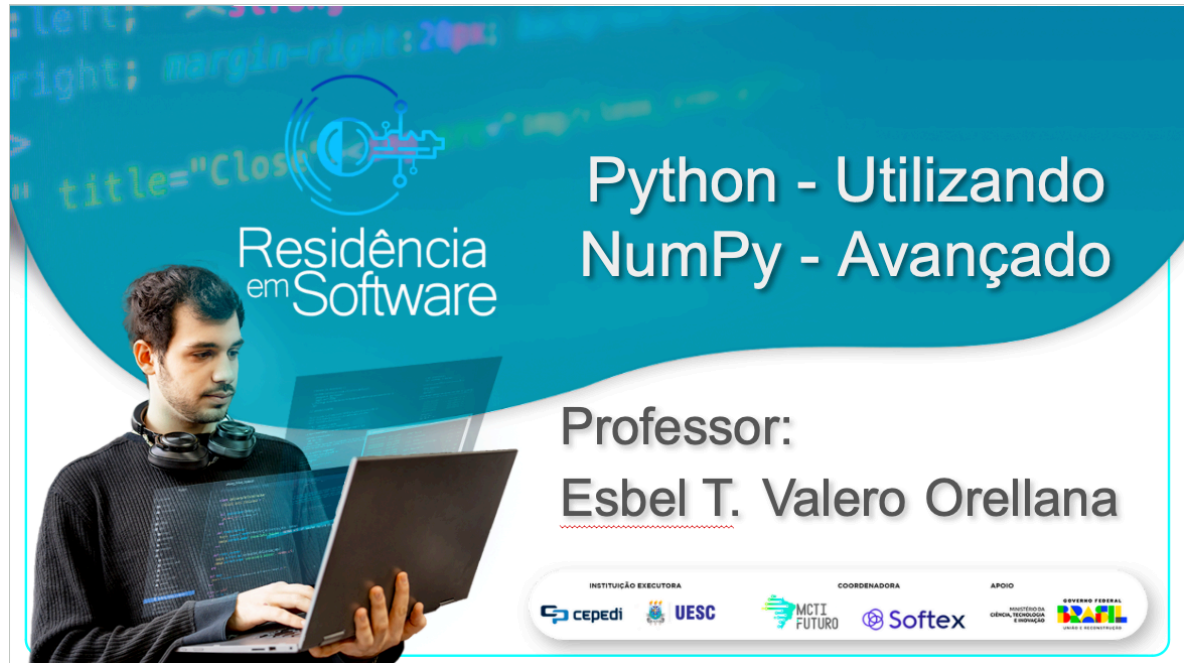


Módulo de Programação Python

Trilha Python - Aula 12: Utilizando NumPy - Avançado



Residência em Software

Python - Utilizando NumPy - Avançado

Professor:
Esbel T. Valero Orellana

INSTITUIÇÃO EXECUTORA: CEPEDI, UESC

COORDENADORA: MCTI FUTURO, Softex

APOIO: INSTITUTO NACIONAL DE CIÊNCIAS E INOVAÇÃO, GOVERNO FEDERAL

Objetivo: Trabalhar com pacotes e módulos disponíveis em python: **Numpy**. Aprender a trabalhar de forma eficiente com **NumPy** arrays utilizando as funções universais (*ufunc*) e outros recursos avançados.

Calculando com *ndarrays*

Como já foi discutido antes, a **NumPy** é muito utilizada, não apenas pelas características dos objetos de tipo *ndarray*, nela implementados. O conjunto de operações e funções para trabalhar com os *ndarray* são um importante diferencial.

Operações com arrays envolvem, na maior parte das linguagens de programação, a utilização de laços ou estruturas de repetição para percorrer os elementos dos mesmos.

A implementação de operações vetoriais, disponíveis para o processamento de *ndarrays* na **NumPy**, representam um diferencial importante na hora de processar estruturas de grande porte.

Na aula anterior tentamos utilizar *ndarrays* para implementar multiplicação de matrizes e o resultado não foi muito promissor.

