Aula 3 – Introdução ao NumPy

EMC410235 - Programação Científica para Engenharia e Ciência Térmicas

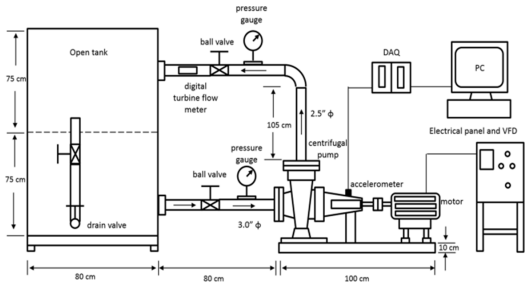
Prof. Rafael F. L. de Cerqueira

2025.2

Objetivos da Aula

- Apresentar o NumPy como ferramenta para computação científica
- Criar e manipular arrays NumPy
- Realizar operações numéricas básicas (vetoriais e matriciais)
- Carregar e salvar dados com NumPy

Exemplo de Aplicação - Aplicação de Leis de Semelhança para Bombas



- Considere o seguinte cenário: uma série de 20 testes é realizada em laboratório para determinar a curva de desempenho de uma bomba, sob uma mesma condição experimental.
- A medição da vazão é altamente precisa, permitindo desprezar sua incerteza.
- No entanto, sabe-se que o transdutor diferencial de pressão apresenta uma incerteza de 5% do fundo de escala (FS).

Por que usar NumPy?

```
conda install numpy
...
import numpy as np
```

- Eficiência computacional para operações com vetores e matrizes
- Sintaxe concisa, próxima ao MATLAB
- Operações vetorizadas eliminam o uso de laços explícitos
- Biblioteca padrão para computação científica em Python

https://numpy.org/

Python puro vs. NumPy

Python Puro

```
x = [1, 2, 3]
y = []
for i in x:
  y.append(i * 2)
  print(y)
```

Python puro vs. NumPy

Python Puro

```
x = [1, 2, 3]
y = []
for i in x:
  y.append(i * 2)
  print(y)
```

NumPy

```
import numpy as np

x = np.array([1, 2, 3])
y = x * 2
print(y)
```

Python puro vs. NumPy - Soma de vetores

Python Puro

```
a = [1, 2, 3]
b = [4, 5, 6]
soma = []
for i in range(len(a)):
  soma.append(a[i] + b[i])
  print(soma)
```

Python puro vs. NumPy - Soma de vetores

Python Puro

```
a = [1, 2, 3]
b = [4, 5, 6]
soma = []
for i in range(len(a)):
  soma.append(a[i] + b[i])
  print(soma)
```

NumPy

```
import numpy as np

a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
soma = a + b
print(soma)
```

Python puro vs. NumPy - Eficiência computacional

Python Puro

```
import time

N = 10**6
a = list(range(N))
b = list(range(N))

start = time.time()
c = [a[i]*b[i] for i in range(N)]
print(\"Tempo:\", time.time() - start)
```

Python puro vs. NumPy - Eficiência computacional

Python Puro

```
import time

N = 10**6
a = list(range(N))
b = list(range(N))

start = time.time()
c = [a[i]*b[i] for i in range(N)]
print(\"Tempo:\", time.time() - start)
```

NumPy

```
import numpy as np
import time

N = 10**6
a = np.arange(N)
b = np.arange(N)

start = time.time()
c = a * b
print(\"Tempo:\", time.
    time() - start)
```

Criação de arrays NumPy

```python
import numpy as np

np.array([1, 2, 3])         # vetor 1D de inteiros
np.zeros((2, 3))            # matriz 2x3 de floats
np.ones(4)                  # vetor 1D com 4 floats
np.full((2, 2), 7)          # matriz 2x2 com inteiros
np.zeros_like(A)           # mesma forma e tipo de A
np.full_like(B, np.nan)    # mesma forma de B, tipo float
```

