Aula 4A - Computação Vetorizada com *numpy*

Gustavo Oliveira[1] e Andrea Rocha[1]

[1]Departamento de Computação Científica / UFPB

Julho de 2020

1 Computação vetorizada com *numpy*

1.1 Computação vetorizada

A computação científica é uma ciência interdisciplinar que procura resolver problemas de alta complexidade utilizando recursos computacionais. Ao longo das décadas, a capacidade de processamento computacional aumentou dramaticamente. Em junho de 2020, o site [Top500], que atualiza semestralmente a lista dos computadores mais potentes do mundo, divulgou que o primeiro da lista agora é o Fugaku, um computador de alto desempenho japonês capaz de realizar 415,5 petaflops no teste de referência Linpack para medição de desempenho. Se um petaflop já equivale à impressionante marca de um quadrilhão (10^{15}) de operações numéricas executadas por segundo, o que dizer de 415,5 vezes isto?! É realmente um número inimaginável. Para saber mais sobre medidas utilizadas na computação de alto desempenho, veja esta página da [Universidade de Indiana].

Todo este potencial serve para resolver problemas de engenharia, das ciências da saúde, das ciências atmosféricas, de energia, entre tantas outras áreas. Por outro lado, a computação científica depende de algoritmos e projetos de software otimizados e bem construídos para atingir cada vez mais eficência, portabilidade e facilidade de execução. Em ciência de dados, embora o foco não seja resolver problemas utilizando os mesmos moldes da computação científica, devemos ter em mente que grande parte das ferramentas computacionais que permitem que um problema da ciência de dados seja resolvido deve-se a um trabalho muito minucioso de cientistas da computação e engenheiros que trabalham em nível de hardware. E é aí que entra a *computação vetorizada*.

O conceito de *computação vetorizada*, ou *computação baseada em arrays* está relacionado à execução de operações que podem ser feitas de uma só vez a um conjunto amplo de valores. Até agora, vimos no curso, por exemplo, que podemos percorrer uma lista de números inteiros e calcular o quadrado desses números um de cada vez utilizando um laço, ou mesmo realizando um mapeamento. A computação vetorizada, por outro lado, evita que tais operações dependam de laços e iterações. Com ela, podemos simplesmente aplicar uma função ao *array* (uma coleção de dados) inteiro de uma só vez e produzir um *array* com o resultado desejado diretamente. Ou seja, a ideia principal da computação vetorizada é evitar laços e cálculos com repetições a fim de acelerar operações matemáticas. O nome *vetorizada* está relacionado a *vetor*. Como veremos aqui, estruturas multidimensionais e, mais geralmente, *arrays*, identificam-se com a nossa compreensão de vetores, matrizes e tensores.

Vetores são arrays unidimensionais. Matrizes são arrays bidimensionais. Tensores são arrays

de três ou mais dimensões. Todavia, é importante salientar que o conceito de "dimensão" em *computação vetorizada* deve ser distinguido da ideia mais abstrata de dimensão como você estudadará em Cálculo Vetorial, Geometria Analítica ou Álgebra Linear. *Arrays* possuem alguns atributos, tais como "comprimento", "formato" e "dimensão, os quais dizem respeito, de certa forma, à quantidade de seus elementos e ao modo como ocupam a memória. Esses nomes variam de linguagem para linguagem. Em Python, existem funções e métodos específicos para verificar comprimento, formato e dimensão, tais como `len`, `shape` e `ndim`. As duas últimas serão apresentadas a seguir. Em outras linguagens, usa-se também `size` para desempenhar o mesmo papel que `shape`. A palavra"dimension", em Python, é encontrada de maneira explicativa em documentações. Para não confundir"comprimento","formato"e "dimensão", redobre seu entendimento.

1.2 Comprimento, tamanho e dimensão

Para exemplificar o que queremos dizer com "comprimento", "tamanho" e "dimensão", vejamos uma ilustração. Se x_1 e x_2 são dois números inteiros, a lista `[x1,x2]` seria um array unidimensional, mas de comprimento dois (você verifica isto com `len`). Agora, imagine que (x_1, x_2) seja a notação matemática para representar as coordenadas de um ponto do plano cartesiano. Sabemos da geometria que o plano cartesiano tem duas dimensões. Porém, poderíamos, computacionalmente, usar a mesma lista anterior para armazenar no computador essas duas coordenadas. A lista continuaria sendo unidimensional, porém de tamanho dois. Logo, embora a entidade matemática seja bidimensional, não necessariamente a sua representação computacional deve ser bidimensional.

Vejamos outra ilustração. Uma esfera é um sólido geométrico. Cada ponto da esfera está no espaço tridimensional. Isto significa que precisamos de 3 coordenadas para localizar este ponto. Embora você talvez tenha estudado pouco ou nada sobre a geometria analítica em três dimensões, o exemplo que daremos aqui será simples. Do mesmo modo que o caso anterior, suponha você tenha não apenas x_1 e x_2 como dois números inteiros, mas também um terceiro, x_3, para montar as coordenadas do seu ponto espacial. Você poderia representá-lo, matematicamente, por uma tripleta (x_1, x_2, x_3) sem problema algum. Por outro lado, no computador, a lista `[x1,x2,x3]` seria um *array* adequado para armazenar os valores das suas coordendas. Entretanto, esta lista continuaria sendo um array unidimensional, mas com tamanho 3. Portanto, *arrays* unidimensionais podem representar dados em dimensões maiores do que um.

