IPS Instituto Politécnico de Setúbal Escola Superior de Tecnologia de Setúbal	COMPUTAÇÃO PARALELA E DISTRIBUÍDA 2022/2023
	Laboratório 01: Vetores e Matrizes em Python

Objetivos do laboratório

Pretende-se que o estudante:

1. Utilize a biblioteca **NumPy para** criar vetores e matrizes;
2. Entenda como obter estatísticas sobre o tempo de execução de um ou vários comandos do código;
3. Perceba como multiplicar matrizes de maneira tradicional, mas eficiente.

Pré-requisitos:

- Python com versão acima de 3.5
- biblioteca **NumPy** (para instalar utilizar comando **pip install numpy**)

Questões

1. Escreva o programa apresentado.

```
import timeit
import numpy as np
rnd = np.random
n = 10000
vetor = rnd.randint(1000, size=n)
melhorTempo = 9999999   # Um valor muito grande
for i in range(3):  # Efetua 3 medições de tempo e mostra a melhor
    print(f"Medição {i+1} para {n} elementos...")
    starttime = timeit.default_timer()
    soma = 0.0
    for j in range(n):
        soma += vetor[j]
    tempo = timeit.default_timer() - starttime
    print(f"\t {tempo} segundos")
    if tempo < melhorTempo:
        melhorTempo = tempo
print(f"Melhor tempo da soma para {n} elementos: {melhorTempo} segundos")
#print(vetor.dtype)
print(f"Largura de banda da RAM: {n*4/melhorTempo/(1024*1024):.2f} Mega Bytes
por segundo")
```

Perguntas:

- Identifique as instruções onde é calculado o tempo da soma dos elementos do vetor
- Interprete a expressão de cálculo da largura de banda (`n*4/melhorTempo/(1024*1024)`)

2. Altere o programa para automaticamente fazer as medições para tamanhos do vetor iguais a mil, dez mil, cem mil, um milhão, dez milhões e vinte milhões (por exemplo, pode colocar estes valores numa lista e usar um **for** fazendo iteração nessa lista). É também necessário que o seu programa vá acrescentando a uma lista as diversas medições de largura de banda e depois apresente a sua média. Por exemplo, se usar uma lista **bwRAM** para guardar as medições, a média será: **sum(bwRAM)/len(bwRAM)**

3. O código abaixo apresenta duas implementações da multiplicação de uma matriz por um vetor. São apresentados os primeiros cinco valores do resultado para permitir confirmar que ambas as implementações produzem o mesmo resultado.

```
import timeit
import numpy as np

rnd = np.random
# n define as dimensões dos vetores e da matriz

n = 3000
vetor = rnd.uniform(0,1000,size=n)
matriz = rnd.uniform(0,1000,size=(n,n))
resultado = np.zeros(n)

# primeira implementação
starttime = timeit.default_timer()
for i in range(len(matriz)):  # número de linhas da matriz
    for j in range(len(vetor)): # número de elementos no vetor
        resultado[i] += matriz[i,j]*vetor[i]
print(f"Tempo 1: {timeit.default_timer() - starttime}")
print(f"Primeiros elementos do vetor resultado: {resultado[0:5]}")

# segunda implementação
starttime = timeit.default_timer()
for i in range(len(matriz)):  # número de linhas da matriz
    temp = 0
    for j in range(len(vetor)): # número de elementos no vetor
        temp += matriz[i,j]*vetor[i]
    resultado[i] = temp
print(f"Tempo 2: {timeit.default_timer() - starttime}")
print(f"Primeiros elementos do vetor resultado: {resultado[0:5]}")
```

Pergunta:

- Analise os resultados observados quando executa o programa e explique a diferença de performance.
- Experimente correr o programa variando o n entre 500 e 5000. Relacione o valor do n com a variação da performance dos dois métodos

4. Multiplicação de matrizes

Considerando a forma mais eficiente utilizada no nível anterior, escreva um programa para multiplicar duas matrizes quadradas, com 500 linhas e 500 colunas, contendo valores reais pertencentes ao intervalo [0,500[gerados aleatoriamente. O programa deverá apresentar o tempo utilizado para na multiplicação das matrizes.

5. Multiplicação de matriz por matriz usando a biblioteca NumPy

Desenvolva um código que faça o mesmo que no número 4 (multiplicação de matrizes), mas utilizando a função ***matmul*** da biblioteca **NumPy.**

ANEXO I

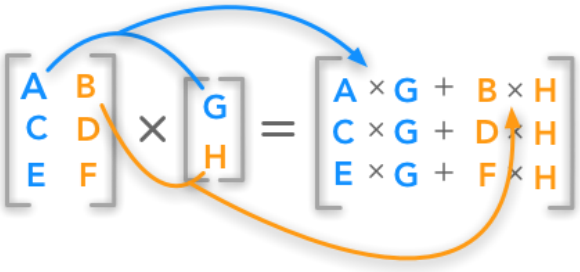
Referências

- https://numpy.org/doc/stable/reference/arrays.ndarray.html#arrays-ndarray

- https://ncert.nic.in/textbook/pdf/keip106.pdf

Conceitos

Multiplicação Vetores x matrizes



$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

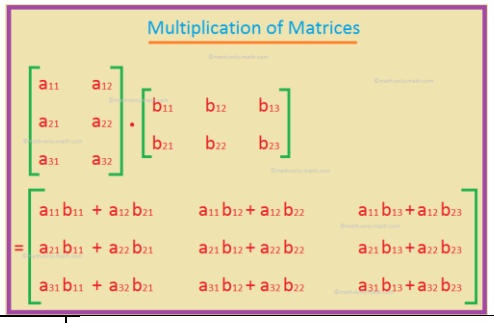
and:

$$b = \begin{bmatrix} 2 \\ 4 \end{bmatrix}$$

$$\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \\ A_{3,1} & A_{3,2} \end{bmatrix} \times \begin{bmatrix} B_{1,1} \\ B_{2,1} \end{bmatrix} = \begin{bmatrix} A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\ A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\ A_{3,1}B_{1,1} + A_{3,2}B_{2,1} \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \times \begin{bmatrix} 2 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 \times 2 + 2 \times 4 \\ 3 \times 2 + 4 \times 4 \\ 5 \times 2 + 6 \times 4 \end{bmatrix} = \begin{bmatrix} 10 \\ 22 \\ 34 \end{bmatrix}$$

