x0
    k = 1
    while k < k_max:
        x_prev = x
        x = x - g(x) / dydx_3
        if abs(g(x)) < eps_1 and abs(x - x_prev) < eps_2 * abs(x):
            print(f"Converge em {k} itera  es.")
            return x
        k += 1
    print(f"Falhou em convergir em {k_max} itera  es.")
    return x
```

Aqui, é definida uma função chamada newton. Esta função implementa o método de Newton para encontrar uma raiz da função **g(x)**. Os parâmetros da função são:

- **g**: A função para a qual desejamos encontrar uma raiz.
- **dydx_3**: A derivada de primeira ordem de **g(x)** calculada usando a regra de três pontos.
- **x0**: O ponto inicial onde começamos a busca pela raiz.
- **eps_1**: A tolerância para a convergência do valor da função para zero.
- **eps_2**: A tolerância para a convergência da raiz entre iterações consecutivas.
- **k_max**: O número máximo de iterações permitidas para o método de Newton.

Dentro da função, é utilizado um loop while para realizar as iterações do método de Newton. A cada iteração, o ponto **x** é atualizado usando a fórmula do método de Newton, que é **x = x - g(x) / dydx_3**. O loop continua até que a convergência seja alcançada (conforme as condições **if**), ou até que o número máximo de iterações **k_max** seja atingido. Caso o método não tenha sucesso em encontrar a raiz dentro do número máximo de iterações, uma mensagem de falha é exibida, e o último valor de **x** é retornado.

```
def three_point_rule(g, x, h):
    return (g(x + h) - g(x - h)) / (2 * h)
```

Aqui é definida a função **three_point_rule**, que calcula a derivada de primeira ordem de uma função **g(x)** usando a regra de três pontos. A função recebe três argumentos: a função **g**, o ponto **x** onde se quer calcular a derivada e o valor de **h**, que é a diferença entre os pontos de **x** para calcular a derivada.

```
g = lambda x: np.log(x)+(1/x**2)-1
x = np.logspace(0, 12, num=1000)
y = g(x)
```

Nestas linhas, a função **g(x)** é definida utilizando uma expressão lambda. A função **g(x)** é a função para a qual queremos encontrar a raiz. A seguir, um array **x** é criado contendo 1000 pontos espaçados logaritmicamente entre 1 e 10^{12}. A função **g(x)** é então avaliada nesses pontos, e os valores correspondentes são armazenados em **y**.

```
x0 = float(input("Digite o valor de x0: "))
```

O programa solicita ao usuário que insira o valor inicial **x0** para iniciar a busca pela raiz.

```
true_dydx = g(x0)
best_h_dydx_3 = 0
best_error_dydx_3 = np.inf
for k in range(1, 15):
    h = 10 ** (-k)
    dydx_3 = three_point_rule(g, x0, h)
    rel_error_dydx_3 = abs((dydx_3 - true_dydx) / true_dydx)
    if rel_error_dydx_3 < best_error_dydx_3:
        best_h_dydx_3 = h
        best_error_dydx_3 = rel_error_dydx_3
```

Neste trecho, o código encontra o melhor valor de **h** para a regra de três pontos, que proporciona a menor taxa de erro relativo na estimativa da derivada de primeira ordem em **x0**. Para isso, ele testa valores de **h** variando de 10^{-1} a 10^{-14} usando um loop **for**. A função **three_point_rule** é chamada para calcular a derivada de primeira ordem em **x0** com cada valor de **h**. Em seguida, é calculado o erro relativo entre a estimativa e o valor real da derivada em

x0. O melhor valor de **h** é aquele que minimiza o erro relativo, e o erro mínimo é armazenado em **best_error_dydx_3**.

```
dydx_3 = three_point_rule(g, x0, best_h_dydx_3)
```

Aqui, a derivada de primeira ordem em **x0** é recalculada usando a regra de três pontos com o melhor valor de **h** encontrado anteriormente.

```
k_max = int(input("Digite o n mero  m ximo de itera  es: "))
eps_1 = float(input("Digite o valor de eps_1: "))
eps_2 = float(input("Digite o valor de eps_2: "))
result = newton(g, dydx_3, x0, eps_1, eps_2, k_max)
print(f"A raiz da fun   o   : {result}")
```

Nesta parte do código, o usuário é solicitado a fornecer o número máximo de iterações **k_max**, bem como os valores de tolerância **eps_1** e **eps_2**. O programa, então, utiliza a função newton para encontrar a raiz da função **g(x)** usando o método de Newton, a partir do ponto inicial **x0**, a derivada de primeira ordem **dydx_3** calculada anteriormente, e as tolerâncias fornecidas. O resultado é a raiz da função, que é exibida na tela.

Resumindo, o código calcula a derivada de primeira ordem da função **g(x)** usando a regra de três pontos com diferentes valores de **h**, encontra o melhor valor de **h** com o menor erro relativo, e em seguida, aplica o método de Newton para encontrar a raiz da função **g(x)** a partir do ponto inicial **x0**. O resultado final é a raiz da função encontrada pelo método de Newton.

Ficha 9

9.1 Exercício 1

```
def euler(h):
    n = int((2 - 0)/h) + 1
    x = np.linspace(0, 2, n)
    y = np.zeros(n)
    y[0] = 1
    for i in range(n-1):
        y[i+1] = y[i] + h*(y[i]*x[i]**2 - y[i])
    return x, y
```

Esta função chamada **euler(h)** implementa o método de Euler. Recebe como argumento o parâmetro **h**, que representa o tamanho do passo utilizado na discretização do intervalo [0, 2]. Dentro da função, o número de pontos de discretização **n** é calculado com base no valor de **h**. Em seguida, é criado um array **x** com **n** pontos igualmente espaçados no intervalo [0, 2]. Um array **y** é inicializado com zeros e o valor inicial **y[0]** é definido como **1**.

A partir do segundo ponto (**índice 1**) até o último ponto (**índice n-1**), o método de Euler é aplicado para calcular **y[i+1]** com base no valor anterior **y[i]**, utilizando a fórmula específica. A função retorna os arrays **x** e **y** resultantes.

```
x = np.linspace(0, 2, 200)
y_analytical = np.exp((x**3)/3 - x)
```

Estas linhas criam um array **x** com 200 pontos igualmente espaçados no intervalo [0, 2]. Em seguida, é calculado um array **y_analytical** contendo a solução analítica da equação diferencial para cada valor de **x**, utilizando uma fórmula específica.

```
x1, y1 = euler(0.01)
x2, y2 = euler(0.0001)
```

Estas linhas chamam a função **euler(h)** para dois valores diferentes de **h**, 0.01 e 0.0001, obtendo os arrays **x1**, **y1** e **x2**, **y2**, respetivamente.

```
plt.plot(x, y_analytical, label='Analytical', color='r')
plt.plot(x1, y1, label='Euler h=0.01', color='g')
plt.plot(x2, y2, label='Euler h=0.0001', color='b',alpha=0.5)
plt.legend()
plt.title('Comparison of Analytical Solution and Numerical Approximations\n (0
    <= x <= 2)')
plt.xlabel('x')
plt.ylabel('y')
plt.grid(color='grey', linestyle='--')
plt.savefig('plot.png', dpi=700)
plt.show()
```

Estas linhas utilizam a biblioteca **matplotlib** para criar um gráfico comparando a solução analítica da equação diferencial com as aproximações numéricas obtidas pelo método de Euler. O comando **plt.plot()** é usado para traçar as curvas correspondentes aos valores de **x**, **y_analytical**, **x1**, **y1**, **x2** e **y2**. A função **plt.legend()** adiciona uma legenda ao gráfico, **plt.title()**, **plt.xlabel()** e **plt.ylabel()** definem os títulos dos eixos e **plt.grid()** adiciona uma grelha ao gráfico. O comando **plt.savefig()** salva o gráfico em um arquivo chamado "**plot.png**"e **plt.show()** exibe o gráfico no ecrã.

9.2 Exercício 2

```
m = 50.0  # massa (kg)
c = 2.0e4  # constante de elasticidade (N/m)
b_values = [5000, 2500, 1000, 500, 100, 50]  # constante de amortecimento (kg/s)
```

Estas linhas definem as constantes do problema, ou seja, a massa do objeto, a constante de elasticidade e uma lista de valores para a constante de amortecimento. Esses valores serão usados posteriormente para calcular e representar graficamente a solução da equação diferencial.

