Aprendendo a usar os pacotes solve_ivp e solve_bvp

Prof. Doherty Andrade- www.metodosnumericos.com.br

1 Introdução

Neste notebook Jupyter apresentamos alguns exemplos de como utilizar o solve_bvp e o solve_ivp. A biblioteca solve_bvp e solve_ivp fazem parte do SciPy que é um sistema baseado no software open-source Python para Ciências Física, Matemática e Engenharias.

Alguns dos pacotes que compõem SciPy:
NumPy: pacote básico de matemática e matrizes;
Biblioteca SciPy: Biblioteca fundamental para computação científica;
Matplotlib: pacote para plotagem;
IPython: console interativo;
SymPy: pacote de Matemática Simbólica;
Pandas: pacote de Estruturas e análise de dados.

2 O pacote solve_bvp: exemplos

O pacote solve_bvp resolve numericamente problemas de valor de bordo (BVP) para sistemas de EDO's de primeira ordem. A sintaxe é:

scipy.integrate.solve_bvp(fun, bc, x, y, p=None, S=None, fun_jac=None, bc_jac=None, tol=0.001, max_nodes=1000, verbose=0, bc_tol=None). onde:

$$\frac{dy}{dx} = f(x, y, p) + \frac{Sy}{x - a}, x \in [a, b]$$

$$bc(y(a), y(b), p) = 0$$

sendo que p é um parâmetro.
fun: é função lado direito do sistema
bc: função de condições de fronteira
Os exemplos considerados aqui são conhecidos e tomados dos papers:
[1] "A BVP Solver Based on Residual Control and the Maltab PSE".
[2] "Solving Boundary Value Problems for Ordinary Differential Equations in Matlab with bvp4c"

```
In [1]: %load_ext autoreload
        %autoreload 2
```

Primeiramente vamos importar os pacotes, NumPy, solve_bvp e o pacote matplotlib.
Fazemos isto com os comandos a seguir:

```
In [2]: import numpy as np
        from scipy.integrate import solve_bvp
        import matplotlib.pyplot as plt
        %matplotlib inline
```

3 Exemplo 1

Vamos estudar o BVP dado a seguir:

$$y'' + \exp(y) = 0$$

$$y(0) = y(1) = 0.$$

Primeira providência é passar para um sistema de primeira ordem:

$$y_1' = y_2 \quad (1)$$
$$y_2' = -\exp(y_1). \quad (2)$$

Agora vamos entrar com os dados.

```
In [3]: #função do lado direito do sistema
        def fun(x, y):
            return np.vstack((y[1], -np.exp(y[0])))
```

Implementando as condições de fronteira

```
In [4]: #condições de fronteiro ou bordo
        def bc(ya, yb):
            return np.array([ya[0], yb[0]])
```

Definindo a malha inicial com 10 nós.

```
In [5]: #malha inicial com 10 nós
        x = np.linspace(0, 1, 10)
```

Este problema é conhecido ter duas soluções aqui denotadas por y_a e y_b.

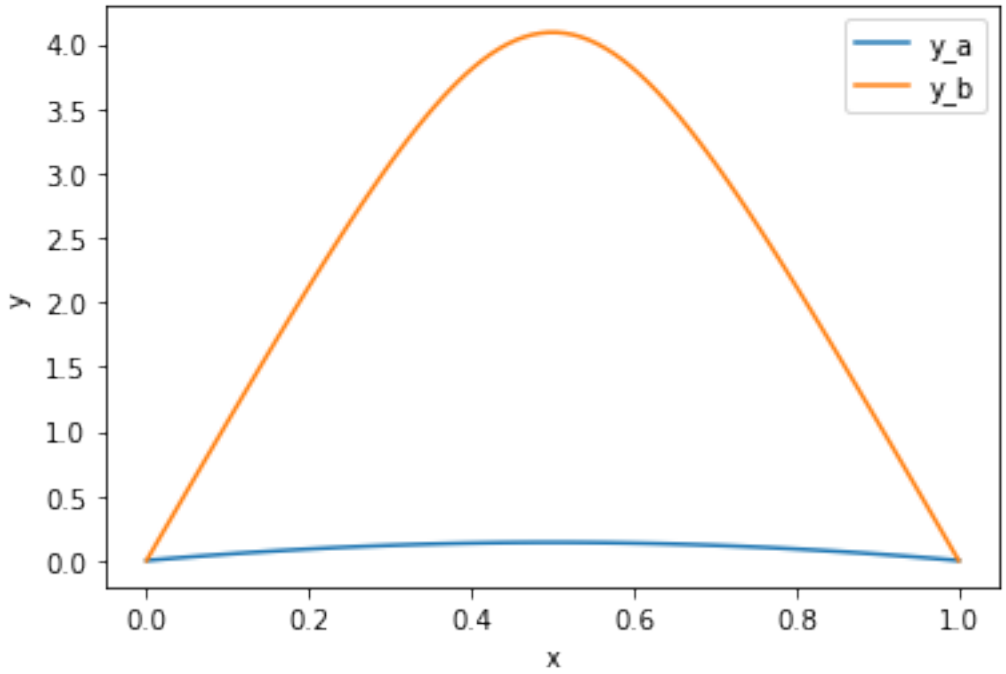
```
In [6]: y_a = np.zeros((2, x.size))
        y_b = np.zeros((2, x.size))
        y_b[0] = 3
```