Vejamos outro exemplo para entender melhor a questão.

```
In [1]: import numpy as np
        from random import uniform
```

```
Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions
4.2 (Intel(R) SSE4.2) enabled only processors has been deprecated.
Intel oneAPI Math Kernel Library 2025.0 will require Intel(R) Advan
ced Vector Extensions (Intel(R) AVX) instructions.
Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions
4.2 (Intel(R) SSE4.2) enabled only processors has been deprecated.
Intel oneAPI Math Kernel Library 2025.0 will require Intel(R) Advan
ced Vector Extensions (Intel(R) AVX) instructions.
```

Vamos criar uma lista muito grande com valores aleatórios entre 1 e 100. Utilizaremos o módulo `uniform` do pacote `random`.

```
In [2]: lista = [uniform(1, 100) for _ in range(1000000)]
        print(lista[:3], "...", lista[-3:])
        print("len(lista) =", len(lista))
```

```
[70.04202653303095, 97.36463582395194, 86.42071979179786] ... [6.
3202587077732675, 11.403852029344547, 84.59511217928622]
len(lista) = 1000000
```

Agora vamos testar o custo computacional de calcular o inverso de cada um dos valores das listas.

```
In [3]: lista_inv = [1/x for x in lista]
print(lista_inv[:3], "...", lista_inv[-3:])

%timeit lista_inv = [1/x for x in lista]
```

[0.014277142588505952, 0.010270669545851652, 0.01157129913299923]
... [0.1582213713451481, 0.08768966814255275, 0.01182101393613209
2]
49.6 ms ± 875 µs per loop (mean ± std. dev. of 7 runs, 10 loops ea
ch)

Podemos fazer o mesmo experimento utilizando *ndarrays*.

```
In [4]: #array = np.random.uniform(1,100,1000000)
array = np.array(lista)
print(array[:3], "...", array[-3:])
print("len(array) =", len(array))
```

[70.04202653 97.36463582 86.42071979] ... [6.32025871 11.4038520
3 84.59511218]
len(array) = 1000000

```
In [5]: def inv(x):
        y = np.empty_like(x)
        for i in range(len(x)):
            y[i] = 1/x[i]
        return y

array_inv = inv(array)

print(array_inv[:3], "...", array_inv[-3:])

%timeit array_inv = inv(array)
```

[0.01427714 0.01027067 0.0115713] ... [0.15822137 0.08768967 0.0
1182101]
224 ms ± 7.88 ms per loop (mean ± std. dev. of 7 runs, 1 loop eac
h)

Vejam que o desempenho com *ndarrays* é pior que com listas. Este não era o resultado esperado.

Entretanto, **NumPy** disponibiliza uma interface apropriada que permite introduzir operações vetoriais, o que acelera significativamente o processamento de *ndarrays* de grande porte. Compare o resultado anterior com o do exemplo a seguir.

```
In [6]: #array = np.random.uniform(1,100,1000000)
array = np.array(lista)
print(array[:3], " ...", array[-3:])

array_inv = 1.0/array
print(array_inv[:3], " ...", array_inv[-3:])

%timeit array_inv = (1.0/array)
```

```
[70.04202653 97.36463582 86.42071979] ... [ 6.32025871 11.4038520
3 84.59511218]
[0.01427714 0.01027067 0.0115713 ] ... [0.15822137 0.08768967 0.0
1182101]
379 µs ± 7.23 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops
each)
```

As operações vetoriais são implementadas em **NumPy** através das chamadas *ufuncs*. As *ufuncs* são eficientes e flexíveis, permitindo realizar, de forma rápida, operações entre escalares e arrays, assim como operações entre arrays.

Realizar os cálculos utilizando *ufuncs* é sempre mais eficiente a implementação do mesmo cálculo utilizando estruturas de repetição. Veja os operadores aritméticos implementados em **NumPy** através de *ufuncs* nos exemplos a seguir.

| Operador | <i>ufunc</i> equivalente | Descrição |
|----------|------------------------------|---------------------------------------------------------------|
| + | <code>np.add</code> | Adição de dois arrays ou de um array com um escalar |
| - | <code>np.subtract</code> | Subtração de dois arrays ou de um array com um escalar |
| - | <code>np.negative</code> | Negativo unário |
| * | <code>np.multiply</code> | Multiplicação de dois arrays ou de um array por um escalar |
| / | <code>np.divide</code> | Divisão de dois arrays ou de um array por um escalar |
| // | <code>np.floor_divide</code> | Divisão truncada de dois arrays ou de um array por um escalar |
| ** | <code>np.power</code> | Exponenciação |
| % | <code>np.mod</code> | Resto da divisão |

```
In [7]: # Adição de arrays
x = np.array([1,2,3])
y = np.array([4,5,6])
z = x + y
print(z)

# este operador adição é implementado como uma _ufunc_ (função univ
print(np.add(x,y))
print("_____")
# Adição de array e escalar
z = x + 10
print(z)
print(np.add(x,10))
```

