Python Científico para Engenharia: Ferramentas Numéricas e Simbólicas com SciPy e SymPy

EMC410235 - Programação Científica para Engenharia e Ciência Térmicas

Prof. Rafael F. L. de Cerqueira

2025.2

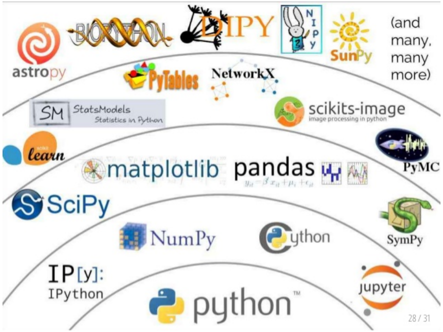
Motivação

- Problemas de engenharia exigem:
  - Integração e diferenciação
  - Resolução de equações (algebraicas e diferenciais)
  - Otimização de funções
  - Análise e suavização de dados experimentais

- **Ferramentas em Python:**
  - `SciPy` – soluções numéricas
    `conda install scipy`
  - `SymPy` – manipulação simbólica
    `conda install sympy`

Ecossistema Científico em Python

- Python é uma linguagem com forte suporte à computação científica.
- Ecossistema formado por bibliotecas amplamente utilizadas:
  - `NumPy` – vetores, matrizes, álgebra linear ✓
  - `Matplotlib` – visualização de dados ✓
  - `Pandas` – análise de dados tabulares ✓
  - `SciPy` – ferramentas numéricas
  - `SymPy` – álgebra simbólica
- Hoje: foco em `SciPy` e `SymPy` como *ferramentas para engenharia*.



Fonte: sagol-python-for-neuroscientists.github.io

O que vamos ver hoje

- **Parte 1 – SciPy: Soluções Numéricas**
  - Solução de Equações Algébricas Não-Lineares: `scipy.integrate.solve` ✓
  - Integração numérica: `scipy.integrate`
  - Solução de EDOs: `scipy.integrate.solve_ivp`
  - Interpolação de dados: `scipy.interpolate.interp1d`
  - Suavização de dados: `scipy.signal`
  - Otimização: `scipy.optimize`
  - Análise de sinais: `scipy.fft`
- **Parte 2 – SymPy: Manipulação Simbólica**
  - Derivadas e integrais analíticas
  - Resolução de equações algébricas
  - Solução simbólica de EDOs: `dsolve`
  - Dedução simbólica em problemas de engenharia

Integração Numérica com quad

Método: quadratura adaptativa (`QUADPACK`)

- Ideal para integrar funções contínuas simples.
- Exemplo: calcular

$$\int_0^5 x^2 e^{-x} \, dx$$

```python
from scipy.integrate import quad
import numpy as np

f = lambda x: x**2 * np.exp(-x)
resultado, erro = quad(f, 0, 5)

print("Integral =", resultado)
```

Integração Numérica com Dados Discretos

Métodos:

- `trapezoid` – regra dos Trapézios
- `simpson` – regra de Simpson

Exemplo: integrar o sinal `y = sin(x)` definido em pontos discretos.

```python
import numpy as np
from scipy.integrate import trapezoid, simpson

x = np.linspace(0, np.pi, 100)
y = np.sin(x)

area_trapz = trapezoid(y, x)
area_simps = simpson(y, x)

print("Integra(simps) =", area_simps)
print("Integral(trapz) =", area_trapz)
```

Aplicação: útil quando os dados vêm de experimentos ou simulações.

