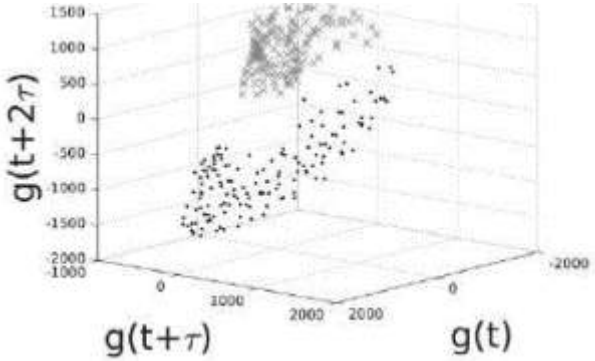
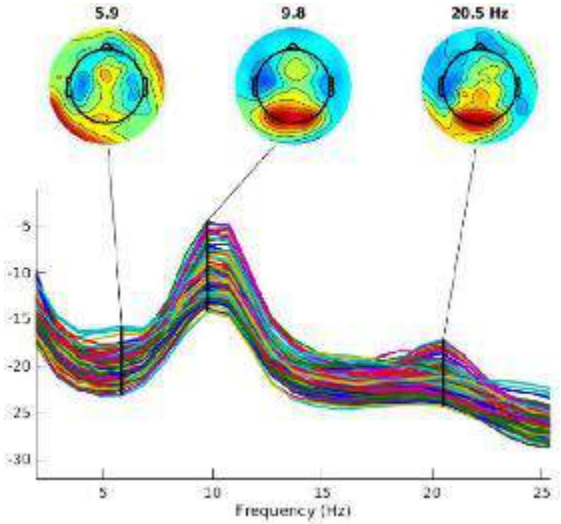




UNICAMP

Instituto de Computação

MC102: Algoritmos e Programação de Computadores (turmas 4,5,6,7)
Extra – Cálculos numéricos: pacote NumPy

ARTHUR VALENCIO

Pós-doutorando IC/Unicamp

Fapesp CEPID NeuroMat

Campinas, 16 de Maio de 2020

>>NumPy

- Em 1995 foi desenvolvido por um um time do MIT e LLNL uma biblioteca para trabalhar com problemas matemáticos envolvendo matrizes e vetores n-dimensionais chamada ***Numeric***. O pacote foi expandindo em diferentes versões paralelas, englobando várias áreas de interesse da matemática, física e engenharias
- Em 2005, buscou-se unificar as versões e apresentar como pacotes padrão do Python:
  - **Assim nasceu o *NumPy*, que trata das operações básicas, mais relacionadas com a estrutura dos dados, a computações relacionas à álgebra linear**
  - **E também o pacote *SciPy*, que trata das operações matemáticas mais complexas (integração, transformadas, funções estatísticas, etc)**

>>NumPy

- A estrutura de dados que o NumPy trabalha é do tipo ***array***, que é bem parecida com uma lista, com o detalhe de que todos os elementos são do mesmo tipo, com precisão em bits definida pelo usuário.
- Isso, associado à otimizações internas, permitem com que arrays ocupem menos espaço de memória e operações com elas sejam bem mais rápidas do que com listas

>>NumPy

- A estrutura de dados que o NumPy trabalha é do tipo ***array***, que é bem parecida com uma lista, com o detalhe de que todos os elementos são do mesmo tipo, com precisão em bits definida pelo usuário.
- Isso, associado à otimizações internas, permitem com que arrays ocupem menos espaço de memória e operações com elas sejam bem mais rápidas do que com listas

>>Construindo uma array

```
import numpy as np
```

```
a=np.array( [1, 2, 3, 4] )
```

Constrói uma array com estes elementos inteiros

```
b=np.array( [1.0, 3.5, 2.2] )
```

Constrói uma array com estes elementos float

Em ambos os casos os elementos possuem 64 bits de precisão

>>Construindo uma array

```
import numpy as np

a=np.array( [1, 2, 3, 4] )
b=np.array( [1.0, 3.5, 2.2] )

c=np.array( [1, 2, 3, 4], dtype=np.int16 )
d=np.array( [1.0, 3.5, 2.2], dtype=np.float16 )
```

Utilizando o parâmetro dtype, podemos especificar a precisão a ser utilizada.
No caso, estes especificam 16bits de precisão para os elementos.
Isso aumenta os erros de arrendondamento, porém reduz consideravelmente o consumo de memória e aumenta a velocidade de processamento