[5 7 9]

[5 7 9]

[11 12 13]

[11 12 13]

```
In [8]: # subtração de arrays
z = x - y
print(z)

# este operador subtração é implementado como uma _ufunc_ (função u
print(np.subtract(x,y))
print("_____")
# subtração de array e escalar
z = x - 10
print(z)
print(np.subtract(x,10))
```

[-3 -3 -3]

[-3 -3 -3]

[-9 -8 -7]

[-9 -8 -7]

```
In [9]: # negativo
z = -z
print(z)

# este operador negativo é implementado como uma _ufunc_ (função un
print(np.negative(z))
```

[9 8 7]

[-9 -8 -7]

```
In [10]: # multiplicação de arrays
z = x * y
print(z)

# este operador multiplicação é implementado como uma _ufunc_ (função
print(np.multiply(x,y))
print("_____")
# multiplicação de array e escalar
z = 2 * x
print(z)
print(np.multiply(2,x))
```

```
[ 4 10 18]
```

```
[ 4 10 18]
```

```
[2 4 6]
```

```
[2 4 6]
```

```
In [11]: # divisão de arrays
z = x / y
print(z)
# este operador divisão é implementado como uma _ufunc_ (função uni
print(np.divide(x,y))
print("_____")
# divisão de array e escalar
z = x / 2
print(z)
print(np.divide(x,2))
```

```
[0.25 0.4  0.5 ]
```

```
[0.25 0.4  0.5 ]
```

```
[0.5 1.  1.5]
```

```
[0.5 1.  1.5]
```

```
In [12]: # divisão truncada de arrays
z = x // y
print(z)
# este operador divisão é implementado como uma _ufunc_ (função uni
print(np.floor_divide(x,y))
print("_____")
# divisão truncada de array e escalar
z = x // 2
print(z)
print(np.floor_divide(x,2))
```

```
[0 0 0]
[0 0 0]
```

```
[0 1 1]
[0 1 1]
```

```
In [13]: # exponenciação de arrays
z = x ** y
print(z)
# este operador divisão é implementado como uma _ufunc_ (função uni
print(np.power(x,y))
print("_____")
# exponenciação de array e escalar
z = x ** 2
print(z)
print(np.power(x,2))
```

```
[ 1 32 729]
[ 1 32 729]
```

```
[1 4 9]
[1 4 9]
```

```
In [14]: # resto da divisão de arrays
z = x % y
print(z)
# este operador divisão é implementado como uma _ufunc_ (função uni
print(np.mod(x,y))
print("_____")
# resto da divisão de array e escalar
z = x % 2
print(z)
print(np.mod(x,2))
```

```
[1 2 3]
[1 2 3]
```

```
[1 0 1]
[1 0 1]
```

Além deste conjunto básico de operadores, implementados na furma de *ufunc* que sobrecarregam os operadores aritméticos tradicionais, **NumPy** disponibiliza um conjunto adicional de funções:

| Nome da Função | Descrição |
|-----------------------------|----------------------------------------------------------------------------------------------------------------------|
| np.abs ou np.absolute | Retorna o valor absoluto dos elementos do <i>ndarray</i> . Também funciona com <i>ndarrays</i> de números complexos. |

```
In [15]: # valor absoluto
x = np.array([-1.2,2.3,-3.4])
z = np.abs(x)
print(z)
# este operador divisão é implementado como uma _ufunc_ (função uni
print(np.absolute(x))
#ou
print(np.abs(x))
# No caso de números complexos, o valor absoluto é a magnitude
x = np.array([-1+1j,2-2j,-3+3j])
z = np.abs(x)
print(z)
```

```
[1.2 2.3 3.4]
[1.2 2.3 3.4]
[1.2 2.3 3.4]
[1.41421356 2.82842712 4.24264069]
```


Funções Trigonométricas

| Nome da Função | Descrição |
|------------------------|------------------------------------------------|
| <code>np.sin</code> | Retorna o seno dos elementos do array |
| <code>np.cos</code> | Retorna o cosseno dos elementos do array |
| <code>np.tan</code> | Retorna a tangente dos elementos do array |
| <code>np.arcsin</code> | Retorna o arco-seno dos elementos do array |
| <code>np.arccos</code> | Retorna o arco-cosseno dos elementos do array |
| <code>np.arctan</code> | Retorna a arco-tangente dos elementos do array |

```
In [16]: ang = np.linspace(0,2*np.pi,25) # ângulos em radianos
# print(ang)
print(np.rad2deg(ang)) # ângulos em graus
```

```
[ 0.  15.  30.  45.  60.  75.  90. 105. 120. 135. 150. 165. 180.
 195.
 210. 225. 240. 255. 270. 285. 300. 315. 330. 345. 360.]
```

```
In [17]: print("Seno: \n", np.sin(ang))
print("Cosseno: \n", np.cos(ang))
print("Tangente: \n", np.tan(ang))
```