Agora vamos obter as duas soluções.

```
In [7]: from scipy.integrate import solve_bvp
        res_a = solve_bvp(fun, bc, x, y_a)
        res_b = solve_bvp(fun, bc, x, y_b)
```

Agora vamos plotar as duas soluções. Vamos nos valer de ter a solução em forma de spline para produzir um gráfico suave.

```
In [8]: x_plot = np.linspace(0, 1, 100)
        y_plot_a = res_a.sol(x_plot)[0]
        y_plot_b = res_b.sol(x_plot)[0]
        import matplotlib.pyplot as plt
        plt.plot(x_plot, y_plot_a, label='y_a')
        plt.plot(x_plot, y_plot_b, label='y_b')
        plt.legend()
        plt.xlabel("x")
        plt.ylabel("y")
        plt.show()
```



Para ver os valores numéricos das soluções simplesmente digite res_a e res_b.

```
In [9]: res_a
```

```
Out[9]:        message: 'The algorithm converged to the desired accuracy.'
                 niter: 1
                     p: None
          rms_residuals: array([8.74717333e-06, 8.66666248e-06, 8.77574257e-06, 9.36989415e-06,
                1.00543952e-05, 9.36989415e-06, 8.77574257e-06, 8.66666248e-06,
                8.74717333e-06])
                   sol: <scipy.interpolate.interpolate.PPoly object at 0x000002A5F9DF8620>
                status: 0
                success: True
                     x: array([0.        , 0.11111111, 0.22222222, 0.33333333, 0.44444444,
                0.55555556, 0.66666667, 0.77777778, 0.88888889, 1.        ])
```

```
           y: array([[ 0.        ,  0.05474535,  0.09646193,  0.12459664,  0.13876329,
            0.13876329,  0.12459664,  0.09646193,  0.05474535,  0.        ],
          [ 0.54935197,  0.43502221,  0.31504312,  0.19079973,  0.06390059,
           -0.06390059, -0.19079973, -0.31504312, -0.43502221, -0.54935197]])
           yp: array([[ 0.54935197,  0.43502221,  0.31504312,  0.19079973,  0.06390059,
           -0.06390059, -0.19079973, -0.31504312, -0.43502221, -0.54935197],
          [-1.        , -1.0562716 , -1.10126766, -1.13269148, -1.14885212,
           -1.14885212, -1.13269148, -1.10126766, -1.0562716 , -1.        ]])
```