Vejamos outra ilustração. Uma matriz 2 x 2 pode ser escrita, em matemática, utilizando 4 números da seguinte forma:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

Uma matriz é um *array* formado por outros *arrays*. Usando listas, a construção anterior poderia ser representada, no computador, por exemplo, com a lista de listas `[[a11,a12],[a21,a22]]`. Porém, teríamos que ter total controle sobre isto para poder dizer que `[a11,a12]` é a primeira linha da matriz e que `[a21,a22]` é a segunda linha. Neste exemplo, a matriz é uma entidade matemática caracterizada como bidimensional por ser um "retângulo (ou quadrado) cheio de números" , mas para a Python, ela deve ser um *array bidimensional* basicamente porque existem "duas direções" nos dados: linhas e colunas. Agora, você consegue imaginar o que seria uma matriz de matrizes?

Imagine duas folhas de papel A4 postas uma sobre a outra em uma mesa. Agora, pense como se cada folha de papel fosse uma matriz. No final das contas, você tem uma "caixa com matrizes dentro". Isto é o que seria um tensor, uma "caixa de números", embora, física e matematicemente, não

seja esta a definição. Uma matriz de matrizes é um *array tridimensional*. Se as suas duas matrizes fossem idênticas e iguais à do exemplo acima, você teria duas listas de listas `[[a11,a12],[a21,a22]]`, quatro listas (2 iguais a `[a11,a12]` e 2 iguais a `[a21,a22]`) e uma superlista com duas listas de listas dentro!

Como veremos adiante, o *numpy* é a ferramenta ideal para lidar com tudo isso.

1.3 O pacote *numpy*

O *numpy* é a biblioteca padrão em Python para trabalhar com *arrays* multidimensionais e computação vetorizada. Ela praticamente dá "superpoderes" às listas e permite que trabalhemos com cálculos numéricos de maneira ágil, simples e eficiente. Com *numpy*, também podemos ler e escrever arquivos, trabalhar com sistemas lineares e realizar muito mais. Para importar o numpy use a instrução abaixo, onde `np` é um alias padrão:

```
import numpy as np
```

Nesta aula, daremos uma introdução aos aspectos elementos do *numpy* para criar e manipular *arrays* multdimensionais. A grande regra é: vetorize seus cálculos numéricos o máximo possível!

[1]:
```
import numpy as np
```

1.4 Motivação

Este exemplo compara a eficiência de operações feitas com listas comuns e com *numpy*.

[2]:
```
# 1 µs = 1/1e6 segundo
# 1 ns = 1/1e9 segundo

L = range(500)
%timeit -n 10 [i**2 for i in L] # executa o laço 10 vezes

a = np.arange(500)
%timeit -n 10 a**2 # eleva ao quadrado diretamente 10 vezes
```

```
117 µs ± 7.12 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
908 ns ± 436 ns per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

1.5 Criação de arrays unidimensionais (1D)

[3]:
```
a = [1,2,3]
np.array(a) # a partir de lista
```

[3]:
```
array([1, 2, 3])
```

[4]:
```
np.array([1,2,3]) # diretamente
```

[4]:
```
array([1, 2, 3])
```

[5]:
```
np.array([2]*5)
```

[5]:
```
array([2, 2, 2, 2, 2])
```

1.6 Criação de arrays bidimensionais (2D)

[6]:
```
A = [ [1,2], [0,2] ] # lista de listas
np.array(A) # matrix 2 x 2
```

[6]:
```
array([[1, 2],
       [0, 2]])
```

[7]:
```
np.array([ [1,2], [0,2] ]) # diretamente
```

[7]:
```
array([[1, 2],
       [0, 2]])
```

[8]:
```
A2 = [[1,2,3],[4,3,2]] # cada lista é uma linha da matriz
np.array(A2) # matriz 2 x 3
```

[8]:
```
array([[1, 2, 3],
       [4, 3, 2]])
```

[9]:
```
np.array([1,1],[0,1]) # parênteses externos são obrigatórios!
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-9-67700dfe8cfa> in <module>
----> 1 np.array([1,1],[0,1]) # parênteses externos são obrigatórios!


TypeError: data type not understood
```

1.6.1 Dimensão, formato e comprimento

[11]:
```
x = np.array(a)
np.ndim(x) # aplica a função ndim
```

[11]:
```
1
```

[12]:
```
x.ndim # como método
```

[12]:
```
1
```

[13]:
```
np.shape(x) # formato
```

[13]:
```
(3,)
```

[14]:
```
x.shape # sem parênteses
```

[14]:
```
(3,)
```

[15]:
```
len(x) # comprimento
```

[15]:
```
3
```

[16]:
```
X = np.array(A)
np.ndim(X) # array bidimensional
```

[16]:
```
2
```

[17]:
```
np.shape(X)
```

[17]:
```
(2, 2)
```

[18]:
```
len(X) # apenas um comprimento. Qual?
```

[18]:
```
2
```

[19]:
```
X2 = np.array(A2)
len(X2) # apenas da primeira dimensão. LINHAS
```

[19]:
```
2
```

[20]:
```
X2.shape
```

[20]:
```
(2, 3)
```

1.7 Funções para criação de arrays

1.7.1 `arange`

Exemplo: crie um array de números ímpares positivos menores do que 36.

[21]:
```
np.arange(1,36,2) # start,stop,step
```

[21]:
```
array([ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33,
       35])
```

Exemplo: crie um array de números pares positivos menores ou iguais a 62.

[22]:
```
np.arange(0,63,2)
```

[22]:
```
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32,
       34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62])
```

Exemplo: calcule o valor de $f(x) = x^3 + 2x$ para x elementos dos arrays anteriores.

[23]:
```
imp, par = np.arange(1,36,2), np.arange(0,63,2)
f = lambda x: x**3 + 2
fi, fp = f(imp), f(par)
print(fi)
print(fp)
```