Seno:

```
[ 0.00000000e+00  2.58819045e-01  5.00000000e-01  7.07106781e-01
 8.66025404e-01  9.65925826e-01  1.00000000e+00  9.65925826e-01
 8.66025404e-01  7.07106781e-01  5.00000000e-01  2.58819045e-01
 1.22464680e-16 -2.58819045e-01 -5.00000000e-01 -7.07106781e-01
 -8.66025404e-01 -9.65925826e-01 -1.00000000e+00 -9.65925826e-01
 -8.66025404e-01 -7.07106781e-01 -5.00000000e-01 -2.58819045e-01
 -2.44929360e-16]
```

Cosseno:

```
[ 1.00000000e+00  9.65925826e-01  8.66025404e-01  7.07106781e-01
 5.00000000e-01  2.58819045e-01  6.12323400e-17 -2.58819045e-01
 -5.00000000e-01 -7.07106781e-01 -8.66025404e-01 -9.65925826e-01
 -1.00000000e+00 -9.65925826e-01 -8.66025404e-01 -7.07106781e-01
 -5.00000000e-01 -2.58819045e-01 -1.83697020e-16  2.58819045e-01
 5.00000000e-01  7.07106781e-01  8.66025404e-01  9.65925826e-01
 1.00000000e+00]
```

Tangente:

```
[ 0.00000000e+00  2.67949192e-01  5.77350269e-01  1.00000000e+00
 1.73205081e+00  3.73205081e+00  1.63312394e+16 -3.73205081e+00
 -1.73205081e+00 -1.00000000e+00 -5.77350269e-01 -2.67949192e-01
 -1.22464680e-16  2.67949192e-01  5.77350269e-01  1.00000000e+00
 1.73205081e+00  3.73205081e+00  5.44374645e+15 -3.73205081e+00
 -1.73205081e+00 -1.00000000e+00 -5.77350269e-01 -2.67949192e-01
 -2.44929360e-16]
```

```
In [18]: print("Arco seno: \n", np.rad2deg(np.arcsin(np.sin(ang))))
print("Arco cosseno: \n", np.rad2deg(np.arccos(np.cos(ang))))
print("Arco tangente: \n", np.rad2deg(np.arctan(np.tan(ang))))
```

Arco seno:

```
[ 0.00000000e+00  1.50000000e+01  3.00000000e+01  4.50000000e+01
 6.00000000e+01  7.50000000e+01  9.00000000e+01  7.50000000e+01
 6.00000000e+01  4.50000000e+01  3.00000000e+01  1.50000000e+01
 7.01670930e-15 -1.50000000e+01 -3.00000000e+01 -4.50000000e+01
-6.00000000e+01 -7.50000000e+01 -9.00000000e+01 -7.50000000e+01
-6.00000000e+01 -4.50000000e+01 -3.00000000e+01 -1.50000000e+01
-1.40334186e-14]
```

Arco cosseno:

```
[ 0.  15.  30.  45.  60.  75.  90. 105. 120. 135. 150. 165. 180.
165.
150. 135. 120. 105.  90.  75.  60.  45.  30.  15.   0.]
```

Arco tangente:

```
[ 0.00000000e+00  1.50000000e+01  3.00000000e+01  4.50000000e+01
 6.00000000e+01  7.50000000e+01  9.00000000e+01 -7.50000000e+01
-6.00000000e+01 -4.50000000e+01 -3.00000000e+01 -1.50000000e+01
-7.01670930e-15  1.50000000e+01  3.00000000e+01  4.50000000e+01
 6.00000000e+01  7.50000000e+01  9.00000000e+01 -7.50000000e+01
-6.00000000e+01 -4.50000000e+01 -3.00000000e+01 -1.50000000e+01
-1.40334186e-14]
```