```
dt = float(input("Introduza o tamanho da itera  o (s): ")) # passo de tempo (s
    )
t = np.arange(0, 1+dt, dt) # array de tempo
z0 = 0.1 # deslocamento inicial (m)
v0 = 0 # velocidade inicial (m/s)
```

Estas linhas solicitam ao usuário que introduza o tamanho do passo de iteração (**dt**) e, em seguida, criam um array **t** de tempo, variando de 0 a 1 com passo **dt**. Também são definidos os valores iniciais de deslocamento (**z0**) e velocidade (**v0**).

```
for b in b_values:
    z = np.zeros_like(t)
    v = np.zeros_like(t)
    z[0] = z0
    v[0] = v0
```

```
    for i in range(1, len(t)):
        z[i] = z[i-1] + v[i-1]*dt
        v[i] = v[i-1] + (-b/m*v[i-1] - c/m*z[i-1])*dt
    plt.plot(t, z, label=f"b={b}")
```

Este é um loop for que itera sobre os valores da lista **b_values**, que correspondem aos diferentes valores de constante de amortecimento. Dentro do loop, são inicializados arrays **z** e **v** com o mesmo tamanho do array **t**, preenchidos com zeros. Os valores iniciais de **z** e **v** são atribuídos às primeiras posições dos arrays.

Em seguida, há outro loop for que itera sobre o intervalo de 1 até o comprimento de **t**. Dentro desse loop, o método de Euler é aplicado para calcular os valores de **z** e **v** em cada iteração. Os cálculos seguem a fórmula específica para a equação diferencial fornecida. Os valores atualizados são armazenados nos arrays **z** e **v**.

Fora dos loops, é utilizado o comando **plt.plot()** para traçar os gráficos de **t** versus **z** para cada valor de **b**, com uma legenda correspondente ao valor de **b**.

```
plt.legend()
plt.xlabel("Tempo (s)")
plt.ylabel("Deslocamento (m)")
plt.show()
```

Estas linhas adicionam uma legenda ao gráfico, definem os rótulos dos eixos **x**, **y** e exibem o gráfico resultante no ecrã.

9.3 Exercício 3

O código é basicamente igual ao do exercício anterior, de modo que em baixo só está explicado o resto do código.

```
for b in b_values:
    z = np.zeros_like(t)
    v = np.zeros_like(t)
    z[0] = z0
    v[0] = v0
    for i in range(1, len(t)):
        k1 = (-b/m*v[i-1] - c/m*z[i-1])
        z_temp = z[i-1] + v[i-1]*dt
        v_temp = v[i-1] + k1*dt
        k2 = (-b/m*v_temp - c/m*z_temp)
        v[i] = v[i-1] + 0.5*(k1+k2)*dt
        z[i] = z[i-1] + 0.5*(v[i-1]+v[i])*dt
    plt.plot(t, z, label=f"b={b}")
```

Este é um loop **for** que itera sobre os valores da lista **b_values**, que correspondem aos diferentes valores de constante de amortecimento. Dentro do loop, são inicializados arrays **z** e

v com o mesmo tamanho do array **t**, preenchidos com zeros. Os valores iniciais de **z** e **v** são atribuídos às primeiras posições dos arrays.

Em seguida, há outro loop **for** que itera sobre o intervalo de 1 até o comprimento de **t**. Dentro desse loop, o método de *Heun* é aplicado para calcular os valores de **z** e **v** em cada iteração.

Primeiro, é calculado **k1**, que é a taxa de variação de **v** com base nos valores atuais de **v** e **z**. Em seguida, são calculados valores temporários **z_temp** e **v_temp** usando o método de *Euler*.

Após isso, **k2** é calculado usando os valores temporários **v_temp** e **z_temp**. Em seguida, os valores atualizados de **v** e **z** são calculados usando a média ponderada de **k1** e **k2** e o passo de tempo **dt**.

Fora dos loops, é utilizado o comando **plt.plot()** para traçar os gráficos de **t** versus **z** para cada valor de **b**, com uma legenda correspondente ao valor de **b**.

9.3.1 Euler vs Heun

O método de *Heun* tem algumas vantagens em relação ao método de *Euler*, pois utiliza uma aproximação mais precisa ao levar em consideração uma estimativa intermediária para a taxa de variação. Isso resulta em uma maior precisão e estabilidade numérica. No entanto, também requer um maior esforço computacional, pois envolve o cálculo de duas estimativas de taxa de variação em cada iteração.

Ficha 10

10.1 Exercício 1

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

Estas linhas importam as bibliotecas necessárias para o código. O **numpy** é utilizado para realizar cálculos numéricos eficientes e o **matplotlib.pyplot** é utilizado para fazer gráficos.

```
def gaussian_elimination(A, b, pivoting=True):
    n = A.shape[0]
    Ab = np.concatenate((A, b.reshape(n, 1)), axis=1)
    for i in range(n-1):
        if pivoting:
            pivot_row = np.argmax(np.abs(Ab[i:, i])) + i
            Ab[[i, pivot_row]] = Ab[[pivot_row, i]]
        pivot = Ab[i, i]
        if pivot == 0:
            raise ValueError('Zero pivot encountered, unable to proceed with elimination.')
        Ab[i, :] /= pivot
        for j in range(i+1, n):
            multiplier = Ab[j, i]
            Ab[j, :] -= multiplier * Ab[i, :]
    if Ab[-1, -2] == 0:
        raise ValueError('Zero pivot encountered in the last row, unable to solve the system uniquely.')
    x = np.zeros(n)
    x[-1] = Ab[-1, -1] / Ab[-1, -2]
    for i in range(n-2, -1, -1):
        x[i] = (Ab[i, -1] - Ab[i, i+1:n] @ x[i+1:]) / Ab[i, i]
    return x
```

A função **gaussian_elimination(A, b, pivoting=True)** implementa o algoritmo de Eliminação Gaussiana com a opção de pivotação. A função é utilizada para resolver sistemas de

equações lineares representados por uma matriz de coeficientes **A** e um vetor coluna de termos independentes **b**. A pivotação é um processo opcional que ajuda a evitar divisões por zero e melhora a estabilidade numérica do algoritmo.

- **A**: é uma matriz quadrada representando os coeficientes das variáveis do sistema de equações lineares.
- **b**: é um vetor coluna representando os termos independentes do sistema de equações.
- **pivoting**: é um valor booleano opcional que determina se a pivotação será utilizada. Por padrão, a pivotação está ativada (**True**).

O algoritmo inicia concatenando a matriz **A** e o vetor coluna **b** em uma matriz estendida **Ab**. Em seguida, realiza a eliminação gaussiana iterativa passo a passo, eliminando uma variável de cada vez.

Se a pivotação estiver ativada (**pivoting=True**), o algoritmo procura o pivô máximo em valor absoluto na coluna atual e troca as linhas para garantir um pivô não nulo. Isso evita divisões por zero durante o processo de eliminação.

Durante a eliminação gaussiana, o pivô atual é normalizado, tornando-se igual a 1, para simplificar o processo de eliminação. O algoritmo então zera os elementos abaixo do pivô atual para criar uma matriz triangular superior.

Após a etapa de eliminação, o algoritmo verifica se o elemento na última linha e penúltima coluna da matriz **Ab** é igual a zero. Se for zero, isso indica que a última equação se tornou redundante após a eliminação gaussiana, e o sistema não tem uma solução única.

Em seguida, o algoritmo calcula as soluções do sistema a partir da matriz triangular superior resultante. As soluções são armazenadas em um vetor **x**, que é retornado como resultado da função.

Observação: Caso o pivô atual seja zero durante a eliminação gaussiana, o algoritmo levanta um erro (**ValueError**) indicando que não é possível prosseguir com a eliminação. Além disso, se o elemento na última linha e última coluna da matriz **Ab** for zero, o algoritmo também levanta um erro, pois o sistema não tem uma solução única.