In [10]: res_b

Out[10]:
```
        message: 'The algorithm converged to the desired accuracy.'
          niter: 3
              p: None
  rms_residuals: array([0.00068604, 0.00079387, 0.00079789, 0.00065649, 0.00034929,
        0.00018776, 0.00062187, 0.00081388, 0.00012499, 0.00026939,
        0.00026939, 0.00012499, 0.00081388, 0.00062187, 0.00018776,
        0.00034929, 0.00065649, 0.00079789, 0.00079387, 0.00068604])
             sol: <scipy.interpolate.interpolate.PPoly object at 0x000002A5F9DF8728>
         status: 0
        success: True
              x: array([0.        , 0.05555556, 0.11111111, 0.16666667, 0.22222222,
        0.27777778, 0.33333333, 0.38888889, 0.44444444, 0.47222222,
        0.5       , 0.52777778, 0.55555556, 0.61111111, 0.66666667,
        0.72222222, 0.77777778, 0.83333333, 0.88888889, 0.94444444,
        1.        ])
              y: array([[ 0.00000000e+00,  6.00709966e-01,  1.19562577e+00,
         1.78004950e+00,  2.34568913e+00,  2.87837308e+00,
         3.35523441e+00,  3.74291343e+00,  4.00053454e+00,
         4.06848074e+00,  4.09147400e+00,  4.06848074e+00,
         4.00053454e+00,  3.74291343e+00,  3.35523441e+00,
         2.87837308e+00,  2.34568913e+00,  1.78004950e+00,
         1.19562577e+00,  6.00709966e-01,  0.00000000e+00],
        [ 1.08470581e+01,  1.07708764e+01,  1.06323682e+01,
         1.03825851e+01,  9.93867914e+00,  9.16993524e+00,
         7.89645218e+00,  5.93435828e+00,  3.22512671e+00,
         1.64921869e+00, -1.54161831e-14, -1.64921869e+00,
        -3.22512671e+00, -5.93435828e+00, -7.89645218e+00,
        -9.16993524e+00, -9.93867914e+00, -1.03825851e+01,
        -1.06323682e+01, -1.07708764e+01, -1.08470581e+01]])
          yp: array([[ 1.08470581e+01,  1.07708764e+01,  1.06323682e+01,
         1.03825851e+01,  9.93867914e+00,  9.16993524e+00,
         7.89645218e+00,  5.93435828e+00,  3.22512671e+00,
         1.64921869e+00, -1.54161831e-14, -1.64921869e+00,
        -3.22512671e+00, -5.93435828e+00, -7.89645218e+00,
        -9.16993524e+00, -9.93867914e+00, -1.03825851e+01,
        -1.06323682e+01, -1.07708764e+01, -1.08470581e+01],
        [-1.00000000e+00, -1.82341290e+00, -3.30562567e+00,
```

```
-5.93014996e+00, -1.04404650e+01, -1.77853143e+01,
-2.86523197e+01, -4.22208183e+01, -5.46273429e+01,
-5.84680671e+01, -5.98280135e+01, -5.84680671e+01,
-5.46273429e+01, -4.22208183e+01, -2.86523197e+01,
-1.77853143e+01, -1.04404650e+01, -5.93014996e+00,
-3.30562567e+00, -1.82341290e+00, -1.00000000e+00]])
```

4 Exemplo 2

Resolver o problema simples de Sturm-Liouville dado por:

$$y'' + k^2 y = 0$$
$$y(0) = y(1) = 0$$

Sabemos que a solução não trivial é $(x) = A * \sin(kx)$ desde que $k = n\pi$, com n inteiro.

Vamos tomar $A = 1$ e adicionar a condição de fronteira $y'(0) = k$.

Como exigido, precisamos escrever a equação diferencial como um sistema de primeira ordem.

$$y_1' = y_2$$
$$y_2' = -k^2 y_1$$

```
In [11]: def fun(x, y, p):
             k = p[0]
             return np.vstack((y[1], -k**2 * y[0]))
```

Vamos implementar as condições de fronteira.

```
In [12]: def bc(ya, yb, p):
             k = p[0]
             return np.array([ya[0], yb[0], ya[1] - k])
```

Vamos definir a malha iniciale o candidato para y.

Vamos determinar a solução para $k = 2\pi$ para para obter os valores de y para aproximar $\sin(2\pi x)$.

```
In [13]: x = np.linspace(0, 1, 5)
         y = np.zeros((2, x.size))
         y[0, 1] = 1
         y[0, 3] = -1
```

Agora vamos rodar o solver com 6 como candidato incial para k .

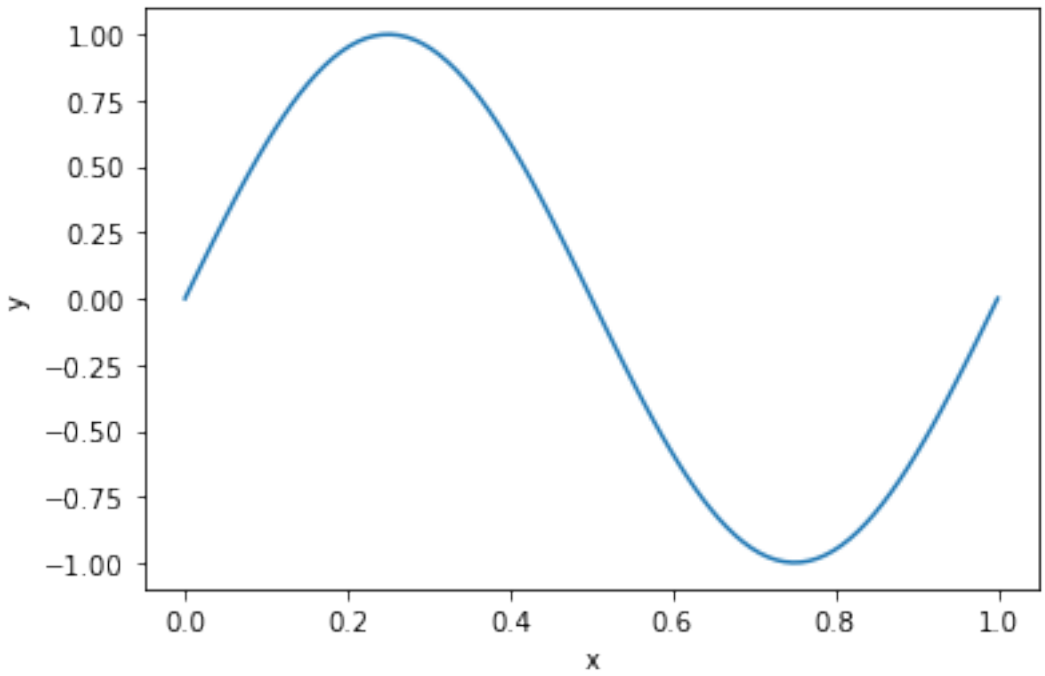
```
In [14]: sol = solve_bvp(fun, bc, x, y, p=[6])
```