Comparando Métodos de Integração Numérica

```python
import numpy as np
from scipy.integrate import quad,
from scipy.integrate import trapezoid, simpson

x = np.linspace(0, np.pi, 100)
y = np.sin(x)

f = lambda x: np.sin(x)
quad_result, _ = quad(f, 0, np.pi)

trapz_result = trapezoid(y, x)
simpson_result = simpson(y, x)

print("quad      =", quad_result)
print("trapezoid =", trapz_result)
print("simpson   =", simpson_result)
```

Função: $\sin(x)$ no intervalo $[0, \pi]$

Resultado exato: 2.0

Métodos:

- `quad` – função contínua
- `trapezoid` – dados discretos (trapézios)
- `simpson` – dados discretos (Simpson)

Integrais Múltiplas com scipy.integrate

```python
from scipy.integrate import dblquad, tplquad

# Integral dupla de x*y em [0,1] x [0,2]
f2 = lambda y, x: x * y
area, _ = dblquad(f2, 0, 1,
                  lambda x: 0,
                  lambda x: 2)

# Integral tripla de x*y*z em x[0,1], y[2,5], z[7,8]
f3 = lambda z, y, x: x * y * z
volume, _ = tplquad(f3, 0, 1,
                    lambda x: 2,
                    lambda x: 5,
                    lambda x, y: 7,
                    lambda x, y: 8)
```

Integral Dupla:

$$\int_0^1 \int_0^2 xy \, dy \, dx$$

Integral Tripla:

$$\int_0^1 \int_2^5 \int_7^8 xyz \, dz \, dy \, dx$$

Ordem dos argumentos:

`tplquad(f(z, y, x), x_a, x_b, y_a, y_b, z_a, z_b)`

Integral Dupla com Limites Variáveis

```python
from scipy.integrate import dblquad

# Região triangular: y vai de 0 a x
f = lambda y, x: x + y
area, _ = dblquad(f, 0, 1,
lambda x: 0,
lambda x: x)
```

Região: triângulo com $0 \leq x \leq 1$, $0 \leq y \leq x$

Função: $f(x, y) = x + y$

Resultado analítico:

$$\int_0^1 \int_0^x (x + y)\, dy\, dx = \frac{5}{12}$$

Resfriamento de Newton: Modelo e Implementação

Modelo Matemático:

$$\frac{dT}{dt} = -k(T - T_\infty)$$

- k: coeficiente de resfriamento
- T_∞: temperatura ambiente
- $T(0) = T_0$: condição inicial

Objetivo: resolver numericamente com `solve_ivp`.

```
from scipy.integrate import solve_ivp
import numpy as np

k = 0.5
T_inf = 20

def dTdt(t, T):
        return -k * (T - T_inf)

T_0 = 80 # Temperatura Inicial

sol = solve_ivp(dTdt, [0, 10], [T_0],
t_eval=np.linspace(0, 10, 100))
```

Visualização da Solução com matplotlib

```python
import matplotlib.pyplot as plt

plt.plot(sol.t, sol.y[0])
plt.xlabel('Tempo [s]')
plt.ylabel('Temperatura [°C]')
plt.title('Resfriamento de Newton')
plt.grid(True)
plt.show()
```

Comportamento esperado:

Decaimento exponencial de $T(t)$ até atingir $T_\infty = 20^\circ C$

Métodos disponíveis no `solve_ivp`

A função `solve_ivp` permite escolher diferentes métodos numéricos:

- `RK45` – Runge-Kutta de ordem 4(5) (padrão)
- `RK23` – Runge-Kutta de ordem 2(3), mais conservador
- `Radau` – Runge-Kutta implícito (bom para sistemas rígidos)
- `BDF` – método de diferenças retroativas (Gear)
- `LSODA` – algoritmo adaptativo que alterna entre métodos (ODEPACK)

Exemplo de uso: `solve_ivp(f, [t0, tf], y0, method='BDF')`

Sistemas de EDOs com solve_ivp

Exemplo: Oscilador harmônico (massa-mola)

$$\begin{cases} \frac{dx}{dt} = v \\ \frac{dv}{dt} = -kx \end{cases}$$

Código:

```
def sistema(t, y):
    x, v = y
    return [v, -k * x]

x_0 = 1; v_0 = 0
sol = solve_ivp(sistema, [0, 10], [x_0, v_0],
t_eval=np.linspace(0, 10, 100))
```