```
def polynomial_approximation(x_data, y_data, degree, pivoting=True):
    n = degree + 1
    X = np.zeros((n,n))
    Y = np.zeros(n)
    for i in range(n):
        Y[i] = np.sum(y_data * x_data**i)
        for j in range(n):
            X[i,j] = np.sum(x_data**(i+j))
    coefficients = gaussian_elimination(X,Y,pivoting)
    return coefficients
```

A função **polynomial_approximation(x_data, y_data, degree, pivoting=True)** é utilizada para realizar uma aproximação polinomial dos dados fornecidos, encontrando os coefici-

entes de um polinômio de grau especificado que melhor se ajusta aos pontos de dados **x_data** e **y_data**.

- **x_data**: é um vetor contendo os valores de x dos pontos de dados.
- **y_data**: é um vetor contendo os valores de y correspondentes aos pontos de dados.
- **degree**: é o grau do polinômio de aproximação.
- **pivoting**: é um valor booleano opcional que determina se a pivotação será utilizada durante a Eliminação Gaussiana na função **gaussian_elimination**. Por padrão, a pivotação está ativada (**True**).

O algoritmo da função é explicado abaixo:

1. **n = degree + 1**: Calcula o número de coeficientes a serem encontrados no polinômio de grau **degree**.
2. **X = np.zeros((n,n))**: Cria uma matriz **X** de tamanho $n \times n$ preenchida com zeros. Essa matriz será usada para calcular os elementos do sistema de equações lineares para encontrar os coeficientes do polinômio.
3. **Y = np.zeros(n)**: Cria um vetor **Y** de tamanho n (número de coeficientes) preenchido com zeros. Esse vetor será usado para armazenar as somas dos produtos dos valores de *y_data* com as potências de *x_data*.
4. Loop **for i in range(n)**:
   - Para cada grau **i**, calcula a soma dos produtos dos valores de *y_data* com as potências de *x_data* elevadas a **i**.
5. Loop **for j in range(n)**:
   - Calcula a soma dos elementos de *x_data* elevados à potência **i+j**, para cada **i** e **j**, para criar a matriz **X** do sistema de equações lineares.
6. **coefficients = gaussian_elimination(X,Y,pivoting)**: Chama a função **gaussian_elimination**, passando a matriz **X** e o vetor **Y**, para calcular os coeficientes do polinômio de grau **degree** que melhor se ajusta aos dados.
7. **return coefficients**: Retorna o vetor **coefficients**, que contém os coeficientes do polinômio de aproximação. Esses coeficientes representam o polinômio que melhor se ajusta aos pontos de dados fornecidos.

Essa função é uma parte essencial do processo de aproximação polinomial utilizada posteriormente no código.

```
x_data = np.array([1, 2, 4, 8, 6, 5, 8, 9, 7])
y_data = np.array([2, 3, 4, 7, 6, 5, 8, 8, 6])
```

Estes são os dados de exemplo para os quais desejamos encontrar a aproximação polinomial.

```
degree = 1
```

Esta variável determina o grau do polinómio que desejarmos usar para a aproximação. Isso afeta a complexidade e a precisão da aproximação polinomial.

```
pivoting = input('Do you want to use pivoting? (y/n): ').lower() == 'y'
coefficients = polynomial_approximation(x_data,y_data,degree,pivoting)
```

Esta variável é usada para decidir se é necessário usar o pivoteamento durante a eliminação gaussiana. O pivoteamento ajuda na estabilidade numérica ao lidar com sistemas de equações mal condicionados. O código então calcula a aproximação polinomial chamando **polynomial_approximation** com os dados fornecidos e o grau.

```
x_range = np.linspace(np.min(x_data),np.max(x_data),100)
y_range = np.polyval(coefficients[::-1],x_range)
plt.plot(x_data,y_data,'ro',label='Data')
plt.plot(x_range,y_range,label='Approximation',linestyle='--',alpha=0.7)
plt.xlabel('X')
plt.ylabel('Y')
plt.legend()
plt.show()
```

Por fim, o código gera um gráfico para visualizar os pontos de dados e a função de aproximação polinomial.

Se executarmos este código e respondermos 'y' (para usar o pivoteamento) quando solicitado, ele usará o pivoteamento durante o processo de eliminação gaussiana. Se respondermos 'n', ele não usará o pivoteamento. Lembre-se de que a escolha do grau do polinómio é importante. Um grau mais alto pode levar ao overfitting (sobreajuste), enquanto um grau mais baixo pode resultar em um ajuste inadequado aos dados. É crucial escolher um grau apropriado com base na natureza dos seus dados e no problema que você está tentando resolver.

10.2 Exercício 2

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

Estas linhas importam as bibliotecas necessárias para o código. O **numpy** é utilizado para realizar cálculos numéricos eficientes, o **matplotlib.pyplot** é utilizado para fazer gráficos e o **requests** é utilizado para fazer solicitações HTTP para obter dados de um URL.

```
def gaussian_elimination(A, b, pivoting=True):
    n = A.shape[0]
    Ab = np.concatenate((A, b.reshape(n, 1)), axis=1)
    for i in range(n-1):
        if pivoting:
            pivot_row = np.argmax(np.abs(Ab[i:, i])) + i
            Ab[[i, pivot_row]] = Ab[[pivot_row, i]]
        pivot = Ab[i, i]
        if pivot == 0:
            raise ValueError('Zero pivot encountered, unable to proceed with
    elimination.')
        Ab[i, :] /= pivot
        for j in range(i+1, n):
            multiplier = Ab[j, i]
            Ab[j, :] -= multiplier * Ab[i, :]
    if Ab[-1, -2] == 0:
        raise ValueError('Zero pivot encountered in the last row, unable to
    solve the system uniquely.')
    x = np.zeros(n)
    x[-1] = Ab[-1, -1] / Ab[-1, -2]
    for i in range(n-2, -1, -1):
        x[i] = (Ab[i, -1] - Ab[i, i+1:n] @ x[i+1:]) / Ab[i, i]
    return x
```

Esta função realiza a eliminação gaussiana com opção de pivotação de linhas. Ela resolve o sistema de equações lineares que surge durante o processo de aproximação polinomial.

```
def polynomial_approximation(x_data, y_data, degree, pivoting=True):
    n = degree + 1
    X = np.zeros((n,n))
    Y = np.zeros(n)
    for i in range(n):
        Y[i] = np.sum(y_data * x_data**i)
        for j in range(n):
            X[i,j] = np.sum(x_data**(i+j))
    coefficients = gaussian_elimination(X,Y,pivoting)
    return coefficients
```

Esta função recebe os dados **x_data**, **y_data** e o grau **degree** como entrada. Ela cria uma matriz de Vandermonde X e um vetor Y com base nos pontos de dados fornecidos. Em seguida, ela chama a função **gaussian_elimination** para resolver o sistema de equações e obter os coeficientes do polinómio.

```
D = requests.get("https://trixi.coimbra.lip.pt/data/fc/fc10/f10p2.dat")
```

```
T = D.text
U = T.split('\n')
```

1. **D = requests.get("https://trixi.coimbra.lip.pt/data/fc/fc10/f10p2.dat")**

   Nesta linha, o código utiliza a biblioteca **requests** para realizar uma solicitação HTTP do tipo GET para o URL https://trixi.coimbra.lip.pt/data/fc/fc10/f10p2.dat. Essa URL é utilizada para acessar um arquivo de dados em formato **.dat** hospedado em algum servidor.

2. **T = D.text**

   Nesta linha, o código armazena o conteúdo da resposta da solicitação HTTP realizada anteriormente na variável **T**. A função **.text** é usada para obter o conteúdo do arquivo de dados que foi retornado como resposta da solicitação.

3. **U = T.split('\n')**

   Nesta linha, o código divide o conteúdo do arquivo de dados (que está armazenado em **T**) em linhas, armazenando-as em uma lista chamada **U**. O método **.split('\n')** é usado para dividir o conteúdo com base no caractere de nova linha (**'\n'**), ou seja, cada elemento da lista **U** conterá uma linha do arquivo de dados.

Agora, após executar esse código, as variáveis **T** e **U** contêm informações importantes:

- **T**: É uma string que contém todo o conteúdo do arquivo de dados que foi obtido na resposta da solicitação HTTP. Cada caractere, incluindo espaços e caracteres de nova linha, está presente nessa string.
- **U**: É uma lista em que cada elemento corresponde a uma linha do arquivo de dados. Cada elemento é uma string que contém o conteúdo de uma linha específica, excluindo o caractere de nova linha.

Isso é especialmente útil quando se trabalha com arquivos de dados que possuem várias linhas, e você precisa processar essas linhas individualmente ou realizar operações específicas em cada linha. Por exemplo, é possível que o arquivo de dados contenha uma tabela de valores, e cada linha represente uma entrada na tabela. Ao armazenar cada linha como um elemento da lista **U**, você pode facilmente iterar por cada linha e extrair informações ou realizar cálculos com base nos valores contidos em cada linha.