Funções exponenciais e logarítmicas

| Nome da Função | | Descrição |
|----------------|--------------------------------------------|-----------------------|
| np.exp | | Retorna e^x |
| np.exp2 | | Retorna 2^x |
| np.power | | Retorna a^x |
| np.ln | Retorna $\log_e x$ ou simplesmente $\ln x$ | |
| np.log2 | | Retorna $\log_2 x$ |
| np.log10 | | Retorna $\log_{10} x$ |

```
In [19]: x = np.array([1,2,3])
print("Exponencial (e**x): \n", np.exp(x))
print("Exponencial (2**x): \n", np.exp2(x))
print("Exponencial (10**x): \n", np.power(10,x))
```

Exponencial (e**x):

```
[ 2.71828183  7.3890561  20.08553692]
```

Exponencial (2**x):

```
[2.  4.  8.]
```

Exponencial (10**x):

```
[ 10  100 1000]
```

```
In [20]: print("Logaritmo natural: \n", np.log(np.exp(x)))
print("Logaritmo base 2: \n", np.log2(np.exp2(x)))
print("Logaritmo base 10: \n", np.log10(np.power(10,x)))
```

```
Logaritmo natural:
[1. 2. 3.]
Logaritmo base 2:
[1. 2. 3.]
Logaritmo base 10:
[1. 2. 3.]
```

```
In [21]: #outros logaritmos
#log base 3 de x pode ser calculado como log(x)/log(3)
print("Logaritmo base 3: \n", np.log(np.power(3,x))/np.log(3))
```

```
Logaritmo base 3:
[1. 2. 3.]
```

As limitações impostas pela aritmética de ponto flutuante faz com que, em alguns casos, seja necessário utilizar artifícios matemáticos para melhorar a precisão dos resultados. Para estes casos a **NumPy** disponibiliza funções especiais como a utilizada no seguinte exemplo.

```
In [22]: x = np.array([0, 0.001, 0.01, 0.1])
print("Exponencial (e**x): \n", np.exp(x))
print("Exponencial (e**x - 1): \n", np.expm1(x))
#print("Logaritmo natural: \n", np.log(x))
print("Logaritmo natural (1 + x): \n", np.log1p(x))
```

```
Exponencial (e**x):
[1.          1.0010005  1.01005017  1.10517092]
Exponencial (e**x - 1):
[0.          0.0010005  0.01005017  0.10517092]
Logaritmo natural (1 + x):
[0.          0.0009995  0.00995033  0.09531018]
```

Para valores do argumento muito pequenos estas funções conseguem retornar um resultado com maior precisão.

MAs vamos retomar o exemplo do final da aula anterior

Relembrando a definição de GEMM que implementa a seguinte operação

$$C = \alpha AB + \beta C$$

De forma que:

$$C[i, j] = \alpha \sum_{k=0}^{k<l} A[i, k]B[k, j] + \beta C[i, j]$$

Vamos primeiramente revisar a implementação baseada exclusivamente no uso de estruturas de repetição.

```
In [23]: #Esta é uma implementação específica para ndarrays
def GEMM_loops(alpha, A, B, beta, C):
    ma, la = A.shape
    lb, nb = B.shape
    mc, nc = C.shape
    if (ma != mc) or (la != lb) or (nb != nc):
        return C

    for i in range(mc):
        for j in range(nc):
            val = 0
            for k in range(la):
                val += A[i,k]*B[k,j]
            C[i,j] = alpha*val + beta*C[i,j]
    return C
```

```
In [24]: # Dada uma matriz A de n linhas e l colunas
n = 256
l = 128
A = np.random.random((n,l))
#A = np.ones((n,l))
# Uma matriz B de l linhas e n colunas
m = 256
B = np.random.random((l,m))
#B= np.ones((l,m))
# Uma matriz C de n linhas e m colunas
C = np.ones((n,m))
#C = np.zeros((n,m))
C = np.random.random((n,m))
# E os escalares alpha e beta
alpha = 0.5
#alpha = 1.0
beta = 1.5
#beta = 1.0
print(A)
print(B)
print(C)
```

```
In [25]: C1 = C.copy()
%timeit Z = GEMM_loops(alpha, A, B, beta, C1)
C1 = GEMM_loops(alpha, A, B, beta, C1)
```

2.69 s \pm 34.8 ms per loop (mean \pm std. dev. of 7 runs, 1 loop each)

Podemos então tentar usar a *ufunc* para melhorar o desempenho

```
In [26]: def GEMM_ufunc1(alpha, A, B, beta, C):
    ma, la = A.shape
    lb, nb = B.shape
    mc, nc = C.shape
    if (ma != mc) or (la != lb) or (nb != nc):
        return C

    C = beta * C
    for i in range(mc):
        for j in range(nc):
            val = 0
            for k in range(la):
                val += A[i,k]*B[k,j]
            C[i,j] += alpha*val

    return C
```

```
In [27]: C2 = C.copy()
%timeit Z = GEMM_ufunc1(alpha, A, B, beta, C2)
C2 = GEMM_ufunc1(alpha, A, B, beta, C2)
```