Resultado:

sol.y[0] $\rightarrow$ posição $x(t)$

sol.y[1] $\rightarrow$ velocidade $v(t)$

Resfriamento de Newton com $k(T) = a + bT$

Modelo não linear:

$$\frac{dT}{dt} = -k(T)(T - T_\infty)$$

com:

$$k(T) = a + bT$$

Parâmetros:

- $a = 0{,}1$
- $b = 0{,}005$
- $T_\infty = 20^\circ C$
- $T(0) = 80^\circ C$

```
from scipy.integrate import solve_ivp
import numpy as np

a, b = 0.1, 0.005
T_inf = 20

def k(T): return a + b * T
def dTdt(t, T): return -k(T) * (T - T_inf)

T_0 = 80
sol = solve_ivp(dTdt, [0, 10], [T_0],
t_eval=np.linspace(0, 10, 100))
```

Interpolação Linear com interp1d

Situação: temos dados experimentais esparsos:

- $x = [0, 1, 2, 3, 4]$
- $y = [0, 2, 1, 3, 7]$

Objetivo: reconstruir a curva com interpolação linear.

```python
import numpy as np
from scipy.interpolate import interp1d
import matplotlib.pyplot as plt

x = np.array([0, 1, 2, 3, 4])
y = np.array([0, 2, 1, 3, 7])

f_interp = interp1d(x, y)

x_new = np.linspace(0, 4, 100)
y_new = f_interp(x_new)

plt.plot(x, y, 'o', label='dados')
plt.plot(x_new, y_new, '-', label='interp linear')
plt.legend()
plt.grid(True)
plt.show()
```

Interpolação: linear vs. quadrática vs. cúbica

Situação: reconstrução de curva a partir de poucos dados.

Função original:

- $x = [0, 1, 2, 3, 4]$
- $y = [0, 2, 1, 3, 7]$

Comparação de métodos:

- `linear` – conexões diretas entre pontos
- `quadratic` – suavização com parábolas
- `cubic` – suavização suave com polinômios de grau 3

```python
from scipy.interpolate import interp1d
import numpy as np
import matplotlib.pyplot as plt

x = np.array([0, 1, 2, 3, 4])
y = np.array([0, 2, 1, 3, 7])

f_lin = interp1d(x, y, kind='linear')
f_quad = interp1d(x, y, kind='quadratic')
f_cub = interp1d(x, y, kind='cubic')

x_new = np.linspace(0, 4, 200)

plt.plot(x, y, 'o', label='dados')
plt.plot(x_new, f_lin(x_new), '--', label='linear')
plt.plot(x_new, f_quad(x_new), '-.', label='quadrática')
plt.plot(x_new, f_cub(x_new), '-', label='cúbica')
plt.legend(); plt.grid(); plt.show()
```

Interpolação Bilinear com RegularGridInterpolator

Grade de dados:

$$x = [0, 1, 2], \quad y = [0, 1, 2]$$

$$f(x, y) = x^2 + y^2$$

Objetivo: interpolar para $x = 0{,}5$, $y = 1{,}3$

Aplicação típica:

- Tabelas 2D de propriedades
- Campos simulados

Extensível para múltiplas dimensões Exemplo: propriedades termodinâmicas: h(T, P, y_1, ...)

```python
from scipy.interpolate import RegularGridInterpolator
import numpy as np

x = [0, 1, 2]
y = [0, 1, 2]
X, Y = np.meshgrid(x, y, indexing='ij')
values = X**2 + Y**2  # f(x, y)

interp = RegularGridInterpolator((x, y), values)

pt = [0.5, 1.3]
print(interp(pt))  # interpolação bilinear
```

Suavização de Dados com `savgol_filter`

Problema: Dados experimentais ruidosos
Solução: Filtro de Savitzky-Golay

- Suavização com preservação de forma
- Ajusta polinômios locais via regressão

Parâmetros:

- `window_length` (ímpar): largura da janela
- `polyorder`: grau do polinômio ajustado

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import savgol_filter

x = np.linspace(0, 2*np.pi, 100)
y = np.sin(x) + 0.2*np.random.randn(100)

y_smooth = savgol_filter(y, window_length=11,
                         polyorder=3)

plt.plot(x, y, label='original (ruído)')
plt.plot(x, y_smooth, label='suavizado')
plt.legend(); plt.grid(); plt.show()
```

Suavização e Derivada com savgol_filter

Problema: Derivar um sinal com ruído amplifica o erro
Solução: Suavizar antes de derivar com savgol_filter
Aplicação típica:

- derivada de posição → velocidade
- derivada de temperatura → fluxo térmico
- dados simulados ou experimentais ruidosos

```python
from scipy.signal import savgol_filter
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 10, 200)
y = np.sin(x) + 0.1*np.random.randn(200)

dy_direct = np.gradient(y, x)  # derivada numérica direta


y_smooth = savgol_filter(y, 31, 3)
dy_dx = savgol_filter(y, 31, 3, deriv=1, delta=x[1]-x[0])

plt.plot(x, y, color='gray', alpha=0.5, label='ruído')
plt.plot(x, y_smooth, label='suavizado')
plt.plot(x, dy_dx, '--', label="1ª derivada (filtrada)")
plt.plot(x, dy_direct, '--', label="1ª derivada")

plt.legend(); plt.grid(); plt.show()
```

Comparando Técnicas de Suavização: medfilt e filtfilt

Sinal de entrada:

$$y(x) = \sin(x) + \text{ruído}$$

Filtros aplicados:

- medfilt (janela = 5)
- filtfilt + butter (ordem 3)

Observações:

- medfilt é robusto contra outliers
- filtfilt preserva fase e suaviza bem

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import medfilt, butter, filtfilt

fs = 1000
fc = 20; ordem = 3

t = np.linspace(0, 0.2, fs, endpoint=False)
y = np.sin(2*np.pi*10*t) + 0.1*np.random.randn(len(t))

wn = fc / (fs / 2)
b, a = butter(ordem, wn, btype='low')

# Filtro Passa-Baixa Butterworth
y_butter = filtfilt(b, a, y)
# Filtro da mediana
y_med = medfilt(y, kernel_size=5)

plt.plot(t, y, alpha=0.4, label='ruído')
plt.plot(t, y_med, '--', label='medfilt')
plt.plot(t, y_butter, '-', label='filtfilt')
plt.legend(); plt.grid(); plt.show()
```

Otimização com scipy.optimize

Problema: encontrar o ponto x^* que minimiza $f(x)$

Função principal: `scipy.optimize.minimize`

Exemplo básico:

```python
from scipy.optimize import minimize

res = minimize(fun, x0)
```

Extras:

- `constraints=[...]`
- `bounds=[(a,b), ...]`

Métodos disponíveis:

- `'BFGS'` – gradiente (sem restrições)
- `'Nelder-Mead'` – não usa derivadas
- `'SLSQP'` – com restrições de igualdade/inequação
- `'trust-constr'` – problemas grandes e com restrições

Sugestão:

- Use `Nelder-Mead` para protótipos simples
- Use `SLSQP` para problemas com restrições
- Use `method=...` para escolher explicitamente

Visualização da Função e do Mínimo

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize

f = lambda x: x*np.sin(x) + x**2 - 10

res = minimize(f, x0=2.0, method='Nelder-Mead')

x_vals = np.linspace(-5, 5, 400)
y_vals = f(x_vals)