>>Construindo uma array

```
import numpy as np

a=np.array( [1, 2, 3, 4] )
b=np.array( [1.0, 3.5, 2.2] )

c=np.array( [1, 2, 3, 4], dtype=np.int16 )
d=np.array( [1.0, 3.5, 2.2], dtype=np.float16 )

e=np.zeros( (3, 4) )
```

Cria uma matriz com 3 linhas e 4 colunas onde todos os elementos são zeros

>>Construindo uma array

```
import numpy as np

a=np.array( [1, 2, 3, 4] )
b=np.array( [1.0, 3.5, 2.2] )

c=np.array( [1, 2, 3, 4], dtype=np.int16 )
d=np.array( [1.0, 3.5, 2.2], dtype=np.float16 )

e=np.zeros( (3, 4) )
f=np.ones( (4, 5) )
```

Cria uma matriz com 4 linhas e 5 colunas onde todos os elementos são 1

>>Construindo uma array

```
import numpy as np

a=np.array( [1, 2, 3, 4] )
b=np.array( [1.0, 3.5, 2.2] )

c=np.array( [1, 2, 3, 4], dtype=np.int16 )
d=np.array( [1.0, 3.5, 2.2], dtype=np.float16 )

e=np.zeros( (3, 4) )
f=np.ones( (4, 5) )
g=np.empty( (6, 3) )
```

Cria uma matriz com 6 linhas e 3 colunas, mas os elementos ainda não foram atribuídos

Cuidado! O valor guardado nos elementos dessa matriz são *lixos de memória* de outros processos do seu computador. Se inicializar um matriz com *np.empty*, certifique-se de que todos os elemntos serão atribuidos em seguida

>>Construindo uma array

```
import numpy as np

a=np.array( [1, 2, 3, 4] )
b=np.array( [1.0, 3.5, 2.2] )

c=np.array( [1, 2, 3, 4], dtype=np.int16 )
d=np.array( [1.0, 3.5, 2.2], dtype=np.float16 )

e=np.zeros( (3, 4) )
f=np.ones( (4, 5) )
g=np.empty( (6, 3) )

h1=np.arange( (10, 30, 0.3) )
```

Cria uma sequência, de 10 à 30, pulando em passos de 0.3.
Isto é:
10, 10.3, 10.6, 10.9, 11.2, ...

arange = **array range**, por isso você inclui os parâmetros na mesma forma que faria com a função range

>>Construindo uma array

```
import numpy as np

a=np.array( [1, 2, 3, 4] )
b=np.array( [1.0, 3.5, 2.2] )

c=np.array( [1, 2, 3, 4], dtype=np.int16 )
d=np.array( [1.0, 3.5, 2.2], dtype=np.float16 )

e=np.zeros( (3, 4) )
f=np.ones( (4, 5) )
g=np.empty( (6, 3) )

h1=np.arange(10, 30, 0.3)
h2=np.linspace(2, 10, 5)
```

Cria uma sequência, de 2 até 10, contendo 5 elementos.
Isto é:
2.0, 4.0, 6.0, 8.0, 10.0

>>Construindo uma array de números aleatórios

```
import numpy as np

a=np.random.rand( 3, 2 )

b=np.random.randint( 0, high=10, size=(5,3) )

c=np.random.randn( 3, 7 )
```

Cria array com 3 linhas e 2 colunas de valores gerados a partir de distribuição **uniforme** entre 0 e 1

Cria array com 5 linhas e 3 colunas de valores **inteiros** gerados a partir de distribuição uniforme entre 0 e 10

Cria array com 3 linhas e 7 colunas de valores gerados a partir de distribuição **normal** entre 0 e 1

>>Construindo uma array de dados temporais

NumPy inclui um tipo de objeto chamado datetime64:

```
import numpy as np

a=np.datetime64( '2020-05-15')

b=np.datetime64( '2020-05-15T13:15:30.231')

c=np.array(['2020-05-15', '2020-05-16' , '2020-05-17'], dtype='datetime64')

c[2]-c[0]
```

Cria um objeto datetime informando uma data

Cria um objeto datetime informando uma data, com horário especificado (Até milisegundos se usuário desejar)

Cria um array com 3 datas

Calcula o tempo entre o terceiro e o primeiro item do array, devolvendo um objeto do tipo *timedelta* com valor (2, 'D') para representar 2 dias

>>Importando array de arquivo

```
import numpy as np

a=np.fromfile(r“pasta_do_Arquivo/Arquivo”, dtype=float, count=-1, sep=“ ”)
```