2.75 s \pm 29.8 ms per loop (mean \pm std. dev. of 7 runs, 1 loop each)

```
In [28]: def GEMM_ufunc2(alpha, A, B, beta, C):
    ma, la = A.shape
    lb, nb = B.shape
    mc, nc = C.shape
    if (ma != mc) or (la != lb) or (nb != nc):
        return C

    C = beta * C
    for i in range(mc):
        for j in range(nc):
            C_ = A[i,:] * B[:,j].T
            val = 0
            for k in range(la):
                val += C_[k]
            C[i,j] += alpha*val

    return C
```

```
In [29]: C3 = C.copy()
%timeit Z = GEMM_ufunc2(alpha, A, B, beta, C3)
C3= GEMM_ufunc2(alpha, A, B, beta, C3)
```

1.13 s ± 5.79 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

Queda significativa no tempo total de processamento. Podemos melhorar ainda mais?

Quando trabalhamos com uma grande quantidade de dados, muitas vezes se faz necessário começar por fazer uma análise estatística dos mesmos,

Algumas métricas utilizadas em estatística, como as medidas de valor central ou as medidas de espalhamento, ou ainda medidas de correlação, podem ser um necessárias.

De forma geral a média e o desvio padrão é bom ponto de partida.

NumPy disponibiliza funções de agregação integradas rápidas para trabalhar em *ndarrays* que são muito relevantes, por exemplo, para este tipo de análises.

Podemos começar pelo algoritmo simples, que já foi utilizado anteriormente, para calcular a soma de um conjunto de elementos, por exemplo, para calcular a média do conjunto.

Se os valores estão numa lista, podemos utilizar a função `sum`

```
In [30]: from random import random
matSize = 512
vetX = [random() for i in range(matSize)]
soma = sum(vetX)
print(soma)
```

254.25557456489736

Podemos obter o mesmo resultado utilizando os recursos da **NumPy**, particularmente a função `sum`

```
In [31]: #import numpy as np
x = np.array(vetX)
soma = np.sum(x)
print(soma)
```

254.2555745648973

Vamos comparar o desempenho destas duas implementações.

```
In [32]: matSize = 1000000
vetX = [random() for i in range(matSize)]
x = np.array(vetX)
%timeit sum(vetX)
%timeit np.sum(x)
```

3.79 ms \pm 65.7 μ s per loop (mean \pm std. dev. of 7 runs, 100 loops each)

196 μ s \pm 3.07 μ s per loop (mean \pm std. dev. of 7 runs, 10,000 loops each)

Repare que esta função pode ser utilizada na nossa implementação da para *ndarrays* do **GEMM**.

```
In [33]: def GEMM_ufunc3(alpha, A, B, beta, C):
    ma, la = A.shape
    lb, nb = B.shape
    mc, nc = C.shape
    if (ma != mc) or (la != lb) or (nb != nc):
        return C

    C = beta * C
    for i in range(mc):
        for j in range(nc):
            C[i,j] += alpha*np.sum(A[i,:] * B[:,j].T)

    return C
```

```
In [34]: C4 = C.copy()
%timeit Z = GEMM_ufunc3(alpha, A, B, beta, C4)
C4= GEMM_ufunc3(alpha, A, B, beta, C4)
```

300 ms \pm 5.78 ms per loop (mean \pm std. dev. of 7 runs, 1 loop each)

No exercício da prova do módulo anterior utilizamos as funções `min` e `max`, que também tem implementações muito eficientes na **NumPy**

```
In [35]: print(max(vetX))
print(np.max(x))
%timeit max(vetX)
%timeit np.max(x)
```

0.9999998776514297

0.9999998776514297

10.8 ms \pm 124 μ s per loop (mean \pm std. dev. of 7 runs, 100 loops each)

321 μ s \pm 8.44 μ s per loop (mean \pm std. dev. of 7 runs, 1,000 loops each)

```
In [36]: print(min(vetX))
print(np.min(x))
%timeit min(vetX)
%timeit np.min(x)
```

1.7100599717378984e-06

1.7100599717378984e-06

10.8 ms \pm 48.7 μ s per loop (mean \pm std. dev. of 7 runs, 100 loops each)