```
[    3    29   127   345   731  1333  2199  3377  4915  6861  9263 12169
 15627 19685 24391 29793 35939 42877]
[     2     10     66    218    514   1002   1730   2746   4098   5834
   8002  10650  13826  17578  21954  27002  32770  39306  46658  54874
  64002  74090  85186  97338 110594 125002 140610 157466 175618 195114
 216002 238330]
```

1.7.2 `linspace`

Exemplo: crie um array igualmente espaçado de elementos em [0,1] com 11 elementos.

[24]:
```
np.linspace(0,1,num=11)
```

[24]:
```
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

[25]:
```
np.linspace(0,1,11) # 'num' pode ser omitido
```

[25]:
```
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

[26]:
```
x = np.linspace(0,1,10,endpoint=False) # exclui último
x
```

[26]:
```
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9])
```

[27]:
```
y = np.arange(0,1,0.1) # equivalente
y
```

[27]:
```
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9])
```

[28]:
```
x == y # comparação é elemento a elemento
```

[28]:
```
array([ True,  True,  True,  True,  True,  True,  True,  True,  True,
        True])
```

[29]:
```
x == -y # apenas 0 é True
```

[29]:
```
array([ True, False, False, False, False, False, False, False, False,
       False])
```

[30]:
```
x[1:] == y[1:]
```

[30]:
```
array([ True,  True,  True,  True,  True,  True,  True,  True,  True])
```

1.7.3 `all` e `any`

[31]:
```
np.all( x == y ) # verifica se todos são 'True'
```

[31]:
```
True
```

[32]:
```
np.any( x == -y ) # verifica se pelo menos um é 'True'
```

[32]:
```
True
```

1.7.4 `random`

Exemplo: crie um *array* 1D com 5 números aleatórios entre [0,1].

[33]:
```
r = np.random.rand(5)
r
```

[33]:
```
array([0.29517359, 0.33502619, 0.01222812, 0.04877423, 0.84567382])
```

Exemplo: crie um *array* 1D com 50 números inteiros aleatórios entre [0,7].

[34]:
```
r2 = np.random.randint(0,7+1,50) # menor, maior é 8 (exclusive), tamanho
r2
```

[34]:
```
array([4, 6, 1, 0, 0, 7, 1, 5, 6, 1, 2, 0, 3, 5, 0, 7, 0, 0, 7, 7, 3, 0,
       1, 1, 0, 1, 0, 1, 1, 1, 2, 5, 2, 2, 2, 0, 5, 1, 5, 0, 0, 2, 1, 2,
       5, 7, 0, 2, 4, 5])
```

Exemplo: crie uma matriz m x n com números inteiros aleatórios entre inteiros [l,h].

[35]:
```
def gera_matriz(m,n,l,h):
    return np.random.randint(l,h,(m,n)) # tupla (m,n)
```

[36]:
```
gera_matriz(2,2,0,1)
```

[36]:
```
array([[0, 0],
       [0, 0]])
```

[37]:
```
gera_matriz(3,2,0,4)
```

[37]:
```
array([[2, 2],
       [3, 1],
       [3, 1]])
```

[38]:
```
gera_matriz(4,4,-2,7)
```

[38]:
```
array([[ 4,  6, -2,  3],
       [ 6,  5,  0,  5],
       [ 2,  0,  5,  2],
       [-2,  5, -2, -1]])
```

1.7.5 ones

Criando arrays unitários.

[39]:
```
np.ones(4)
```

[39]:
```
array([1., 1., 1., 1.])
```

[40]:
```
np.ones((3,2)) # tupla necessária para linhas e colunas
```

[40]:
```
array([[1., 1.],
       [1., 1.],
       [1., 1.]])
```

1.7.6 eye

Criando arrays 2D identidade. 1 na diagonal e 0 nas demais.

[41]:
```
np.eye(3) # matriz identidade 3 x 3
```

[41]:
```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

1.7.7 zeros

Arrays nulos.