Especifica o tipo dos elementos

Número de itens a serem lidos (opcional). Escolher -1 significa “ler todos”

Separador entre os números (em geral espaço ou vírgula)

>>Constantes especiais

```
import numpy as np

np.pi  # 3.141592653589793

np.e # 2.718281828459045

np.euler_gamma # 0.5772156649015329

np.inf # +∞

np.NINF # -∞

np.nan # not-a-number
```

>>Método reshape

```
import numpy as np
```

```
A=np.arange(20)
```

Nesse momento temos:
A=[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19]

```
A=A.reshape(4,5)
```

Agora:

$$A = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 & 9 \\ 10 & 11 & 12 & 13 & 14 \\ 15 & 16 & 17 & 18 & 19 \end{bmatrix}$$

>>Método reshape

```
import numpy as np

A=np.arange(20)
```

Nesse momento temos:
A=[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19]

```
A=A.reshape(4,-1)
```

Utilizando -1 o reshape calcula automaticamente qual deve ser o tamanho para caber todos os elementos

Agora:

$$A = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 & 9 \\ 10 & 11 & 12 & 13 & 14 \\ 15 & 16 & 17 & 18 & 19 \end{bmatrix}$$

>>Operações elemento à elemento

$$A = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

$$B = \begin{bmatrix} 5 & 6 & 7 \\ 8 & 9 & 10 \\ 11 & 12 & 13 \end{bmatrix}$$

O uso dos operadores aritméticose relacionais que vimos anteriormente podem ser aplicados aos **arrays** construídos com o pacote **NumPy**.

Neste caso, as operações são realizadas elemento à elemento:

```
import numpy as np
A=np.arange(0,9).reshape(3,3)
B=np.arange(5,14).reshape(3,3)
```

>>Operações elemento à elemento

O uso dos operadores aritméticose relacionais que vimos anteriormente podem ser aplicados aos **arrays** construídos com o pacote **NumPy**.

Neste caso, as operações são realizadas elemento à elemento:

```
import numpy as np
A=np.arange(0,9).reshape(3,3)
B=np.arange(5,14).reshape(3,3)
```

$$A = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

$$B = \begin{bmatrix} 5 & 6 & 7 \\ 8 & 9 & 10 \\ 11 & 12 & 13 \end{bmatrix}$$

A+B →

$$\begin{bmatrix} 5 & 7 & 9 \\ 11 & 13 & 15 \\ 17 & 19 & 21 \end{bmatrix}$$

>>Operações elemento à elemento

O uso dos operadores aritméticose relacionais que vimos anteriormente podem ser aplicados aos **arrays** construídos com o pacote **NumPy**.

Neste caso, as operações são realizadas elemento à elemento:

```
import numpy as np
A=np.arange(0,9).reshape(3,3)
B=np.arange(5,14).reshape(3,3)
```

$$A = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

$$B = \begin{bmatrix} 5 & 6 & 7 \\ 8 & 9 & 10 \\ 11 & 12 & 13 \end{bmatrix}$$

A+B

A*B →

$$\begin{bmatrix} 0 & 6 & 14 \\ 24 & 36 & 50 \\ 66 & 84 & 104 \end{bmatrix}$$

>>Operações elemento à elemento

O uso dos operadores aritméticose relacionais que vimos anteriormente podem ser aplicados aos **arrays** construídos com o pacote **NumPy**.

Neste caso, as operações são realizadas elemento à elemento:

```
import numpy as np
A=np.arange(0,9).reshape(3,3)
B=np.arange(5,14).reshape(3,3)
A+B
A*B
A<B
```

$$A = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

$$B = \begin{bmatrix} 5 & 6 & 7 \\ 8 & 9 & 10 \\ 11 & 12 & 13 \end{bmatrix}$$

$$\begin{bmatrix} \text{True} & \text{True} & \text{True} \\ \text{True} & \text{True} & \text{True} \\ \text{True} & \text{True} & \text{True} \end{bmatrix}$$