- Use o parâmetro `dtype=` para forçar tipos como `int`, `float64`, `bool`, etc.

```python
import numpy as np

np.array([1, 2, 3], dtype=float)       # vetor 1D de inteiros
np.ones((4,), dtype=int)    # vetor unitario, do tipo int
```

Operações com arrays

```
import numpy as np

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

c1 = a * 2        # multiplicacao por escalar
c2 = a + b        # soma elemento a elemento
c3 = a * b        # produto elemento a elemento
c4 = np.dot(a, b) # produto escalar
c5 = a @ b        # produto escalar (equivalente ao dot)
```

- Operações vetorizadas evitam laços explícitos
- O operador `@` é mais legível para produto vetorial/matricial

Funções estatísticas (com e sem NaN)

```
import numpy as np

a = np.array([1, 2, np.nan])

media1 = np.mean(a)     # retorna nan
media2 = np.nanmean(a)  # ignora o nan
soma = np.nansum(a)
desvio = np.nanstd(a)
```

- Use as funções com prefixo nan para ignorar valores ausentes
- Ideal para dados experimentais ou incompletos

Criação programada de arrays

```
np.arange(0, 10, 2)      # passo fixo
np.linspace(0, 1, 5)     # 5 pontos igualmente espacados

x = np.linspace(-1, 1, 5)
y = np.linspace(-1, 1, 5)
X, Y = np.meshgrid(x, y) # grade 2D
```

Operações lógicas e booleanas

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

mask = a > 0          # array([ True, False, True ])
idx = np.where(a > 0) # (array([0, 2]),)

tem_positivos = np.any(a > 0)
todos_positivos = np.all(a > 0)
```

Funções matemáticas

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

raiz = np.sqrt(a)
exponencial = np.exp(a)
logaritmo = np.log(a)

senos = np.sin(a)
cossenos = np.cos(a)
```

Agregações por eixo em arrays 2D

```
A = np.array([[1, 2, 3],
              [4, 5, 6]])

soma_colunas = np.sum(A, axis=0) # array([5, 7, 9])
soma_linhas = np.sum(A, axis=1) # array([ 6, 15])
```

Transposição e reshaping

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

transposta = a.T
achatado = a.ravel()
reshape = a.reshape((4, 1))
```

Broadcasting e cópias

```
a = np.array([[1], [2], [3]]) # shape (3,1)
b = np.array([10, 20, 30])    # shape (3,)

soma = a + b                  # broadcasting automatico

# copia vs. referencia
c = a.copy()  # nova memoria
d = a         # apenas referencia
```

Indexação e slicing

```
a = np.array([10, 20, 30, 40, 50])

a[0]        # 10 (primeiro elemento)
a[-1]       # 50 (ultimo elemento)
a[1:4]      # array([20, 30, 40])
a[:3]       # array([10, 20, 30])
a[::2]      # array([10, 30, 50]) - de 2 em 2
```

Indexação e slicing

```
a = np.array([10, 20, 30, 40, 50])

a[0]        # 10 (primeiro elemento)
a[-1]       # 50 (ultimo elemento)
a[1:4]      # array([20, 30, 40])
a[:3]       # array([10, 20, 30])
a[::2]      # array([10, 30, 50]) - de 2 em 2
```

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

A[0, 1]     # 2 (linha 0, coluna 1)
A[1:, :2]   # submatriz 2x2: [[4,5],[7,8]]
A[:, 1]     # segunda coluna: array([2, 5, 8])
```

Salvando arquivos com np.savetxt

```
import numpy as np

# Criando dados simulados
x = np.linspace(0, 1, 5)
y = x**2
dados = np.column_stack((x, y))

# Salvando em arquivo CSV com cabecalho
np.savetxt('saida.csv',
           dados,
           delimiter=',', # define separador CSV
           header='x, y = x^2', # escreve um cabecalho no topo do arquivo
           comments='', # evita prefixar o cabecalho com '\#'
           fmt='%.4f') # formata os numeros com 4 casas decimais
```