[42]:
```
np.zeros(3)
```

[42]:
```
array([0., 0., 0.])
```

[43]:
```
np.zeros((2,4)) # 2 x 4
```

[43]:
```
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

1.7.8 `full`

Arrays de valor constante.

[44]:
```
np.full(3,0) # 1 x 3 com constante 0
```

[44]:
```
array([0, 0, 0])
```

[45]:
```
np.full(shape=(3,),fill_value=0)
```

[45]:
```
array([0, 0, 0])
```

[46]:
```
F1 = np.full(shape=(2,2),fill_value=1) # 2 x 2 com 1
F1
```

[46]:
```
array([[1, 1],
       [1, 1]])
```

[47]:
```
F1 == np.ones(2) # mesmo resultado que ones
```

[47]:
```
array([[ True,  True],
       [ True,  True]])
```

Outras maneiras:

[48]:
```
F2 = 3*np.ones((4,4))
F2
```

[48]:
```
array([[3., 3., 3., 3.],
       [3., 3., 3., 3.],
       [3., 3., 3., 3.],
       [3., 3., 3., 3.]])
```

1.8 Especificando tipos de dados

Observe o seguinte exemplo:

[49]:
```
F2
```

[49]:
```
array([[3., 3., 3., 3.],
       [3., 3., 3., 3.],
       [3., 3., 3., 3.],
       [3., 3., 3., 3.]])
```

[50]:
```
F3 = np.full((4,4),3)
F3
```

[50]:
```
array([[3, 3, 3, 3],
       [3, 3, 3, 3],
       [3, 3, 3, 3],
       [3, 3, 3, 3]])
```

[51]:
```
F2 == F3 # valores iguais
```

[51]:
```
array([[ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True]])
```

[52]:
```
F2.dtype == F3.dtype # tipos diferentes
```

[52]:
```
False
```

[53]:
```
F2.dtype
```

[53]:
```
dtype('float64')
```

[54]:
```
F3.dtype
```

[54]:
```
dtype('int64')
```

Especificamos o tipo de dados com dtype.

[55]:
```
np.ones((4,2),dtype=bool) # matriz de booleanos
```

[55]:
```
array([[ True,  True],
       [ True,  True],
       [ True,  True],
       [ True,  True]])
```

[56]:
```
np.ones((4,2),dtype=str) # matriz de strings; 'U1' diz que há no máximo 1␣
↪caracter
```

[56]:
```
array([['1', '1'],
       ['1', '1'],
       ['1', '1'],
       ['1', '1']], dtype='<U1')
```

[57]:
```
S = np.array(['dias','mes','ano'])
S.dtype # 4 é o no. máximo de caracteres nas strings
```

[57]:
```
dtype('<U4')
```

1.9 Indexação e fatiamento

Funcionam de maneira similar ao uso com listas.

[58]:
```
I = np.linspace(0,20,11)
I
```

[58]:
```
array([ 0.,  2.,  4.,  6.,  8., 10., 12., 14., 16., 18., 20.])
```

[59]:
```
I[3],I[2:4],I[5:8],I[-4:-1]
```

[59]:
```
(6.0, array([4., 6.]), array([10., 12., 14.]), array([14., 16., 18.]))
```

[60]:
```
I[::-1] # invertendo o array
```

[60]:
```
array([20., 18., 16., 14., 12., 10.,  8.,  6.,  4.,  2.,  0.])
```

[61]:
```
I2 = np.array([I,2*I,3*I,4*I])
I2
```

[61]:
```
array([[ 0.,  2.,  4.,  6.,  8., 10., 12., 14., 16., 18., 20.],
       [ 0.,  4.,  8., 12., 16., 20., 24., 28., 32., 36., 40.],
       [ 0.,  6., 12., 18., 24., 30., 36., 42., 48., 54., 60.],
       [ 0.,  8., 16., 24., 32., 40., 48., 56., 64., 72., 80.]])
```

Em arrays bidimensionais, a indexação é feita por meio de uma tupla. Porém, a explicitação dos parênteses é desnecessária.

[62]:
```
I2[(2,3)] # 3a. linha; 4a. coluna
```

[62]:
```
18.0
```

[63]:
```
I2[2,3]
```

[63]:
```
18.0
```

[64]:
```
I2[0,:] # 1a. linha
```

[64]:
```
array([ 0.,  2.,  4.,  6.,  8., 10., 12., 14., 16., 18., 20.])
```

[65]:
```
I2[1,:] # 2a. linha
```

[65]:
```
array([ 0.,  4.,  8., 12., 16., 20., 24., 28., 32., 36., 40.])
```

[66]:
```
I2[-1,:] # última linha
```

[66]:
```
array([ 0.,  8., 16., 24., 32., 40., 48., 56., 64., 72., 80.])
```

[67]:
```
I2[:,0] # 1a. coluna
```

[67]:
```
array([0., 0., 0., 0.])
```

[68]:
```
I2[:,1] # 2a. coluna
```

[68]:
```
array([2., 4., 6., 8.])
```

[69]:
```
I2[:,8] # 9a. coluna
```

[69]:
```
array([16., 32., 48., 64.])
```

[70]:
```
I2[:,2:4] # 3a. - 4a. coluna
```

[70]:
```
array([[ 4.,  6.],
       [ 8., 12.],
       [12., 18.],
       [16., 24.]])
```

[71]:
```
I2[1:3,6:10] # submatriz: linhas 2 - 3; 7-10
```

[71]:
```
array([[24., 28., 32., 36.],
       [36., 42., 48., 54.]])
```

1.9.1 Alteração de valores

Os arrays são mutáveis por indexação.

[72]:
```
A3 = np.random.rand(4,4)
A3
```

[72]:
```
array([[0.99358221, 0.91631058, 0.78539578, 0.01297965],
       [0.62793278, 0.10920324, 0.1335377 , 0.1296766 ],
       [0.43640906, 0.58841502, 0.38486414, 0.83868661],
       [0.68593655, 0.56702575, 0.69449312, 0.11953334]])
```

[73]:
```
A3[0:4,0] = -1
A3
```

[73]:
```
array([[-1.        ,  0.91631058,  0.78539578,  0.01297965],
       [-1.        ,  0.10920324,  0.1335377 ,  0.1296766 ],
       [-1.        ,  0.58841502,  0.38486414,  0.83868661],
       [-1.        ,  0.56702575,  0.69449312,  0.11953334]])
```

[74]:
```
A3[:,-1] = -1
A3
```

[74]:
```
array([[-1.        ,  0.91631058,  0.78539578, -1.        ],
       [-1.        ,  0.10920324,  0.1335377 , -1.        ],
```