```
x_data = []
y_data = []
for row in U:
    if row:
        s = row.split()
        x_data.append(float(s[0]))
        y_data.append(float(s[1]))
x_data = np.array(x_data)
y_data = np.array(y_data)
```

1. **x_data = []** e **y_data = []**

   Essas duas linhas criam duas listas vazias chamadas **x_data** e **y_data**. Essas listas serão usadas para armazenar os valores das coordenadas **x** e **y**, respectivamente.

2. **for row in U:**

   Esta linha inicia um loop **for** que percorre cada elemento (**row**) da lista **U**. Lembre-se de que **U** foi criada a partir da divisão do conteúdo do arquivo de dados em linhas (como explicado anteriormente).

3. **if row:**

   Nesta linha, o código verifica se a variável **row** não está vazia. Isso é importante porque algumas linhas podem estar em branco ou conter apenas espaços em branco. O **if row:** garante que apenas linhas não vazias sejam processadas.

4. **s = row.split()**

   Aqui, a string **row** é dividida em uma lista de palavras (ou tokens) usando o método **split()**. Por padrão, o método **split()** divide a string nos espaços em branco (ou seja, usa os espaços como separadores). O resultado dessa divisão é armazenado na variável **s**.

5. **x_data.append(float(s[0]))** e **y_data.append(float(s[1]))**

   Essas duas linhas adicionam os valores convertidos em ponto flutuante (**float**) à lista **x_data** e **y_data**, respetivamente. O **float(s[0])** obtém o primeiro valor da lista **s** (correspondente à coordenada **x**) e o converte para um número de ponto flutuante antes de adicioná-lo à lista **x_data**. Da mesma forma, **float(s[1])** obtém o segundo valor da lista **s** (correspondente à coordenada **y**) e o converte para um número de ponto flutuante antes de adicioná-lo à lista **y_data**.

6. **x_data = np.array(x_data)** e **y_data = np.array(y_data)**

   Essas duas linhas convertem as listas **x_data** e **y_data** em arrays NumPy usando a função **np.array()**. Isso é útil porque os arrays NumPy são mais eficientes para realizar cálculos matemáticos e manipulações numéricas.

Em resumo, o código percorre cada linha do arquivo de dados representado pela lista **U**, verifica se a linha não está vazia e, em seguida, extrai os valores numéricos de coordenadas **x** e **y** de cada linha. Esses valores são armazenados nas listas **x_data** e **y_data**, respectivamente, e, posteriormente, essas listas são convertidas em arrays NumPy para uso em cálculos matemáticos e plotagem de gráficos. Essa abordagem é comum quando se trabalha com dados tabulares ou arquivos de dados em formato de texto.

```
degrees = range(1, 9)
pivoting = input('Do you want to use pivoting? (y/n): ').lower() == 'y'
for degree in degrees:
    coefficients = polynomial_approximation(x_data, y_data, degree, pivoting)
```

```
5
6     # Plot data points and approximation function
7     x_range = np.linspace(np.min(x_data), np.max(x_data), 100)
8     y_range = np.polyval(coefficients[::-1], x_range)
9     plt.plot(x_range, y_range, label=f'Degree {degree}', linestyle='--', alpha
      =0.7)
```

1. **degrees = range(1, 9)**

   Nesta linha, uma sequência de números inteiros é criada usando a função **range()**. A sequência vai de 1 até 8 (o número 9 não está incluso). Essa sequência representa os graus dos polinómios que serão usados para a aproximação.

2. **pivoting = input('Do you want to use pivoting? (y/n): ').lower() == 'y'**

   Nesta linha, o código solicita uma entrada do usuário, perguntando se deseja utilizar o pivoteamento. O **input()** é usado para receber a resposta do usuário, que pode ser "y"(sim) ou "n"(não). O método **lower()** é aplicado ao resultado para transformar qualquer letra maiúscula em minúscula, tornando a entrada do usuário não sensível a maiúsculas/minúsculas. O resultado é comparado com **'y'**, resultando em **True** se o usuário digitar "y"e **False** se digitar "n". O valor resultante é armazenado na variável **pivoting**.

3. **for degree in degrees:**

   Nesta linha, o código inicia um loop **for** que percorre cada valor na sequência de graus **degrees**. Ou seja, ele vai iterar de 1 até 8.

4. **coefficients = polynomial_approximation(x_data, y_data, degree, pivoting)**

   Dentro do loop, a função **polynomial_approximation** é chamada para obter os coeficientes do polinómio de aproximação. A função recebe os argumentos **x_data** e **y_data**, que são as listas de dados de coordenadas **x** e **y**, respetivamente. O argumento **degree** é o grau do polinómio de aproximação que será usado. O argumento **pivoting** é o valor booleano que indica se o pivoteamento deve ser utilizado ou não. Os coeficientes calculados são armazenados na variável **coefficients**.

5. **x_range = np.linspace(np.min(x_data), np.max(x_data), 100)**

   Nesta linha, a função **linspace()** do NumPy é utilizada para criar um array **x_range** com 100 pontos igualmente espaçados entre o valor mínimo e o valor máximo dos dados **x_data**. Isso é feito para criar uma faixa de valores **x** onde a função de aproximação será traçado.

6. **y_range = np.polyval(coefficients[::-1], x_range)**

   Nesta linha, a função **polyval()** do NumPy é utilizada para calcular os valores de **y** correspondentes à faixa de valores **x** (**x_range**) usando os coeficientes do polinómio de

aproximação (**coefficients**). O argumento **coefficients[::-1]** é usado para inverter a ordem dos coeficientes, pois o NumPy espera os coeficientes em ordem decrescente de potência do polinómio.

7. **plt.plot(x_range, y_range, label=f'Degree degree', linestyle='–', alpha=0.7)**

   Nesta linha, a função **plot()** do Matplotlib é utilizada para traçar a curva de aproximação do polinómio no gráfico. A faixa de valores **x** (**x_range**) é usada como o eixo **x**, e os valores de **y** correspondentes (**y_range**) são usados como o eixo **y**. O parâmetro **label** é usado para adicionar uma legenda à curva, indicando o grau do polinómio utilizado. O parâmetro **linestyle** é definido como **'–'**, especificando que a curva será traçado com linhas tracejadas. O parâmetro **alpha** define a transparência da curva, com o valor 0.7 indicando uma opacidade de 70

Em resumo, essa parte do código percorre diferentes graus de polinómios, calcula os coeficientes para cada grau usando a função **polynomial_approximation**, traça a curva de aproximação no gráfico para cada grau e exibe a legenda para indicar o grau correspondente em cada curva. Esse processo permite visualizar a aproximação polinomial dos dados para diferentes graus, ajudando a escolher o grau mais adequado para o ajuste dos dados.

```
1 plt.xlabel('X')
2 plt.ylabel('Y')
3 plt.legend()
4 plt.show()
```

Estas linhas usam o **Matplotlib** para tralçar os dados originais (pontos vermelhos) e as aproximações polinomiais (linhas tracejadas) no mesmo gráfico. Cada grau de aproximação é rotulado com o rótulo "**Degree degree**".

Ficha 11

11.1 Exercício 1

```
import numpy as np
import pandas as pd
import requests
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
from scipy.stats import t
```

Estas linhas importam as bibliotecas necessárias para o código. O **numpy** é utilizado para realizar cálculos numéricos eficientes, o **pandas** é usado para manipulação de dados em formato de tabela, o **requests** é utilizado para fazer solicitações HTTP para obter dados de um URL, o **matplotlib.pyplot** é utilizado para fazer gráficos e o **scipy.optimize.curve_fit** e **scipy.stats.t** são usados para realizar ajustes de curvas e cálculos estatísticos.