Multiplicação matrizes



$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}$$

and:

$$B = \begin{bmatrix} 2 & 7 \\ 1 & 2 \\ 3 & 6 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix} \begin{bmatrix} 2 & 7 \\ 1 & 2 \\ 3 & 6 \end{bmatrix} =$$

$$\begin{bmatrix} 2\times1+1\times2+3\times3 & 7\times1+2\times2+6\times3 \\ 2\times4+1\times5+3\times6 & 7\times4+2\times5+6\times6 \\ 2\times7+1\times8+3\times9 & 7\times7+2\times8+6\times9 \\ 2\times10+1\times11+3\times12 & 7\times10+2\times11+6\times12 \end{bmatrix}$$

$$= \begin{bmatrix} 13 & 29 \\ 31 & 74 \\ 49 & 119 \\ 67 & 164 \end{bmatrix}$$

NumPy

NumPy (**Numerical Python**) é uma biblioteca Python de código aberto usada em quase todos os campos da ciência e engenharia. É o padrão universal para trabalhar com dados numéricos em Python e está no centro dos ecossistemas científicos Python e PyData. Os utilizadores do NumPy incluem desde programadores iniciantes a pesquisadores experientes em pesquisa e desenvolvimento científico e industrial de última geração. A API NumPy é usada extensivamente na maioria dos pacotes de ciência de dados e científicos em Python.

A biblioteca NumPy contém estruturas de dados para vetores multidimensionais e matrizes. Fornece por exemplo **ndarray**, um objeto vetorial homogêneo n-dimensional, com métodos para trabalhar com eficiência. O NumPy pode ser usado para realizar uma ampla variedade de operações matemáticas em matrizes. Adiciona estruturas de dados poderosas ao Python que garantem cálculos eficientes com vetores e matrizes e fornece uma enorme biblioteca de funções matemáticas de alto nível que operam nesses vetores e matrizes.

A instalação da biblioteca NumPy é feita com o comando:

```
pip install numpy
```

Qual é a diferença entre uma lista Python e uma matriz NumPy?

O NumPy oferece uma gama enorme de maneiras rápidas e eficientes de criar matrizes e manipular dados numéricos dentro delas. Embora uma lista Python possa conter diferentes tipos de dados em uma única lista, todos os elementos em uma matriz NumPy devem ser homogêneos. As operações matemáticas que devem ser realizadas em matrizes seriam extremamente ineficientes se as matrizes não fossem homogêneas.

Por que usar o NumPy?

Os vetor NumPy são mais rápidos e compactos do que as listas Python. Um vetor consome menos memória e é conveniente de usar. O NumPy usa muito menos memória para armazenar dados e fornece um mecanismo de especificação dos tipos de dados. Isso permite que o código seja otimizado ainda mais.

Criação de vetores e matrizes

O código abaixo cria um vetor *v* e uma matriz *m*:

```
import numpy as np
v = np.array([1, 2, 3, 4, 5, 6])
m = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
```

O acesso aos elementos individuais de vetores e matrizes é feito usando-se índices, da mesma forma que se faz com listas. As dimensões de uma matriz são denominadas **eixos** no NumPy: no caso da matriz m, ela tem dois eixos. O primeiro tem um comprimento (*length*) de 3 ("linhas") e o segundo eixo tem um comprimento de 4 ("colunas").

Algumas operações úteis com matrizes são ilustradas no código abaixo:

```
print(type(m))  # Mostra <class 'numpy.ndarray'>
print(m.shape)  # Mostra (3,4)
print(m.dtype)  # Mostra int32
m1 = m[:,1]
print(m1)       # Mostra [ 2  6 10]
m1[0] = 8
print(m1)       # Mostra [ 8  6 10]
print(m)        # Mostra [[ 1  8  3  4]
                #         [ 5  6  7  8]
                #         [ 9 10 11 12]]
```

Para trabalhar com geração de números pseudo-aleatórios podemos utilizar comandos semelhantes aos mostrados abaixo:

```
from numpy.random import default_rng
rng = default_rng()
vals = rng.standard_normal(3)  # Cria um vetor com 3 valores p-aleatórios
                               # com média 0 e desvio padrão igual a 1
print(vals)                    # Mostra o vetor gerado
rnd = np.random
m2 = rnd.rand(2,3)*100         # Cria uma matriz com 2 linhas e 3 colunas
                               # e valores reais p-aleatórios com
                               # distribuição uniforme no intervalo [0,100[
print(m2)
m3 = rnd.randint(0,100,size=(2,3)) # Cria uma matriz com 2 linhas e 3 colunas
                                   # com valores p-aleatórios inteiros no
                                   # intervalo [0,100[
print(m3)
m4 = rnd.uniform(0,100,size=(2,3)) # O mesmo que o anterior
print(m4)
```

Se quisermos medir o tempo de uma ou de várias operações, podemos importar o módulo ***timeit***:

```
import timeit
starttime = timeit.default_timer()
m3 = rnd.randint(0,100,size=(8000,8000))
print("Tempo de criação da matriz m3: ", timeit.default_timer() - starttime)
```