```
       [-1.        ,  0.58841502,  0.38486414, -1.        ],
       [-1.        ,  0.56702575,  0.69449312, -1.        ]])
```

[75]:
```
A3[1:3,1:3] = 0
A3
```

[75]:
```
array([[-1.        ,  0.91631058,  0.78539578, -1.        ],
       [-1.        ,  0.        ,  0.        , -1.        ],
       [-1.        ,  0.        ,  0.        , -1.        ],
       [-1.        ,  0.56702575,  0.69449312, -1.        ]])
```

Podemos alterar valores com arrays.

[76]:
```
A3[1,:] = -2*np.ones(4)
A3
```

[76]:
```
array([[-1.        ,  0.91631058,  0.78539578, -1.        ],
       [-2.        , -2.        , -2.        , -2.        ],
       [-1.        ,  0.        ,  0.        , -1.        ],
       [-1.        ,  0.56702575,  0.69449312, -1.        ]])
```

A indexação pode usar um comprimento de passo (*step*).

[77]:
```
A3[0:4:3,1:3] = np.full((1,2),8) # na indexação esquerda, 1a. linha : 4a. linha :
 ↪ step de 3
A3
```

[77]:
```
array([[-1.,  8.,  8., -1.],
       [-2., -2., -2., -2.],
       [-1.,  0.,  0., -1.],
       [-1.,  8.,  8., -1.]])
```

1.9.2 `newaxis`

`newaxis` é uma instância do `numpy` que permite aumentar de 1 a dimensão de um array existente.

Exemplo: como inserir a diagonal de uma matriz em uma segunda matriz como uma coluna adicional?

Criamos duas matrizes aleatórias.

[78]:
```
# matriz 4 x 4 de inteiros aleatórios entre 0 e 9
B1 = np.random.randint(0,10,(4,4))
B1
```

[78]:
```
array([[4, 1, 8, 5],
       [2, 5, 6, 2],
       [5, 4, 7, 8],
       [4, 9, 7, 9]])
```

[79]:
```
# matriz 4 x 4 de inteiros aleatórios entre -10 e 9
B2 = np.random.randint(-10,10,(4,4))
B2
```

[79]:
```
array([[ 9, -7, -2, -6],
       [-7,  3,  6,  3],
       [ 8,  4, -8, -3],
       [ 1,  6,  5,  9]])
```

Extraímos a diagonal da primeira.

[80]:
```
# diagonal de B1
db1 = np.diag(B1)
db1
```

[80]:
```
array([4, 5, 7, 9])
```

Notemos agora que as dimensões são diferentes.

[81]:
```
print(B2.ndim)
print(db1.ndim)
```

```
2
1
```

Para podermos aglutinar a diagonal como uma nova coluna na primeira matriz, primeiro temos que transformar o array unidimensional para uma matriz.

[82]:
```
db1 = db1[:,np.newaxis]
print(db1.ndim) # agora o array é bidimensional
db1
```