>>Operações elemento à elemento

$$A = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

Operações matemáticas usuais (“universal functions”) também são realizadas elemento à elemento:

```
import numpy as np
A=np.arange(0,9).reshape(3,3)
```

np.sqrt(A)

$$\begin{bmatrix} 0 & 1.0 & 1.41 \\ 1.73 & 2.0 & 2.24 \\ 2.45 & 2.65 & 2.83 \end{bmatrix}$$

np.exp(A)

$$\begin{bmatrix} 1 & 2.72 & 7.39 \\ 20.09 & 54.60 & 148.41 \\ 403.43 & 1096.66 & 2980.96 \end{bmatrix}$$

np.sin(A)

$$\begin{bmatrix} 0 & 0.84 & 0.91 \\ 0.14 & -0.76 & -0.96 \\ -0.28 & 0.66 & 0.99 \end{bmatrix}$$

>>Operações matriciais

Utilizam-se métodos para realizar operações mariciais de interesse. A exceção é o produto matricial que possui o atalho @

$$A = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

$$B = \begin{bmatrix} 5 & 6 & 7 \\ 8 & 9 & 10 \\ 11 & 12 & 13 \end{bmatrix}$$

```
import numpy as np
A=np.arange(0,9).reshape(3,3)
B=np.arange(5,14).reshape(3,3)

#produto matricial:
A@B
#ou:
A.dot(B)
```

$$\begin{bmatrix} 30 & 33 & 36 \\ 102 & 114 & 126 \\ 174 & 195 & 216 \end{bmatrix}$$

>>Operações matriciais

Utilizam-se métodos para realizar operações mariciais de interesse. A exceção é o produto matricial que possui o atalho @

$$A = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

$$B = \begin{bmatrix} 5 & 6 & 7 \\ 8 & 9 & 10 \\ 11 & 12 & 13 \end{bmatrix}$$

```
import numpy as np
A=np.arange(0,9).reshape(3,3)
B=np.arange(5,14).reshape(3,3)

#produto matricial:
A@B
#ou:
A.dot(B)

#transposta:
A.T
```

$$\begin{bmatrix} 0 & 3 & 6 \\ 1 & 4 & 7 \\ 2 & 5 & 8 \end{bmatrix}$$

>>Operações matriciais

$$A = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

$$B = \begin{bmatrix} 5 & 6 & 7 \\ 8 & 9 & 10 \\ 11 & 12 & 13 \end{bmatrix}$$

Utilizam-se métodos para realizar operações mariciais de interesse. A exceção é o produto matricial que possui o atalho @

```
import numpy as np
A=np.arange(0,9).reshape(3,3)
B=np.arange(5,14).reshape(3,3)

#produto matricial:
A@B
#ou:
A.dot(B)

#transposta:
A.T

#traço:
A.trace()
B.trace()
```

A.trace() → 12

B.trace() → 27

>>Álgebra Linear

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

$$B = \begin{bmatrix} 5 & 6 & 7 \\ 8 & 9 & 10 \\ 11 & 12 & 13 \end{bmatrix}$$

```
import numpy as np
A=np.arange(1,10).reshape(3,3)
B=np.arange(5,14).reshape(3,3)

#determinante:
np.linalg.det(a)
#autovalores e autovetores:
np.linalg.eig(a)
```

→ ~0 (exceto pequeno erro de arredondamento)

- Array com autovalores: 16.12, -1.12, ~0
- Array com cada autovetor sendo uma das linhas:

$$\begin{bmatrix} -0.23 & -0.79 & 0.41 \\ -0.53 & -0.09 & -0.82 \\ -0.82 & 0.61 & 0.41 \end{bmatrix}$$

>>Álgebra Linear

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

$$B = \begin{bmatrix} 5 & 6 & 7 \\ 8 & 9 & 10 \\ 11 & 12 & 13 \end{bmatrix}$$

```
import numpy as np
A=np.arange(1,10).reshape(3,3)
B=np.arange(5,14).reshape(3,3)

#determinante:
np.linalg.det(a)
#autovalores e autovetores:
np.linalg.eig(a)
#norma:
np.linalg.norm(a)
#potência:
np.linalg.matrixpower(a,5)
```

→ 16.88

$$\begin{bmatrix} 121824 & 149688 & 177552 \\ 275886 & 338985 & 402084 \\ 429948 & 528282 & 626616 \end{bmatrix}$$

>>Álgebra Linear

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

$$B = \begin{bmatrix} 5 & 6 & 7 \\ 8 & 9 & 10 \\ 11 & 12 & 13 \end{bmatrix}$$

```
import numpy as np
A=np.arange(1,10).reshape(3,3)
B=np.arange(5,14).reshape(3,3)

#determinante:
np.linalg.det(a)
#autovalores e autovetores:
np.linalg.eig(a)
#norma:
np.linalg.norm(a)
#potência:
np.linalg.matrixpower(a,5)
#decomposição QR
#(A=QR, sendo Q ortogonal e R triangular):
np.linalg.qr(a)
```