Vemos que o valor de k encontrado é aproximadamente correto.

```
In [15]: sol.p[0]

Out[15]: 6.283294600464725
```

Por fim, vamos plotar a solução.

```
In [16]: x_plot = np.linspace(0, 1, 100)
         y_plot = sol.sol(x_plot)[0]
         plt.plot(x_plot, y_plot)
         plt.xlabel("x")
         plt.ylabel("y")
         plt.show()
```



5 O pacote solve_ivp: exemplos

6 Exemplo 1

https://pundit.pratt.duke.edu/wiki/Python:Ordinary_Differential_Equations/Examples

Suponha que desejamos resolver numericamente o seguinte PVI:

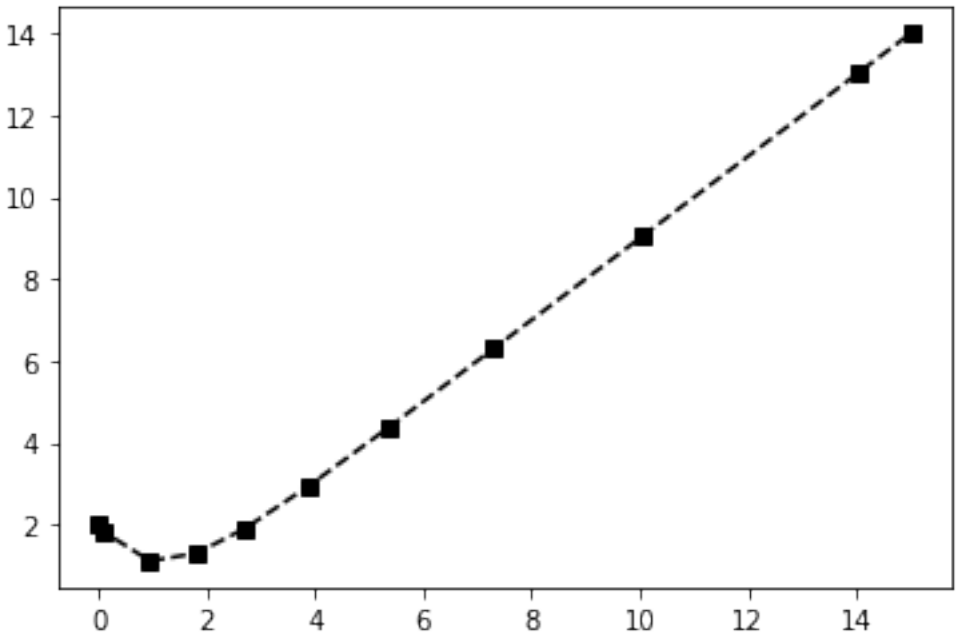
$$\frac{dy(t)}{dt} = t - y(t)$$

para $t \in [0, 15]$ e com condição inicial dada por $y(0) = 2$. Vamos fazer isso usando o solve_ivp do Python.

```
In [17]: import numpy as np
         from scipy.integrate import solve_ivp
         sol = solve_ivp(lambda t, y: t-y, [0, 15], [2])
```