```
2
```

[82]:
```
array([[4],
       [5],
       [7],
       [9]])
```

`newaxis` é um "eixo imaginário" incluído *inplace*, mas que altera dinamicamente o array. No caso acima, o array tornou-se em uma coluna.

Agora, podemos "colar" um array 2D com outro por uma concatenação.

1.9.3 `concatenate`

`concatenate` é usado para concatenar *arrays*. A concatenação requer uma tupla contendo os *arrays* a concatenar e o eixo de referência.

[83]:
```
B3 = np.concatenate((B2,db1), axis=1)
B3
```

[83]:
```
array([[ 9, -7, -2, -6,  4],
       [-7,  3,  6,  3,  5],
       [ 8,  4, -8, -3,  7],
       [ 1,  6,  5,  9,  9]])
```

No caso acima, `axis=1` indica que a concatenação é ao longo da coluna. Dessa forma, inserimos a segunda diagonal como uma coluna adicional na segunda matriz. Claramente, isto só é possível porque ambas as matrizes eram de mesmo formato.

`axis` Nos arrays multidimensionais do Python, `axis` é usado para indicar a "direção" dos dados. Em arrays bidimensionais, `axis=0` refere-se à direção de cima para baixo (ao longo das linhas), ao passo que `axis=1` refere-se à direção da esquerda para a direita (ao longo das colunas).

Obs.: note que a palavra `axis` ("eixo") deve ser usada, e não "axes" ("eixos").

Para aglutinar uma linha na matriz anterior, fazemos uma concatenação em linha.

[84]:
```
# array de zeros com mesmo número de colunas de B3
db2 = np.zeros(np.shape(B3)[1])
db2
```

[84]:
```
array([0., 0., 0., 0., 0.])
```

[85]:
```
db2 = db2[np.newaxis,:] # cria o "eixo imaginário" na direção 0
```

[86]:
```
B4 = np.concatenate((B3,db2),axis=0) # concatena ao longo das linhas
B4
```

[86]:
```
array([[ 9., -7., -2., -6.,  4.],
       [-7.,  3.,  6.,  3.,  5.],
       [ 8.,  4., -8., -3.,  7.],
       [ 1.,  6.,  5.,  9.,  9.],
       [ 0.,  0.,  0.,  0.,  0.]])
```

1.10 Indexação avançada

Podemos usar máscaras como artifícios para indexação avançada.

[87]:
```
IA1 = np.arange(-10,11)
IA1
```

[87]:
```
array([-10,  -9,  -8,  -7,  -6,  -5,  -4,  -3,  -2,  -1,   0,   1,   2,
         3,   4,   5,   6,   7,   8,   9,  10])
```

Vamos criar um *array* aleatório de True e False no mesmo formato que o *array* anterior.

[88]:
```
mask1 = np.random.randint(0,2,np.shape(IA1),dtype=bool)
mask1
```

[88]:
```
array([False, False, False, False, False,  True, False, False,  True,
       False,  True, False, False, False,  True, False,  True, False,
        True,  True, False])
```

Esta *máscara booleana* pode ser aplicada no array para extrair apenas os elementos cujos índices são marcados como `True` pela máscara.

[89]:
```
IA1[mask1]
```

[89]:
```
array([-5, -2,  0,  4,  6,  8,  9])
```

Há maneiras mais diretas aplicáveis a filtragens. Para extrair os valores negativos do array:

[90]:
```
IA1 < 0 # máscara booleana
```

[90]:
```
array([ True,  True,  True,  True,  True,  True,  True,  True,  True,
        True, False, False, False, False, False, False, False, False,
       False, False, False])
```

[91]:
```
IA1[IA1 < 0]
```

[91]:
```
array([-10,  -9,  -8,  -7,  -6,  -5,  -4,  -3,  -2,  -1])
```

Para extrair os valores positivos do array:

[92]:
```
IA1[IA1 > 0] # máscara booleana para positivos
```

[92]:
```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

Para extrair os valores no intervalo $]-2,5[$, fazemos:

[93]:
```
IA1[(IA1 > -2) & (IA1 < 5)] # & é o operador booleano 'elemento a elemento'
```

[93]:
```
array([-1,  0,  1,  2,  3,  4])
```

Para extrair pares e ímpares, poderíamos fazer:

[94]:
```
pares, impares = IA1[IA1 % 2 == 0] , IA1[IA1 % 2 == 1]
pares,impares
```

[94]:
```
(array([-10,  -8,  -6,  -4,  -2,   0,   2,   4,   6,   8,  10]),
 array([-9, -7, -5, -3, -1,  1,  3,  5,  7,  9]))
```

Podemos usar listas como máscaras:

[95]:
```
alguns = pares[[0,2,3,5]] # acessa 1o., 3o. 4o. e 6o. elemento de 'pares'
```

[96]:
```
impares[alguns] # estude este caso
```

[96]:
```
array([-9, -1,  3, -9])
```

No caso acima, por exemplo, -10 é uma indexação reversa que excede o compriemento do array à esquerda. Portanto, ele retorna o primeiro elemento do array, que é -9. O índice -6 acessa o sexto elemento a partir da direita, que é -1. O índice -4 acessa o quarto elemento a partir da direita. Por fim, o índice 0 acessa o primeiro elemento que é -9.

1.11 Operações elemento a elemento

As operações aritméticas e de cálculo são feitas elemento a elemento nos *arrays*. Já mostramos alguns exemplos acima, mas vamos tornar isto mais claro aqui.

[97]:
```
a = np.array([1,2,3])
b = np.array([4,5,6])
```

[98]:
```
# operações elemento a elemento
print(a + b)
print(a - b)
print(a * b)
print(a / b)
print(a ** b)
```

```
[5 7 9]
[-3 -3 -3]
[ 4 10 18]
[0.25 0.4  0.5 ]
[  1  32 729]
```

[99]:
```
2*a + 4*b - 6*b**2 + 1.1/2*a
```

[99]:
```
array([ -77.45, -124.9 , -184.35])
```

1.12 Funções matemáticas

O `numpy` possui a maioria dass funções disponíveis no módulo `math` e outras mais. As funções são diretamente aplicáveis aos *arrays*. Lembre-se que para fazer o mesmo usando em listas, tínhamos de construir meios de iterar sobre elas e aplicar a função a cada elemento por vez. Isto não é mais necessário com o `numpy`. Eis a beleza da computação vetorizada!

Vejamos uma série de exemplos.