316 μ s \pm 3.67 μ s per loop (mean \pm std. dev. of 7 runs, 1,000 loops each)

Este tipo de funções são chamadas de funções agregadoras. Outras funções de agregação estão disponíveis. A maioria delas uma versão segura para **NaN**, que calcula o resultado ignorando os valores ausentes, que são marcados pelo valor **NaN** de ponto flutuante. Algumas dessas funções seguras para NaN não foram adicionadas até o NumPy 1.8, portanto, não estarão disponíveis em versões mais antigas do NumPy. Veja a tabela a seguir.

| Função | Versão __NaN__safe | Descrição |
|----------------------------|-------------------------------|----------------------------------------------|
| <code>np.sum</code> | <code>np.nansum</code> | Calcula a soma dos elementos |
| <code>np.prod</code> | <code>np.nanprod</code> | Calcula o produto dos elementos |
| <code>np.mean</code> | <code>np.nanmean</code> | Calcula o valor médio |
| <code>np.std</code> | <code>np.nanstd</code> | Calcula o desvio padrão |
| <code>np.var</code> | <code>np.nanvar</code> | Calcula a variância |
| <code>np.min</code> | <code>np.nanmin</code> | Retorna o valor mínimo |
| <code>np.max</code> | <code>np.nanmax</code> | Retorna o valor máximo |
| <code>np.argmin</code> | <code>np.nanargmin</code> | Retorna o índice de valor mínimo |
| <code>np.argmax</code> | <code>np.nanargmax</code> | Retorna o índice de valor máximo |
| <code>np.median</code> | <code>np.nanmedian</code> | Calcula a mediana |
| <code>np.percentile</code> | <code>np.nanpercentile</code> | Calcula os percentil |
| <code>np.any</code> | N/A | Avalie se algum elemento é verdadeiro |
| <code>np.all</code> | N/A | Avalie se todos os elementos são verdadeiros |

Estas funções permitam trabalhar com arrays multidimensionais.

```
In [37]: A = np.random.random((3,3))
print(A)
print("Soma: ", np.sum(A))
print("Máximo: ", np.max(A))
print("Índice do máximo: ", np.argmax(A))
print("Mínimo: ", np.min(A))
print("Índice do mínimo: ", np.argmin(A))
print("Média: ", np.mean(A))
print("Mediana: ", np.median(A))
print("Desvio padrão: ", np.std(A))
```

```
[[0.39375745 0.23865957 0.07124863]
 [0.41841007 0.35866372 0.04508146]
 [0.86101667 0.02969938 0.15189183]]
Soma: 2.568428767203257
Máximo: 0.8610166691949342
Índice do máximo: 6
Mínimo: 0.02969937571279946
Índice do mínimo: 7
Média: 0.28538097413369523
Mediana: 0.23865956574073277
Desvio padrão: 0.24835924967575895
```

As funções de agregação pode receber também um argumento adicional que especifica o eixo ao longo do qual a agregação deve ser calculada. Por exemplo, podemos encontrar o valor máximo de cada coluna especificando `axis=0`, e de cada linha especificando `axis=1`

```
In [38]: print(A)
print("Soma: ", np.sum(A, axis=0))
print("Máximo: ", np.max(A, axis=0))
print("Índice do máximo: ", np.argmax(A, axis=0))
print("Mínimo: ", np.min(A, axis=0))
print("Índice do mínimo: ", np.argmin(A, axis=0))
print("Média: ", np.mean(A, axis=0))
print("Mediana: ", np.median(A, axis=0))
print("Desvio padrão: ", np.std(A, axis=0))
```

```
[[0.39375745 0.23865957 0.07124863]
 [0.41841007 0.35866372 0.04508146]
 [0.86101667 0.02969938 0.15189183]]
Soma: [1.67318419 0.62702266 0.26822192]
Máximo: [0.86101667 0.35866372 0.15189183]
Índice do máximo: [2 1 2]
Mínimo: [0.39375745 0.02969938 0.04508146]
Índice do mínimo: [0 2 1]
Média: [0.55772806 0.20900755 0.08940731]
Mediana: [0.41841007 0.23865957 0.07124863]
Desvio padrão: [0.21469346 0.135926 0.04545633]
```

Temos ainda a implementação dos produtos vetoriais e matriciais da álgebra linear.

A função `np.dot` implementa o produto escalar de duas matrizes.