plt.plot(x_vals, y_vals, label='f(x)')
plt.plot(res.x, f(res.x), 'ro', label='mínimo')
plt.xlabel('x'); plt.ylabel('f(x)')
plt.title('Minimização de f(x) = x·sin(x) + x²')
plt.grid(); plt.legend(); plt.show()
```

Observação: o método encontra um *mínimo local* próximo ao chute inicial.

Visualização: Contorno da Função Quadrática

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize

# Definição da função
def f(x):
        return x[0]**2 + x[1]**2 + x[0]*x[1] + x[0] + x[1]

# Otimização
res = minimize(f, [1, 1])

# Grid para contorno
x = np.linspace(-3, 1, 100)
y = np.linspace(-3, 1, 100)
X, Y = np.meshgrid(x, y)
Z = f([X, Y])

# Plot
plt.contour(X, Y, Z, levels=30, cmap='viridis')
plt.plot(res.x[0], res.x[1], 'ro', label='mínimo')
plt.xlabel('x'); plt.ylabel('y')
plt.title('Contorno de $f(x, y) = x^2 + y^2 + xy + x + y$')
plt.grid(); plt.legend(); plt.show()
```

Minimização com Limites (Bounds)

Função:

$$f(x, y) = x^2 + y^2 + xy + x + y$$

Restrição:

$$\begin{cases} 1.0 \leq x \leq 1.5 \\ 0.5 \leq y \leq 2.0 \end{cases}$$

Observação: mínimo global pode estar fora da região viável.

```
from scipy.optimize import minimize

def f(x):
        return x[0]**2 + x[1]**2 + x[0]*x[1] +

bounds = [(1.0, 1.5), (0.5, 2.0)]

res = minimize(f, x0=[1.2, 1.5],
bounds=bounds)

print("Solução com bounds:", res.x)
```

Transformada Rápida de Fourier (FFT)

Objetivo: decompor um sinal no domínio do tempo em componentes de frequência

$$x(t) \xrightarrow{\texttt{fft}} X(f)$$

Aplicações típicas:

- Análise de vibrações e ruídos
- Sinais de sensores (pressão, temperatura, som)
- Extração de frequências dominantes
- Processamento de sinais simulados ou experimentais

Função principal: `scipy.fft.fft` (1D) e `scipy.fft.fftfreq` (frequências)

FFT de um Sinal Senoidal

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.fft import fft, fftfreq

Fs = 1000
t = np.linspace(0, 1, Fs, endpoint=False)
f0 = 50
y = np.sin(2 * np.pi * f0 * t)

N = len(t)
Y = fft(y)
freq = fftfreq(N, 1/Fs)

amp = 2 * np.abs(Y[:N//2]) / N
freq_pos = freq[:N//2]

plt.plot(freq_pos, amp)
plt.xlabel('Frequência [Hz]')
plt.ylabel('Amplitude')
plt.title('FFT de uma senoide 50 Hz')
plt.grid(); plt.show()
```

- `Fs`: frequência de amostragem (Hz)
- `f0`: frequência da senoide
- `Y = fft(y)`: FFT do sinal
- `freq = fftfreq(...)`: vetor de frequências
- Apenas a metade positiva ($f \in [0, Fs/2]$) é usada
- `amp = 2 * np.abs(...) / N`: normaliza a FFT
- A multiplicação por 2 compensa a simetria do espectro
- O resultado mostra a contribuição de cada frequência

FFT de Sinal com Quatro Senoides

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.fft import fft, fftfreq

# Parâmetros
Fs = 1000  # Hz
t = np.linspace(0, 1, Fs, endpoint=False)

# Frequências
f1, f2 = 20, 50
f3, f4 = 80, 150

# Sinal composto
y = (np.sin(2*np.pi*f1*t) +
0.8*np.sin(2*np.pi*f2*t) +
0.6*np.sin(2*np.pi*f3*t) +
0.4*np.sin(2*np.pi*f4*t))
```

```python
# FFT
Y = fft(y)
freq = fftfreq(len(t), 1/Fs)

# Espectro
plt.plot(freq[:Fs//2], np.abs(Y[:Fs//2]))
plt.xlabel('Frequência [Hz]')
plt.ylabel('Amplitude')
plt.title('FFT - Sinal Multicomponente')
plt.grid(); plt.show()
```