[100]:
```
x = np.arange(10)
x
```

[100]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

[101]:
```
np.sqrt(x)
```

[101]:
```
array([0.        , 1.        , 1.41421356, 1.73205081, 2.        ,
       2.23606798, 2.44948974, 2.64575131, 2.82842712, 3.        ])
```

[102]:
```
np.cos(x) + 2*np.sqrt(x)
```

[102]:
```
array([1.        , 2.54030231, 2.41228029, 2.47410912, 3.34635638,
       4.75579814, 5.85914977, 6.04540488, 5.51135422, 5.08886974])
```

[103]:
```
y = np.sin(2*x)
z = np.exp(x + y)
y - z
```

[103]:
```
array([-1.00000000e+00, -5.83904850e+00, -4.22348298e+00, -1.54686133e+01,
       -1.45852799e+02, -8.66844526e+01, -2.36441078e+02, -2.95209938e+03,
       -2.23551181e+03, -3.82459986e+03])
```

1.12.1 Problema resolvido (Laboratório Computacional 1C)

Observe a tabela a seguir, onde **DS (UA)** é a distância do referido planeta do até o Sol em unidades astronômicas (UA), **Tm (F)** sua temperatura superficial mínima em graus Farenheit e **TM (F)** sua temperatura superficial máxima em graus Farenheit.

	DS (UA)	Tm (F)	TM (F)	DS (km)	TmM (C)
Mercúrio	0.39	-275	840	?	?
Vênus	0.723	870	870	?	?
Terra	1.0	-129	136	?	?
Marte	1.524	-195	70	?	?

- Escreva um código para converter a temperatura dos planetas de graus Farenheit (**F**) para Celsius (**C**).
- Escreva um código para converter unidades astronômicas em quilômetros.
- Imprima os valores que deveriam ser inseridos na coluna **DS (km)** horizontalmente usando `print`.
- Repita o item anterior para a coluna **TmM (C)**, que é a média aritmética entre **Tm** e **TM**.

Observação: use notação científica (exemplo: 4.2×10^8 pode ser escrito como `4.2e8` em Python).

Resolução Há várias maneiras de resolver. Aqui apresentamos uma estratégia com `lambdas`.

- Montar os arrays dos dados numéricos.

[104]:
```
DS = np.array([0.39,0.723,1.0,1.524])
Tm = np.array([-275,870,-129,-195])
TM = np.array([840,870,136,70])
```

- Fórmula e cálculo da conversão Farenheit para Celsius:

[105]:
```
C = lambda F: 5/9*(F-32)
CTm = C(Tm)
CTM = C(TM)
print(CTm) # minimas em C
print(CTM) # maximas em C
```

```
[-170.55555556  465.55555556  -89.44444444 -126.11111111]
[448.88888889 465.55555556  57.77777778  21.11111111]
```

- Fórmula e cálculo da conversão UA para km:

[106]:
```
UA = lambda km: 1.496e+8*km
UADS = UA(DS)
print(UADS) # valores a inserir
```

```
[5.834400e+07 1.081608e+08 1.496000e+08 2.279904e+08]
```

- Cálculo da média

[107]:
```
TmM = 0.5*(CTm + CTM)
print(TmM)
```

```
[139.16666667 465.55555556 -15.83333333 -52.5       ]
```

1.12.2 `reshape` e `hstack`

A montagem do array bidimensional com os cálculos resultantes não foi requisitada no problema. Porém, vamos mostrar uma maneira de fazer isto usando `reshape`, que é uma função utilizada para reformatar os dados e `hstack`, que é usada para "empilhar" arrays horizontalmente.

Note que todos os nossos *arrays* são unidimensionais. Vamos torná-los bidimensionais com formato 4 x 1 e empilhá-los horizontalmente, isto é, na direção do eixo 1 (esquerda para direita).

Obs: consulte também `vstack`.

[108]:
```
todos = [DS,CTm,CTM,UADS,TmM] # lista com todos os arrays

for i,ar in enumerate(todos):
    todos[i] = np.reshape(ar, (4,1)) # reformata

final = np.hstack(todos) # empilha
```

Explicando o que fizemos:

- Colocamos todos os arrays em uma lista: neste ponto, nada novo.
- Iteramos sobre a lista, reformatamos um por um e reatribuímos na mesma lista como arrays bidimensionais

Para o segundo ponto, observe:

```
[109]: DS.shape # formato é 1 x 4 (unidimensional)
```

```
[109]: (4,)
```

```
[110]: np.reshape(DS,(4,1)) # reformata
```

```
[110]: array([[0.39 ],
              [0.723],
              [1.   ],
              [1.524]])
```

```
[111]: np.reshape(DS,(4,1)).shape # novo formato é 4 x 1
```

```
[111]: (4, 1)
```

```
[112]: np.reshape(DS,(4,1)).ndim # o array agora é bidimensional
```

```
[112]: 2
```

Procedendo assim para todos, conseguimos reformatá-los e adicioná-los em uma lista. Se desejarmos, podemos sobrescrever essa lista ou não. Na resolução anterior, escolhemos sobrescrever. Assim, suponha que a lista dos arrays reformatados seja:

```
[113]: L = [np.reshape(DS,(4,1)),np.reshape(TmM,(4,1))] # apenas DS e TmM
       L
```

```
[113]: [array([[0.39 ],
               [0.723],
               [1.   ],
               [1.524]]),
        array([[139.16666667],
               [465.55555556],
               [-15.83333333],
               [-52.5       ]])]
```

- Criamos o array final por empilhamento.