```
def linear_function(x, m, b):
    return m * x + b
```

Esta função **linear_function** é definida para representar uma função linear com dois parâmetros: m e b. Ela retorna o valor de **m * x + b**.

```
url = 'https://trixi.coimbra.lip.pt/data/fc/fc11/courteau99.dat'
response = requests.get(url)
lines = response.text.splitlines()
```

Estas linhas definem um URL de onde serão obtidos os dados. Em seguida, a biblioteca **requests** é usada para fazer uma solicitação **get** para esse URL e obter o conteúdo da resposta. O conteúdo é dividido em linhas e armazenado na variável **lines**.

```
data = np.genfromtxt(lines[2:], skip_header=1)
df = pd.DataFrame(data, columns=lines[1].split())
```

Estas linhas convertem os dados obtidos em formato de texto em um array numpy (**data**). As duas primeiras linhas são ignoradas (**lines[2:]**) e a terceira linha é usada para criar os nomes das colunas do DataFrame do pandas (**pd.DataFrame(data, columns=lines[1].split())**).

```
print("Available Variables:")
variable_names = lines[0].split()
for i, var in enumerate(variable_names):
    print(f"{i + 1}: {var}")
```

Estas linhas imprimem as variáveis disponíveis no conjunto de dados. A primeira linha é apenas uma mensagem de texto. A segunda linha divide a primeira linha do conteúdo (**lines[0]**) em palavras separadas e armazena-as na lista **variable_names**. Em seguida, um loop é usado para imprimir o número e o nome de cada variável na lista.

```
x_col_name = input("Enter the name of the x-variable: ")
y_col_name = input("Enter the name of the y-variable: ")
```

Estas linhas solicitam ao usuário que insira o nome das variáveis **x** e **y** que serão usadas no ajuste de curva. Os nomes inseridos são armazenados nas variáveis **x_col_name** e **y_col_name**.

```
x_col_index = variable_names.index(x_col_name)
y_col_index = variable_names.index(y_col_name)
```

Estas linhas encontram os índices das colunas correspondentes aos nomes inseridos pelo usuário. Os índices são armazenados nas variáveis **x_col_index** e **y_col_index**.

```
x = data[:, x_col_index]
y = data[:, y_col_index]
```

Estas linhas extraem as colunas selecionadas (**x** e **y**) dos dados com base nos índices das colunas.

```
mask = ~np.isnan(x) & ~np.isnan(y) & np.isfinite(x) & np.isfinite(y)
x = x[mask]
y = y[mask]
```

Estas linhas criam uma máscara booleana para remover linhas que contenham valores **NaN** (não é um número) ou infinitos tanto em **x** quanto em **y**. Os valores correspondentes às linhas válidas são armazenados novamente nas variáveis **x** e **y**.

```
popt, pcov = curve_fit(linear_function, x, y)
```

Esta linha realiza o ajuste de curva dos dados utilizando a função **curve_fit** da biblioteca **scipy.optimize**. Ela ajusta a função **linear_function** aos dados (**x**, **y**) e retorna os parâmetros ótimos (**popt**) e a matriz de covariância dos parâmetros (**pcov**).

```
m, b = popt
m_err = np.sqrt(pcov[0, 0])
```

Estas linhas extraem os valores dos parâmetros ótimos **m** e **b** do resultado do ajuste (**popt**). Além disso, o erro do parâmetro **m** é calculado como o desvio padrão da primeira diagonal da matriz de covariância (**pcov**).

```
residuals = y - linear_function(x, m, b)
ss_residuals = np.sum(residuals**2)
ss_total = np.sum((y - np.mean(y))**2)
r_squared = 1 - (ss_residuals / ss_total)
correlation_coefficient = np.sqrt(r_squared)
```

Estas linhas calculam o coeficiente de correlação entre os dados e a função ajustada. Primeiro, os resíduos são calculados como a diferença entre os valores observados (**y**) e os valores previstos pela função ajustada (**linear_function(x, m, b)**). Em seguida, a soma dos quadrados dos resíduos (**ss_residuals**), a soma dos quadrados totais (**ss_total**), o coeficiente de determinação (**r_squared**) e o coeficiente de correlação (**correlation_coefficient**) são calculados.

```
print(f"Correlation coefficient: {correlation_coefficient:.2f}")
```

Esta linha imprime o valor do coeficiente de correlação com duas casas decimais.

```
alpha = 0.05  # Significance level
n = len(x)  # Number of data points
t_critical = np.abs(t.ppf(alpha / 2, n - 2))  # T critical value
```

Estas linhas definem o nível de significância (**alpha**) como 0,05, o número de pontos de dados (**n**) como o comprimento da variável **x** e calculam o valor crítico de **T** (**t_critical**) com base no nível de significância e no número de graus de liberdade (**n - 2**).

```
if correlation_coefficient > t_critical * np.sqrt((1 - r_squared) / (n - 2)):
    print("The correlation is statistically significant at a 95% confidence
    level.")
else:
    print("The correlation is not statistically significant at a 95% confidence
    level.")
```

Estas linhas testam a hipótese de que a correlação é estatisticamente significativa a um nível de confiança de 95%. Se o coeficiente de correlação for maior que o valor crítico de **t** multiplicado pelo erro padrão da correlação, a correlação é considerada estatisticamente significativa.

```
m_err *= t_critical * np.sqrt((ss_residuals / (n - 2)) / np.sum((x - np.mean(x))
    **2))
```

Esta linha calcula o erro do fator de inclinação (**m_err**) com um nível de confiança de 95%, ajustando-o pelo valor crítico de **t** e pelo desvio padrão dos resíduos.

```
print(f"Error of the slope factor (with 95% confidence level): {m_err:.2f}")
```

Esta linha imprime o erro do fator de inclinação.

```
plt.scatter(x, y, label='Data')
plt.plot(x, linear_function(x, m, b), color='red', label='Fitted Line')
plt.xlabel(x_col_name)
plt.ylabel(y_col_name)
plt.legend()
```

Estas linhas criam um gráfico de dispersão dos dados (pontos) e traça a linha ajustada (linha vermelha). Os eixos **x** e **y** são rotulados com os nomes das variáveis fornecidas pelo usuário.

```
plt.savefig('figure.png', dpi=600)
```

Esta linha salva o gráfico como um arquivo de imagem chamado "**figure.png**"com uma resolução de 600 dpi.

```
output_data = np.column_stack((x, y))
np.savetxt('data.dat', output_data, delimiter='\t', header=f"{x_col_name}\t{
    y_col_name}", comments='', fmt='%.8f')
```

Estas linhas combinam as colunas de **x** e **y** em uma única matriz **output_data** e salvam os valores **x** e **y** em um arquivo de dados chamado "**data.dat**". O arquivo é salvo com delimitador de tabulação e os nomes das variáveis são incluídos no cabeçalho. Os valores numéricos são formatados com 8 casas decimais.

11.1.1 O que se podia ter melhorado?

Salienta-se que o código fornecido assume que as colunas específicas das variáveis "**vlg**"(7) e "**L B**"(51) são as corretas para a análise da correlação *Tully-Fisher*. Se essas colunas não corresponderem aos dados corretos para essa correlação específica, será necessário modificar o código para usar as colunas corretas.

Existem algumas melhorias que podem ser consideradas:

1. Comentários: Adicionar comentários ao código pode ajudar a entender melhor as diferentes partes do programa e a lógica por trás delas. Comentários claros e explicativos tornam o código mais legível e facilitam a manutenção futura.

2. Funções/modularidade: Pode ser útil dividir o código em funções separadas para realizar tarefas específicas, como leitura dos dados, ajuste de curva, cálculo do coeficiente de correlação, etc. Isso melhora a modularidade do código e facilita a reutilização e manutenção.

3. Tratamento de erros: O código assume que a URL de onde os dados serão baixados está sempre acessível e que os dados estão corretamente formatados. É uma boa prática adicionar tratamento de erros para lidar com situações em que a URL não esteja disponível ou os dados não estejam no formato esperado.

4. Manipulação de exceções: É possível adicionar manipulação de exceções para capturar erros durante a execução do programa, como erros de entrada do usuário ou problemas de acesso à Internet. Isso garantirá que o programa seja robusto e não falhe abruptamente em caso de erros.

O código em baixo implementa algumas dessas melhorias:

```
import numpy as np
import pandas as pd
import requests
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
from scipy.stats import t

# Define a fun   o para ajuste linear
def linear_function(x, m, b):
    return m * x + b

# Fun   o para obter os dados do URL
def get_data(url):
    response = requests.get(url)
    lines = response.text.splitlines()
    return lines

# Fun   o para ler os dados e criar o DataFrame
def read_data(lines):
    data = np.genfromtxt(lines[2:], skip_header=1)
    df = pd.DataFrame(data, columns=lines[1].split())
    return df

# Fun   o para exibir as vari veis dispon veis
def display_variables(variable_names):
    print("Available Variables:")
    for i, var in enumerate(variable_names):
        print(f"{i + 1}: {var}")

# Fun   o para realizar o ajuste de curva e retornar os par metros
def perform_curve_fit(x, y):
    popt, pcov = curve_fit(linear_function, x, y)
    return popt, pcov
```

```
34
35 # Fun  o para calcular o coeficiente de correla  o
36 def calculate_correlation_coefficient(x, y, popt):
37     residuals = y - linear_function(x, *popt)
38     ss_residuals = np.sum(residuals**2)
39     ss_total = np.sum((y - np.mean(y))**2)
40     r_squared = 1 - (ss_residuals / ss_total)
41     correlation_coefficient = np.sqrt(r_squared)
42     return correlation_coefficient, r_squared, ss_residuals
43
44 # Fun  o para testar a hip tese de correla  o
45 def test_correlation(correlation_coefficient, r_squared, n, alpha):
46     t_critical = np.abs(t.ppf(alpha / 2, n - 2))
47     threshold = t_critical * np.sqrt((1 - r_squared) / (n - 2))
48     if correlation_coefficient > threshold:
49         return True
50     else:
51         return False
52
53 # Fun  o para calcular o erro do fator de inclina  o
54 def calculate_slope_error(m_err, t_critical, ss_residuals, n, x):
55     m_err *= t_critical * np.sqrt((ss_residuals / (n - 2)) / np.sum((x - np.mean
    (x))**2))
56     return m_err
57
58 # Fun  o para plotar os dados e a linha ajustada
59 def plot_graph(x, y, x_col_name, y_col_name, popt):
60     plt.scatter(x, y, label='Data')
61     plt.plot(x, linear_function(x, *popt), color='red', label='Fitted Line')
62     plt.xlabel(x_col_name)
63     plt.ylabel(y_col_name)
64     plt.legend()
65
66 # Fun  o para salvar a figura do gr fico
67 def save_figure():
68     plt.savefig('figure.png', dpi=600)
69
70 # Fun  o para salvar os valores x e y em um arquivo de dados
71 def save_data(x, y, x_col_name, y_col_name):
72     output_data = np.column_stack((x, y))
73     np.savetxt('data.dat', output_data, delimiter='\t', header=f"{x_col_name}\t{
    y_col_name}", comments='', fmt='%.8f')
74
75 # URL do arquivo de dados
76 url = 'https://trixi.coimbra.lip.pt/data/fc/fc11/courteau99.dat'
77
78 # Obter os dados do URL
79 lines = get_data(url)
80
81 # Ler os dados e criar o DataFrame
```

```
82 df = read_data(lines)
83
84 # Exibir as vari veis dispon veis
85 variable_names = lines[0].split()
86 display_variables(variable_names)
87
88 # Obter as vari veis escolhidas pelo usu rio
89 x_col_name = input("Enter the name of the x-variable: ")
90 y_col_name = input("Enter the name of the y-variable: ")
91
92 # Encontrar os  ndices  das colunas das vari veis escolhidas
93 x_col_index = variable_names.index(x_col_name)
94 y_col_index = variable_names.index(y_col_name)
95
96 # Extrair as colunas selecionadas
97 x = df.iloc[:, x_col_index].values
98 y = df.iloc[:, y_col_index].values
99
100 # Remover linhas com valores inv lidos
101 mask = ~np.isnan(x) & ~np.isnan(y) & np.isfinite(x) & np.isfinite(y)
102 x = x[mask]
103 y = y[mask]
104
105 # Realizar o ajuste de curva
106 popt, pcov = perform_curve_fit(x, y)
107
108 # Extrair os par metros do ajuste
109 m, b = popt
110 m_err = np.sqrt(pcov[0, 0])
111
112 # Calcular o coeficiente de correla  o
113 correlation_coefficient, r_squared, ss_residuals =
    calculate_correlation_coefficient(x, y, popt)
114 print(f"Correlation coefficient: {correlation_coefficient:.2f}")
115
116 # Testar a hip tese de correla  o (com n vel de confian a de 95%)
117 alpha = 0.05
118 n = len(x)
119 if test_correlation(correlation_coefficient, r_squared, n, alpha):
120     print("The correlation is statistically significant at a 95% confidence
    level.")
121 else:
122     print("The correlation is not statistically significant at a 95% confidence
    level.")
123
124 # Calcular o erro do fator de inclina  o (com n vel de confian a de 95%)
125 t_critical = np.abs(t.ppf(alpha / 2, n - 2))
126 m_err = calculate_slope_error(m_err, t_critical, ss_residuals, n, x)
127 print(f"Error of the slope factor (with 95% confidence level): {m_err:.2f}")
128
```

```
129 # Plotar os dados e a linha ajustada
130 plot_graph(x, y, x_col_name, y_col_name, popt)
131
132 # Salvar a figura do gr fico
133 save_figure()
134
135 # Salvar os valores x e y em um arquivo de dados
136 save_data(x, y, x_col_name, y_col_name)
```

Ficha 12

12.1 Código realizado

```
# Importar a classe ErrNum
from math import sqrt
```

Essa linha importa a função **sqrt** da biblioteca **math**, que é usada para calcular a raiz quadrada em algumas operações dentro da classe **ErrNum**.

```
class ErrNum:
```

Aqui, a classe **ErrNum** é definida.

```
    def __init__(self, value, error):
        self.value = float(value)
        self.error = float(error)
```

O método especial **__init__** é o construtor da classe e é chamado quando um novo objeto **ErrNum** é criado. Ele inicializa os atributos **value** (valor do número) e **error** (incerteza) com os valores passados como argumentos para o construtor. Os valores são convertidos para tipo **float** para garantir que sejam tratados como números de ponto flutuante.

```
    def __repr__(self):
        return f"{self.value}({self.error})"
```

O método especial **__repr__** retorna uma representação em string do objeto **ErrNum**. A representação é da forma *"valor(incerteza)"*.

```
    def __add__(self, other):
        if isinstance(other, ErrNum):
            value = self.value + other.value
            error = sqrt(self.error**2 + other.error**2)
            return ErrNum(value, error)
```

```
6         elif isinstance(other, (int, float)):
7             value = self.value + other
8             return ErrNum(value, self.error)
9         else:
10            raise TypeError("Unsupported operand types")
```

O método especial **__add__** permite a adição de objetos **ErrNum**. Se o objeto passado como argumento for um **ErrNum**, a adição é realizada levando em conta as incertezas e retorna um novo objeto **ErrNum** resultante da operação. Se o argumento for um número (int ou float), a adição é realizada apenas ao valor do **ErrNum** e a incerteza não muda.