Dois arrays, sendo o primeiro:

$$Q = \begin{bmatrix} -0.12 & 0.90 & 0.41 \\ -0.49 & 0.30 & -0.82 \\ -0.86 & -0.30 & 0.41 \end{bmatrix}$$

E o segundo:

$$R = \begin{bmatrix} -8.12 & -9.60 & 11.08 \\ 0 & 90.45 & 1.81 \\ 0 & 0 & \sim 0 \end{bmatrix}$$

>>Álgebra Linear

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

$$B = \begin{bmatrix} 5 & 6 & 7 \\ 8 & 9 & 10 \\ 11 & 12 & 13 \end{bmatrix}$$

```
import numpy as np
A=np.arange(1,10).reshape(3,3)
B=np.arange(5,14).reshape(3,3)

#determinante:
np.linalg.det(a)
#autovalores e autovetores:
np.linalg.eig(a)
#norma:
np.linalg.norm(a)
#potência:
np.linalg.matrixpower(a,5)
#decomposição QR
#(A=QR, sendo Q ortogonal e R triangular):
np.linalg.qr(a)
#inversa e pseudo-inversa:
np.linalg.inv(a)
np.linalg.pinv(a)
```

Depende de condições estritas de inversibilidade

Mais geral

>>Álgebra Linear

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

$$B = \begin{bmatrix} 5 & 6 & 7 \\ 8 & 9 & 10 \\ 11 & 12 & 13 \end{bmatrix}$$

```
import numpy as np
A=np.arange(1,10).reshape(3,3)
B=np.arange(5,14).reshape(3,3)

#determinante:
np.linalg.det(A)
#autovalores e autovetores:
np.linalg.eig(A)
#norma:
np.linalg.norm(A)
#potência:
np.linalg.matrixpower(A,5)
#decomposição QR
#(A=QR, sendo Q ortogonal e R triangular):
np.linalg.qr(A)
#inversa e pseudo-inversa:
np.linalg.inv(A)
np.linalg.pinv(A)
#solução da equação de autovalores (Ax=B):
np.linalg.solve(A,b)
np.linalg.lstsq(A,b)
```

$$\begin{bmatrix} -3 & -4 & -5 \\ 4 & 5 & 6 \\ 0 & 0 & 0 \end{bmatrix}$$

← Solução exata

← Solução aproximada via método dos mínimos quadrados

>>Estatística

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

$$B = \begin{bmatrix} 5 & 6 & 7 \\ 8 & 9 & 10 \\ 11 & 12 & 13 \end{bmatrix}$$

```
import numpy as np
A=np.arange(1,10).reshape(3,3)
B=np.arange(5,14).reshape(3,3)

#médias e desvio padrão:
np.mean(A)
np.average(A,axis=1,weights=np.array(1,2,1.5))
np.std(A)
```

Média não ponderada de todos os valores

Média ponderada (pesos 1,2 e 1.5) sobre os valores distribuídos pelas colunas (axis=1).
Se fossem distribuídos pelas linhas, diga axis=0.

Desvio padrão dos dados.
Também aceita parâmetro *axis* caso necessário

If your data has NaNs (not-a-number), then use *np.nanmean*, *np.nanstd*, etc

>>Estatística

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

$$B = \begin{bmatrix} 5 & 6 & 7 \\ 8 & 9 & 10 \\ 11 & 12 & 13 \end{bmatrix}$$

```
import numpy as np
A=np.arange(1,10).reshape(3,3)
B=np.arange(5,14).reshape(3,3)

#médias e desvio padrão:
np.mean(A)
np.average(A,axis=1,weights=np.array(1,2,1.5))
np.std(A)

#percentil
np.percentile(A,25)
```

De todos os valores de A, escolhe aquele que está a 25% do caminho entre o menor e o maior.
Neste caso, 3.0

>>Estatística

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

$$B = \begin{bmatrix} 5 & 6 & 7 \\ 8 & 9 & 10 \\ 11 & 12 & 13 \end{bmatrix}$$

```
import numpy as np
A=np.arange(1,10).reshape(3,3)
B=np.arange(5,14).reshape(3,3)

#médias e desvio padrão:
np.mean(A)
np.average(A,axis=1,weights=np.array(1,2,1.5))
np.std(A)

#percentil
np.percentile(A,25)

#correlação de Pearson
corrcoef(A,B)
```