FFT de Sinal Ruidoso

```
# Senoide com ruído branco
y_noise = y + 0.5*np.random.randn(len(t))
Y_noise = fft(y_noise)

plt.plot(freq[:Fs//2], np.abs(Y_noise[:Fs//2]))
plt.xlabel('Frequência [Hz]')
plt.ylabel('Amplitude')
plt.title('FFT de senoide com ruído')
plt.grid(); plt.show()
```

Observação: o pico na frequência dominante ainda é visível (50 Hz)

Interpretação do Espectro de Frequências

FFT retorna:

- Um vetor de amplitudes complexas
- Frequências associadas com cada componente

Pós-processamento comum:

- Usar apenas a metade positiva do espectro ($f > 0$)
- Tomar módulo: $|Y(f)|$
- Opcional: normalizar, converter em dB, filtrar

Importante:

- Quanto maior o número de amostras $\rightarrow$ melhor resolução espectral
- Quanto maior a taxa de amostragem $\rightarrow$ maior frequência detectável

Motivação Física com SymPy: Raio Crítico de Isolamento

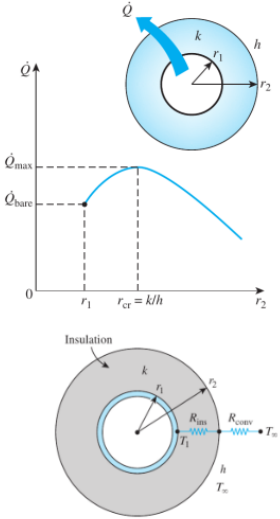
Problema: determinar o raio externo r_2 ótimo de isolamento para um tubo, de modo a maximizar a perda de calor

Modelo térmico:

$$R_{\text{cond}} = \frac{\ln(r_2/r_1)}{2\pi kL}, \quad R_{\text{conv}} = \frac{1}{2\pi r_2 hL}$$

$$R_{\text{eq}} = R_{\text{cond}} + R_{\text{conv}}, \quad Q = -\frac{T_i - T_\infty}{R_{\text{eq}}}$$

Solução: usar `SymPy` para derivar $Q(r_2)$ e resolver $\frac{dQ}{dr_2} = 0$



SymPy – Criando Expressões Simbólicas

Passo 1: Definir os símbolos simbólicos que serão usados

```
from sympy import symbols, log, pi

# Definindo variáveis simbólicas
r1, r2, L, k = symbols('r_1 r_2 L k')

# Resistência térmica por condução (cilindro)
R_cond = log(r2 / r1) / (2 * pi * k * L)

print(R_cond)
```

Resultado: Uma expressão simbólica manipulável, como:

$$R_{\text{cond}} = \frac{\ln(r_2/r_1)}{2\pi kL}$$

SymPy – Modelo de Transferência de Calor

Passo 2: Adicionar convecção e construir a equação do fluxo de calor

```python
from sympy import symbols, log, pi, simplify

# Novos símbolos
h, T_i, T_inf = symbols('h T_i T_inf')

# Resistência por convecção
R_conv = 1 / (2 * pi * r2 * h * L)

# Soma das resistências
R_eq = R_cond + R_conv

# Equação do fluxo de calor
Q = -(T_i - T_inf) / R_eq

# Expressão simplificada
Q_simplified = simplify(Q)
```

Resultado: Expressão simbólica para $Q(r_2)$, pronta para ser analisada.

SymPy – Derivada e Raio Crítico

Passo 3: Calcular derivada simbólica e resolver equação

```python
from sympy import diff, Eq, solve

# Derivada de Q em relação a r2
dQ_dr2 = diff(Q_simplified, r2)

# Condição de máximo
equation = Eq(dQ_dr2, 0)

# Solução simbólica para r2
r2_critical = solve(equation, r2)
print(r2_critical)
```

Resultado: Valor ótimo simbólico de r_2, ou seja:

$$r_2^* = \frac{k}{h}$$