```
1     def __sub__(self, other):
2         if isinstance(other, ErrNum):
3             value = self.value - other.value
4             error = sqrt(self.error**2 + other.error**2)
5             return ErrNum(value, error)
6         elif isinstance(other, (int, float)):
7             value = self.value - other
8             return ErrNum(value, self.error)
9         else:
10            raise TypeError("Unsupported operand types")
```

O método especial **__sub__** permite a subtração de objetos **ErrNum**. Se o objeto passado como argumento for um **ErrNum**, a subtração é realizada levando em conta as incertezas e retorna um novo objeto **ErrNum** resultante da operação. Se o argumento for um número (int ou float), a subtração é realizada apenas ao valor do **ErrNum** e a incerteza não muda.

```
1     def __mul__(self, other):
2         if isinstance(other, ErrNum):
3             value = self.value * other.value
4             error = sqrt(
5                 (other.value * self.error)**2 + (self.value * other.error)**2
6             )
7             return ErrNum(value, error)
8         elif isinstance(other, (int, float)):
9             value = self.value * other
10            return ErrNum(value, abs(self.error * other))
11        else:
12            raise TypeError("Unsupported operand types")
```

O método especial **__mul__** permite a multiplicação de objetos **ErrNum**. Se o objeto passado como argumento for um **ErrNum**, a multiplicação é realizada levando em conta as incertezas e retorna um novo objeto **ErrNum** resultante da operação. Se o argumento for um número (int ou float), a multiplicação é realizada apenas ao valor do **ErrNum** e a incerteza é ajustada de acordo com a propagação do erro.

```
    def __truediv__(self, other):
        if isinstance(other, ErrNum):
            if other.value != 0:
                value = self.value / other.value
                error = sqrt(
                    (self.error / other.value)**2
                    + (self.value * other.error / other.value**2)**2
                )
                return ErrNum(value, error)
            else:
                raise ZeroDivisionError("Division by zero")
        elif isinstance(other, (int, float)):
            if other != 0:
                value = self.value / other
                return ErrNum(value, abs(self.error / other))
            else:
                raise ZeroDivisionError("Division by zero")
        else:
            raise TypeError("Unsupported operand types")
```

O método especial **__truediv__** permite a divisão de objetos **ErrNum**. Se o objeto passado como argumento for um **ErrNum**, a divisão é realizada levando em conta as incertezas e retorna um novo objeto **ErrNum** resultante da operação. Se o argumento for um número (int ou float), a divisão é realizada apenas ao valor do **ErrNum** e a incerteza é ajustada de acordo com a propagação do erro.

```
    def __pow__(self, power):
        if isinstance(power, (int, float)):
            value = self.value**power
            error = abs(power * self.value**(power - 1) * self.error)
            return ErrNum(value, error)
        else:
            raise TypeError("Unsupported operand types")
```

O método especial **__pow__** permite elevar um objeto **ErrNum** a uma potência. Se o argumento for um número (int ou float), a potência é aplicada ao valor do **ErrNum**, e a incerteza é ajustada de acordo com a propagação do erro.

```
    def __eq__(self, other):
        if isinstance(other, ErrNum):
            return abs(self.value - other.value) <= sqrt(
            self.error**2 + other.error**2
        )
        elif isinstance(other, (int, float)):
            return abs(self.value - other) <= self.error
        else:
            return False
```

O método especial __**eq**__ permite comparar se dois objetos **ErrNum** são iguais. Se o objeto passado como argumento for um **ErrNum**, a comparação é feita levando em conta as incertezas. Caso o argumento seja um número (int ou float), a comparação é feita apenas com o valor do **ErrNum**.

Ficha 13

13.1 Apreciação

Este código têm vários pontos que podem e devem ser melhorados, devido a questões de tempo não o pude fazer, pelo que aconselho a consultares o site da universidade de Harvard pois têm um curso gratuito em que usam exatamente este exemplo e pode-te ajudar.

13.2 Exercício 1

```
1 import json
2 import yaml
3
4 maxpos = 1000
5 nr_games = [10, 100, 1000, 10000]
```

Nas primeiras linhas, são importados os módulos **json** e **yaml**. Em seguida, duas variáveis são definidas: **maxpos** com o valor 1000, que representa o número de peças iniciasi, e **nr_games** é uma lista com diferentes números de jogos a serem executados.

```
1 Stat = {}
2 for i in range(1, maxpos + 1):
3     Stat[i] = {}
4     for j in range(1, min(i, 3) + 1):
5         Stat[i][j] = 0
```

A variável **Stat** é inicializada como um dicionário vazio. Em seguida, é feito um loop de 1 até **maxpos**, onde para cada posição **i**, é criado um dicionário vazio **Stat[i]**. Dentro deste loop, outro loop é executado de 1 até o mínimo entre **i** e 3, criando uma entrada **Stat[i][j]** com valor 0 para cada movimento possível na posição **i**.

```
1 for game_count in nr_games:
2     for g in range(game_count):
3         moves = {}
```

```
        moves[1] = {}
        moves[2] = {}
        pos = maxpos
        player = 0
        while pos:
            player = 2 if player == 1 else 1
            move = max(Stat[pos], key=Stat[pos].get)
            moves[player][pos] = move
            pos -= move
        for pos in moves[player]:
            Stat[pos][moves[player][pos]] += 1
        player = 2 if player == 1 else 1
        for pos in moves[player]:
            Stat[pos][moves[player][pos]] -= 1
```

Aqui começa o loop principal, que itera sobre cada número de jogos em **nr_games**. Dentro desse loop, outro loop é executado **game_count** vezes.

Para cada jogo, é criado um dicionário **moves** para armazenar os movimentos feitos pelos jogadores. São criados sub-dicionários **moves[1]** e **moves[2]** para representar os movimentos feitos pelos jogadores 1 e 2, respetivamente.

A variável **pos** é definida como **maxpos**, representando a posição atual do jogo, e a variável **player** é inicializada como 0, indicando o jogador atual.

Dentro do loop **while pos**, enquanto ainda houver peças na posição atual, o jogador é alternado entre 1 e 2. O movimento escolhido pelo jogador é determinado selecionando o movimento com maior valor em **Stat[pos]**, usando **max(Stat[pos], key=Stat[pos].get)**. Esse movimento é armazenado no dicionário **moves** correspondente ao jogador atual na posição atual **pos**, usando **moves[player][pos] = move**. Em seguida, a posição é atualizada subtraindo o valor do movimento escolhido: **pos -= move**.

Após o loop **while pos**, são feitas atualizações em **Stat** com base nos movimentos feitos pelos jogadores. O jogador atual é alternado novamente, e para cada posição em **moves[player]**, é aumentado **Stat[pos][moves[player][pos]]** por 1. Em seguida, o jogador é alternado novamente, e para cada posição em **moves[player]**, é diminuído **Stat[pos][moves[player][pos]]** por 1.

```
def evaluate_moves(Stat):
    best_moves = {}
    for i in range(1, maxpos + 1):
        best_move = max(Stat[i], key=Stat[i].get)
        best_moves[i] = best_move
    return best_moves

def detect_learning_limit(Stat):
    for i in range(maxpos, 0, -1):
        correct_move = i % 4
        if correct_move == 0:
            correct_move = 3
```

```
13          best_move = best_moves[i]
14          if best_move != correct_move:
15              return i
16      return 0
```

Aqui estão as definições das funções **evaluate_moves** e **detect_learning_limit**. A função **evaluate_moves** recebe **Stat** como entrada e retorna um dicionário **best_moves** que contém os melhores movimentos para cada posição com base nas estatísticas acumuladas em **Stat**.

A função **detect_learning_limit** verifica até que ponto o algoritmo aprendeu corretamente as melhores jogadas. Ela percorre as posições de **maxpos** até 1 e compara o melhor movimento registado em **best_moves** com o movimento correto (determinado pelo jogo *NIM*). Se houver uma discrepância, a posição é retornada como limite de aprendizado. Caso contrário, 0 é retornado.