Note que a lista `L` possui 2 arrays de formato 4 x 1. Para criar o array 4 x 2, faremos um empilhamento horizontal similar à uma concatenação na direção 1.

```
[114]: Lh = np.hstack(L)
       Lh
```

```
[114]: array([[ 3.90000000e-01,  1.39166667e+02],
              [ 7.23000000e-01,  4.65555556e+02],
              [ 1.00000000e+00, -1.58333333e+01],
              [ 1.52400000e+00, -5.25000000e+01]])
```

Agora podemos verificar que, de fato, o array está na forma como queremos.

[115]:
```
Lh[:,0] # 1a. coluna idêntica à DS
```

[115]:
```
array([0.39 , 0.723, 1.   , 1.524])
```

[116]:
```
Lh[:,0] == DS # teste
```

[116]:
```
array([ True,  True,  True,  True])
```

[117]:
```
np.all( Lh[:,0] == DS ) # teste completo
```

[117]:
```
True
```

[118]:
```
Lh[:,1] # 2a. coluna idêntica a TmM
```

[118]:
```
array([139.16666667, 465.55555556, -15.83333333, -52.5       ])
```

[119]:
```
Lh[:,1] == TmM # teste
```

[119]:
```
array([ True,  True,  True,  True])
```

[120]:
```
np.all( Lh[:,1] == TmM ) # teste completo
```

[120]:
```
True
```

1.13 *Broadcasting*

Broadcasting é a capacidade que o *numpy* oferece para realizarmos operações em arrays com diferentes dimensões.

1.13.1 Regras do *broadcasting*

1. Se dois *arrays* tiverem dimensões diferentes, o formato do array com menor dimensão é preenchido por 1 do lado esquerdo.
2. Se o formato dos *arrays* não for igual em dimensão alguma, o array com tamanho igual a 1 é esticado nesta direção para ficar no mesmo tamanho correspondente do outro array.
3. Se em qualquer direção os tamanhos dos *arrays* forem diferentes e nenhum deles for igual a 1, então um erro é retornado.

Exemplo da Regra 1

[121]:
```
A = np.array([[1, 2, 3],[4, 5, 6]]) # array 2D
b = np.array([10, 20, 30]) # array 1D
print(A.shape)
print(b.shape)
```

```
(2, 3)
(3,)
```

[122]:
```
A + b
```

[122]:
```
array([[11, 22, 33],
       [14, 25, 36]])
```

A soma pode ser realizada mesmo assim. O que ocorreu? Cada linha de `A` foi somada à única linha de `b`. O *broadcasting* amplia o array de menor dimensão automaticamente da seguinte forma:

Pela regra 1, o *array* `b` tem dimensão menor. Então, ele é preenchido de modo que:

```
A.shape -> (2, 3)
b.shape -> (1, 3)
```

Pela regra 2, a primeira dimensão de `A` é 2 e a de `b` é 1. Então, a dimensão de `b` é "esticada", de modo que:

```
A.shape -> (2, 3)
b.shape -> (2, 3)
```

A mesma operação poderia ter sido feita com:

[123]:
```
A + np.array([b,b])
```

[123]:
```
array([[11, 22, 33],
       [14, 25, 36]])
```

Exemplo da Regra 2

[124]:
```
A = np.arange(3).reshape((3, 1))
b = np.arange(3)
print(A.shape)
print(b.shape)
```

```
(3, 1)
(3,)
```

[125]:
```
A + b
```

[125]:
```
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4]])
```

Neste caso, ambos os arrays sofrem *broadcasting*. Ele ocorre da seguinte forma.

Como

```
A.shape = (3, 1)
b.shape = (3,)
```

a regra 1 diz que `b` deve ser preenchido de modo que

```
A.shape -> (3, 1)
b.shape -> (1, 3)
```

e, pela regra 2, cada uma das dimensões 1 deve ser alterada de modo que:

```
A.shape -> (3, 3)
b.shape -> (3, 3)
```

Assim, o *broadcasting* é permitido.

Exemplo da Regra 3

[126]:
```
A = np.ones((3, 2))
b = np.arange(3)
print(A.shape)
print(b.shape)
```

```
(3, 2)
(3,)
```

[127]:
```
A + b
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-127-48207f55069c> in <module>
----> 1 A + b


ValueError: operands could not be broadcast together with shapes (3,2) (3,)
```

Neste exemplo, o *broadcasting* não é permitido. O caso é levemente diferente do primeiro exemplo em que `A` é transposta.

Temos que

```
M.shape = (3, 2)
a.shape = (3,)
```

Pela regra 1, devemos ter

```
M.shape -> (3, 2)
a.shape -> (1, 3)
```

e, pela regra 2, a primeira dimensão deve ser esticada para combinar-se com a de `A` enquanto a segunda não é alterada por não ser 1.

```
M.shape -> (3, 2)
a.shape -> (3, 3)
```

Porém, o formato final de ambos não se combina. Sendo incompatíveis, o *broadcasting* falha.