Retorna a matriz de correlação entre as variáveis.
Neste caso todos os elementos são 1, pois ambas as variáveis crescem na mesma proporção

>>Estatística

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

$$B = \begin{bmatrix} 5 & 6 & 7 \\ 8 & 9 & 10 \\ 11 & 12 & 13 \end{bmatrix}$$

```
import numpy as np
A=np.arange(1,10).reshape(3,3)
B=np.arange(5,14).reshape(3,3)

#médias e desvio padrão:
np.mean(A)
np.average(A,axis=1,weights=np.array(1,2,1.5))
np.std(A)

#percentil
np.percentile(A,25)

#correlação de Pearson
corrcoef(A,B)

#histograma
histogram(A)
```

Cria o histograma através de duas arrays: A primeira é o número de elementos no intervalo (bin) e a segunda são os intervalos de valores correspondentes à cada bin.

>>Processamento de sinais

```
import numpy as np
A=np.arange(1,300)
```

#Transformada de Fourier
(converte da escala do sinal ao longo do tempo para um espaço equivalente aonde se extrai a frequência):

`Af=fft(A)` ← Geral

`Af=rfft(A)` ← Método mais simples se A for de números reais

>>Processamento de sinais

```
import numpy as np
A=np.arange(1,300)
```

#Transformada de Fourier
(converte da escala do sinal ao longo do tempo para um espaço equivalente aonde se extrai a frequência):

```
Af=fft(A)
Af=rfft(A)
```

#Transformada inversa de Fourier
(converte o sinal no espaço equivalente de volta para a escala de sinal ao longo do tempo):

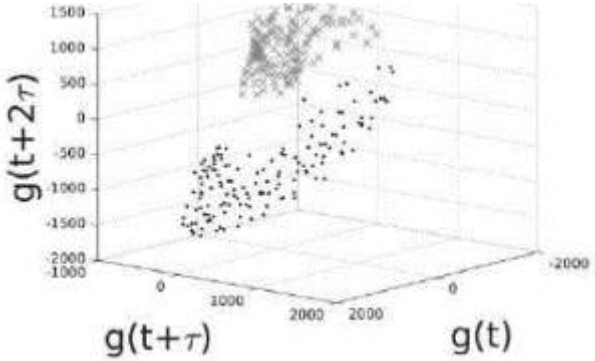
```
A2=ifft(Af)
```
← Geral

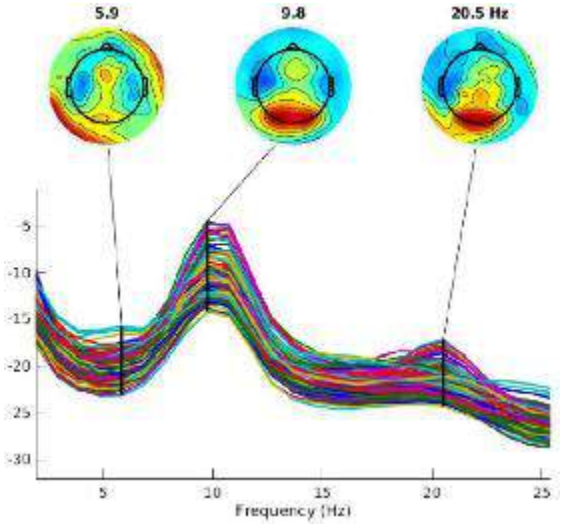
```
A2=irfft(Af)
```
← Método mais simples se A for de números reais

>>Processamento de sinais

```
import numpy as np
A=np.arange(1,300)

#Transformada de Fourier
(converte da escala do sinal ao longo do tempo para um espaço equivalente aonde se extrai a frequência):
Af=fft(A)
Af=rfft(A)

#Transformada inversa de Fourier
(converte o sinal no espaço equivalente de volta para a escala de sinal ao longo do tempo):
A2=ifft(Af)
A2=irfft(Af)

#Janelamento:
Hann=np.hanning(A.size/100)
Kais=np.kaiser(A.size/100)
```

Dois tipos de janela diferentes (Hanning e Kaiser), com comprimento igual à 1/100 do comprimento dos dados. Basta multiplicar esse valores pela região de comprimento 1/100 do tamanho total dos dados sendo analisada no intante t





UNICAMP

Instituto de Computação

That´s all folks!

arthur_valencio@physics.org

http://www.arthurvalencio.com/mc102

http://www.ic.unicamp.br/~mc102