```
1  best_moves = evaluate_moves(Stat)
2  learning_limit = detect_learning_limit(Stat)
3
4  print(f"Game Count: {game_count}\tLearning Limit: {learning_limit}")
5
6  # Save Stat in json and yaml file:
7  with open(f'nim_{game_count}_games.json', 'w') as f:
8      json.dump(Stat, f, sort_keys=True, indent=4, separators=(',', ': '))
9
10 with open(f'nim_{game_count}_games.yaml', 'w') as f:
11     f.write(yaml.dump(Stat))
```

Após cada iteração do loop principal, as melhores jogadas são avaliadas usando a função **evaluate_moves** e o limite de aprendizado é detetado usando a função **detect_learning_limit**. Em seguida, são impressos o número de jogos executados e o limite de aprendizado.

Os dados de **Stat** são salvos em arquivos **JSON** e **YAML** para posterior análise. Os arquivos são nomeados com base no número de jogos executados.

13.2.1 Explicação teórica do algoritmo de Q-Learning

O código implementa um algoritmo de aprendizagem por reforço simples chamado Q-Learning. O Q-Learning é um método de aprendizagem por reforço baseado em valores de Q (valor de qualidade) que associa ações a estados específicos do ambiente. O objetivo é encontrar a política de ação ótima que maximize a recompensa acumulativa ao longo do tempo.

No contexto do jogo NIM, o ambiente consiste nas posições das peças e as ações são as escolhas dos jogadores de remover uma certa quantidade de peças. O objetivo final é ganhar o jogo, ou seja, remover a última peça.

Aqui está uma descrição teórica dos principais conceitos e métodos usados no código:

- **Valor de Q**: Em Q-Learning, o valor de Q é uma medida do valor esperado de uma ação em um determinado estado. No jogo NIM, cada posição e ação possuem um valor de Q

associado, que representa a qualidade da ação naquele estado. O objetivo da aprendizagem é atualizar e aprimorar os valores de Q para escolher as melhores ações.

- **Exploração e explotação**: Durante a aprendizagem, o agente precisa equilibrar a exploração do ambiente (tentar novas ações para descobrir informações) e a explotação do conhecimento existente (usar ações com valores de Q mais altos). A exploração permite ao agente descobrir melhores ações, enquanto a explotação usa o conhecimento existente para tomar decisões mais vantajosas.

- **Atualização dos valores de Q**: Após cada ação tomada, os valores de Q são atualizados com base no resultado dessa ação. O agente aprende com o feedback recebido em termos de recompensa. Os valores de Q são atualizados usando a fórmula:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \cdot (r + \gamma \cdot \max Q(s', a') - Q(s, a))$$

  onde:

  - $Q(s, a)$ é o valor de Q atual para o estado s e a ação a.
  - α (alfa) é a taxa de aprendizagem que controla a rapidez com que o agente atualiza seus valores de Q.
  - r é a recompensa recebida após tomar a ação.
  - γ (gama) é o fator de desconto que determina a importância das recompensas futuras.

- **Política ótima**: A política ótima é uma estratégia de ação que maximiza a recompensa acumulativa ao longo do tempo. No contexto do jogo NIM, a política ótima é aquela que leva o agente a tomar as ações corretas para ganhar o jogo. O objetivo da aprendizagem é encontrar essa política ótima atualizando gradualmente os valores de Q.

- **Limite de aprendizagem**: O limite de aprendizagem é o ponto em que o agente aprendeu a tomar as ações corretas para ganhar o jogo até uma determinada posição. No código, o limite de aprendizagem é determinado verificando se o movimento escolhido pelo agente coincide com o movimento correto para cada posição. Se houver uma discrepância, isso indica que o agente ainda não aprendeu a jogar corretamente até essa posição.

Estes são os conceitos-chave envolvidos no algoritmo de aprendizagem por reforço usado no código do jogo NIM. O Q-Learning é um método fundamental de aprendizagem por reforço, e existem muitas extensões e variações mais avançadas que podem ser exploradas para melhorar ainda mais o desempenho do agente em jogos e outros cenários.

13.3 Exercício 2

De modo a implementar o solicitado foi adicionada a lógica necessária para obter todos os movimentos com a pontuação máxima na posição atual usando uma list comprehension:

```
best_moves = [move for move in Stat[pos] if Stat[pos][move] == max(Stat[pos].
    values())]
```

Em seguida, usou-s a função **random.choice()** para selecionar aleatoriamente um dos movimentos da lista **best_moves**.

```
move = random.choice(best_moves)
```

Estas modificações garantiram que, quando houver múltiplos movimentos com a mesma pontuação mais alta, o programa escolherá aleatoriamente um deles em vez de sempre selecionar o primeiro. Isso ajudará a reduzir qualquer viés na escolha dos movimentos.

Relembra-se de que a eficiência do processo de aprendizagem pode variar dependendo dos parâmetros e da natureza do problema. Pode ser necessário ajustar os parâmetros, como o número de jogos (**nr_games**), para obter melhores resultados. A experiência e a análise dos resultados são importantes para entender o desempenho do algoritmo em diferentes cenários.

13.4 Exercício 3

Neste código, há algumas diferenças em relação ao código anterior que simulava o jogo NIM. Destaca-se as principais diferenças:

1. Configuração inicial das fileiras:
    - Em vez de ter uma única lista de peças iniciais, agora temos a lista **rows** que contém o número de peças em cada fileira. Por exemplo, **[1, 3, 5, 7]** representa que a primeira fileira possui 1 peça, a segunda possui 3 peças, a terceira possui 5 peças e a quarta possui 7 peças.

```
rows = [1, 3, 5, 7]
```

2. Simulação do jogo:
    - Em vez de trabalhar com uma única variável **pos**, agora temos uma lista **rows** que representa o estado atual de cada fileira.
    - Durante cada jogo simulado, o loop **while any(rows)** é usado para continuar o jogo enquanto houver pelo menos uma fileira com peças restantes.
    - O jogador é alternado entre 0 e 1 usando **player = 1 - player**.
    - A escolha do movimento aleatório é feita com base nas fileiras que ainda possuem peças. A função **random.choice()** é usada para selecionar aleatoriamente uma fileira e, em seguida, o movimento é escolhido aleatoriamente entre os melhores movimentos disponíveis para aquela fileira.

```
for g in range(game_count):
    moves = {}
    for i in range(len(rows)):
```

```
        moves[i] = {}
    player = 0
    while any(rows):
        player = 1 - player
        non_empty_rows = [i for i in range(len(rows)) if rows[i] > 0]
        if not non_empty_rows:
            break
        row = random.choice(non_empty_rows)
        best_moves = [move for move in Stat[row] if Stat[row][move] == max(
    Stat[row].values())]
        move = random.choice(best_moves)
        moves[row][rows[row]] = move
        rows[row] -= move
    for i in range(len(rows)):
        for row in moves[i]:
            Stat[i][moves[i][row]] += 1 if player == i else -1
```

3. Função `detect_winning_strategy()`:

   - Esta função é introduzida para determinar se o jogo possui uma estratégia vencedora.
   - Ela calcula o XOR (ou exclusivo) dos valores de todas as fileiras usando um loop **for** e verifica se o resultado é diferente de zero.
   - Se o resultado for diferente de zero, isso indica que existe uma estratégia vencedora.

```
def detect_winning_strategy(rows):
    xor_sum = 0
    for row in rows:
        xor_sum ^= row
    return xor_sum != 0
```

4. Impressão dos resultados:

   - O código imprime o número de jogos simulados e, em seguida, verifica se o jogo possui uma estratégia vencedora com base na função **detect_winning_strategy()**.
   - Se o jogo tiver uma estratégia vencedora, ele imprime os movimentos vencedores para cada fileira.
   - A formatação da impressão também é ajustada para se adequar aos requisitos.

```
best_moves = evaluate_moves(Stat)
has_winning_strategy = detect_winning_strategy(rows)

print(f"Game Count: {game_count}")
if has_winning_strategy:
    print("O jogo possui uma estrategia vencedora!")
    print("Movimentos Vencedores:")
    for i in range(len(rows)):
        print(f"Fileira {i+1}: Retire {best_moves[i]} pecas")
else:
```

```
11     print("O jogo nao possui uma estrategia vencedora.")
12 print()
```

Em resumo, estas são as principais alterações no código para lidar com o jogo NIM com quatro fileiras. A configuração inicial, a simulação do jogo, a determinação de uma estratégia vencedora e a impressão dos resultados são adaptados para as novas condições do jogo.