- Se `a` e `b` são matrizes 1-D, o resultado é o produto interno de vetores (sem complexa conjugada).
- Se `a` e `b` forem matrizes 2-D, o resultado é uma multiplicação de matrizes, mas é preferível usar `matmul` ou `a @ b`.

Se `a` ou `b` for escalar, é equivalente a multiplicar e usar `numpy.multiply(a, b)` ou `a * b`.

```
In [39]: x = np.array([1,2,3])
         y = np.array([4,5,6])

         z = np.dot(x,y)
         print(z)
```

32

```
In [40]: A = np.random.random((3,3))
         B = np.random.random((3,3))

         C = np.dot(A,B)
         print(C)
```

```
[[0.23901021 0.56937862 0.53451353]
 [0.26087655 0.46218655 0.32691028]
 [0.33864464 0.78052917 0.6912328 ]]
```

Podemos retomar nossa implementação a GEMM para usar o produto escalar de vetores.

```
In [41]: # Dada uma matriz A de n linhas e l columnas
n = 256
l = 128
A = np.random.random((n,l))
#A = np.ones((n,l))
# Uma matriz B de l linhas e n columnas
m = 256
B = np.random.random((l,m))
#B= np.ones((l,m))
# Uma matriz C de n linhas e m columnas
#C = np.ones((n,m))
#C = np.zeros((n,m))
C = np.random.random((n,m))
# E os escalares alpha e beta
alpha = 0.5
#alpha = 1.0
beta = 1.5
#beta = 1.0
#print(A)
#print(B)
#print(C)
```

```
In [42]: def GEMM_ufunc4(alpha, A, B, beta, C):
    ma, la = A.shape
    lb, nb = B.shape
    mc, nc = C.shape
    if (ma != mc) or (la != lb) or (nb != nc):
        return C

    C = beta * C
    for i in range(mc):
        for j in range(nc):
            C[i,j] += alpha*np.dot(A[i,:],B[:,j]).T

    return C
```

```
In [43]: C5 = C.copy()
%timeit Z = GEMM_ufunc4(alpha, A, B, beta, C5)
C5= GEMM_ufunc4(alpha, A, B, beta, C4)
```

101 ms ± 1.51 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

```
In [44]: %timeit Z = alpha*np.dot(A,B)+beta*C
print((alpha*np.dot(A,B)+beta*C).shape)
```

600 µs ± 25.5 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
(256, 256)

```
In [45]: %timeit Z = alpha*np.matmul(A,B) + beta*C
print((alpha*np.matmul(A,B) + beta*C).shape)
```

579 μ s \pm 29.8 μ s per loop (mean \pm std. dev. of 7 runs, 1,000 loops each)
(256, 256)

```
In [46]: %timeit Z = alpha*A@B + beta*C
print((alpha*A@B + beta*C).shape)
```

626 μ s \pm 87.6 μ s per loop (mean \pm std. dev. of 7 runs, 1,000 loops each)
(256, 256)

```
In [47]: # Dada uma matriz A de n linhas e l columnas
n = 1024
l = 1024
A = np.random.random((n,l))
#A = np.ones((n,l))
# Uma matriz B de l linhas e n columnas
m = 1024
B = np.random.random((l,m))
#B= np.ones((l,m))
# Uma matriz C de n linhas e m columnas
#C = np.ones((n,m))
#C = np.zeros((n,m))
C = np.random.random((n,m))
# E os escalares alpha e beta
alpha = 0.5
#alpha = 1.0
beta = 1.5
#beta = 1.0
#print(A)
#print(B)
#print(C)
```

```
In [48]: %timeit Z = GEMM_ufunc4(alpha, A, B, beta, C)
```

8.5 s \pm 196 ms per loop (mean \pm std. dev. of 7 runs, 1 loop each)

```
In [49]: gflop = (2.0*1024 + 2)*(1024**2)*1E-9
print(gflop/8.44)
```

0.25468966824644557

```
In [50]: %timeit Z = alpha*np.dot(A,B)+beta*C
```

31.7 ms \pm 1.11 ms per loop (mean \pm std. dev. of 7 runs, 10 loops each)

```
In [51]: print(gflop/(31.8*1E-3))
```

67.59688050314466

```
In [52]: %timeit Z = alpha*np.matmul(A,B) + beta*C
```

32.4 ms \pm 1.53 ms per loop (mean \pm std. dev. of 7 runs, 10 loops each)

```
In [53]: %timeit Z = alpha*A@B + beta*C
```

36.1 ms \pm 3.94 ms per loop (mean \pm std. dev. of 7 runs, 10 loops each)

```
In [ ]:
```