Carregando arquivos com np.loadtxt

```
# Carrega um CSV ignorando a primeira linha (cabecalho)
dados = np.loadtxt('saida.csv',
delimiter=',',
skiprows=1) # ignora o cabecalho

# Seleciona apenas a primeira coluna
x = np.loadtxt('saida.csv',
delimiter=',',
skiprows=1,
usecols=0) # permite selecionar colunas especificas

# Converte para tipo float64 explicitamente
dados = np.loadtxt('saida.csv',
delimiter=',',
skiprows=1,
dtype=np.float64) # garante tipo numerico desejado
```

Salvando arquivos com np.savez_compressed

```
import numpy as np

a = np.array([1, 2, 3])
b = np.array([10, 20, 30])

# Salvando multiplas variaveis em um unico arquivo .npz
np.savez_compressed('dados_compactados.npz',
                    vetor_a=a,
                    vetor_b=b)
```

- Cria um arquivo binário compactado (.npz)
- Permite salvar vários arrays nomeados juntos
- Ideal para projetos e grandes volumes de dados

Carregando arquivos com np.load

```
import numpy as np

# Abrindo o arquivo salvo
conteudo = np.load('dados_compactados.npz')

# Acessando arrays pelo nome
a = conteudo['vetor_a']
b = conteudo['vetor_b']

print(a)
print(b)
```

- O retorno de `np.load` é similar a um dicionário
- A leitura é eficiente e preserva os tipos e formatos

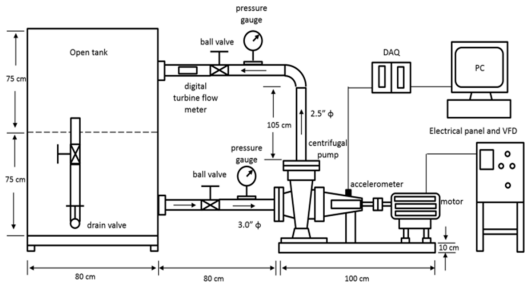
Exemplo de Aplicação

Gerador sintético de dados para bomba

e

Aplicação de Leis de Semelhança para Bombas

Exemplo de Aplicação - Aplicação de Leis de Semelhança para Bombas



- Considere o seguinte cenário: uma série de 20 testes é realizada em laboratório para determinar a curva de desempenho de uma bomba, sob uma mesma condição experimental.
- A medição da vazão é altamente precisa, permitindo desprezar sua incerteza.
- No entanto, sabe-se que o transdutor diferencial de pressão apresenta uma incerteza de 5% do fundo de escala (FS).

Gerador sintético de dados para bomba

Infelizmente, para essa aula, não tenho dados de um experimento real. Isso não é problema! **Podemos gerar nossos próprios dados sintéticos!**

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

HOUR_TO_SECOND = 3600

def H_equation(Q, H_shut_off, A):
  return H_shut_off - A * np.power(Q,
       2.0)

H_shut_off = 50.0 # m
A = 80000
H_unc = 0.05 * H_shut_off

Q_max = 80 # m3/h
flowrate_values = np.linspace(0.0, Q_max,
      10)
flowrate_values = np.linspace(0.0, Q_max,
      10) / HOUR_TO_SECOND
```

```
N_experiments = 20
for synthetic_experiment in range(
     N_experiments):
  H_unc_values = np.random.normal(
       flowrate_values, H_unc)
  H_values = H_equation(flowrate_values,
       H_shut_off, A)
  H_values += H_unc_values

plt.scatter(flowrate_values, H_values,
     edgecolor='k')

plt.xlabel(r'Q [$m^3/s$]')
plt.ylabel('H [m]')
plt.grid()
plt.show()
```

Gerador sintético de dados para bomba

Realizados um total de 20 “experimentos” temos a seguinte relação entre carga líquida H e vazão Q.