Depois de rodar o solv_ivp, a solução numérica já foi calculada e contém 10 objetos diferentes:
sol.t é malha na variável t
sol.y é um array contendo a solução (cada coluna uma solução de y, se for vetorial)
Para plotar a solução usamos o matplotlib.

```
In [18]: print('t=', sol.t)

t= [ 0.          0.09222001  0.95192723  1.81163445  2.71266218  3.86465167
  5.34073095  7.30859458 10.05620309 14.06163429 15.        ]


In [19]: print('y=', sol.y)

y= [[ 2.          1.82793351  1.11039319  1.30219875  1.91202091  2.92787286
   4.35552188  6.3111682   9.05712136 13.0646847  14.00119445]]


In [20]: import matplotlib.pyplot as plt
         plt.plot(sol.t, sol.y[0], 'k--s')

Out[20]: [<matplotlib.lines.Line2D at 0x2a5fa682978>]
```



7 Exemplo 2

Considere o problema de valor inicial (IVP) dado por

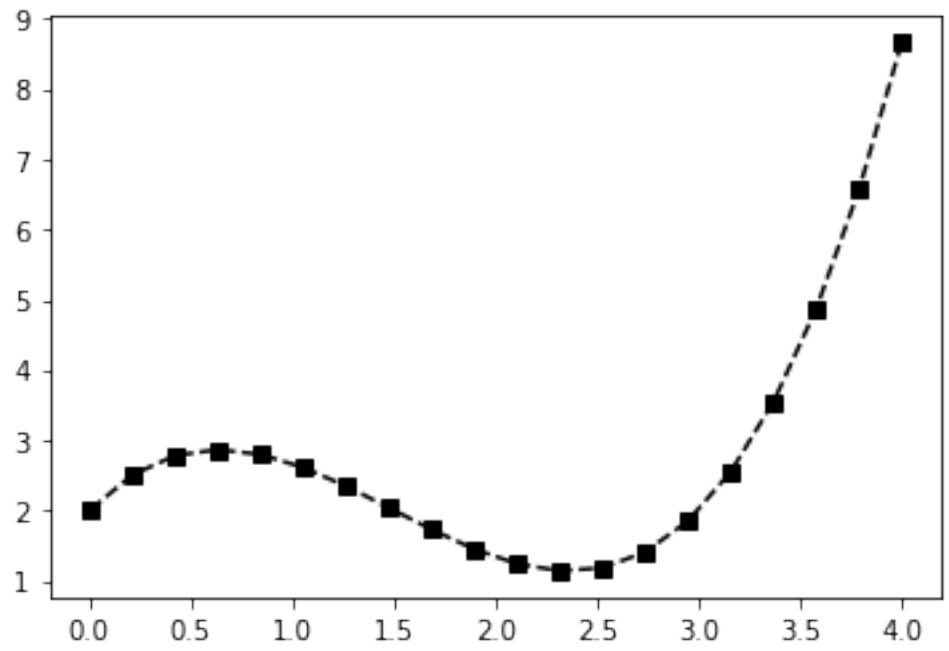
$$\frac{dy}{dt} = \alpha t^2 + \beta t + \gamma$$

com

$$y(0) = 2.$$

In [21]:
```
# %% Imports
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp

def fprime(t, y):
    return 2 * t**2 -6 * t + 3

def event(x, y):
    return y

sol = solve_ivp(fprime, (0, 4), np.array([2]), t_eval=np.linspace(0, 4, 20), events=[ev
plt.plot(sol.t, sol.y[0], 'k--s')
sol
```

Out[21]:
```
  message: 'The solver successfully reached the end of the integration interval.'
     nfev: 20
     njev: 0
      nlu: 0
      sol: None
   status: 0
  success: True
        t: array([0.        , 0.21052632, 0.42105263, 0.63157895, 0.84210526,
       1.05263158, 1.26315789, 1.47368421, 1.68421053, 1.89473684,
       2.10526316, 2.31578947, 2.52631579, 2.73684211, 2.94736842,
       3.15789474, 3.36842105, 3.57894737, 3.78947368, 4.        ])
 t_events: [array([], dtype=float64)]
        y: array([[2.        , 2.5048355 , 2.78106624, 2.86601545, 2.79700637,
        2.6113622 , 2.34640618, 2.03946153, 1.72785148, 1.44889926,
        1.23992808, 1.13826117, 1.18122175, 1.40613306, 1.85031832,
        2.55110074, 3.54580357, 4.87175001, 6.5662633 , 8.66666667]])
```




```
In [22]: print('t=', sol.t)

t= [0.         0.21052632 0.42105263 0.63157895 0.84210526 1.05263158
 1.26315789 1.47368421 1.68421053 1.89473684 2.10526316 2.31578947
 2.52631579 2.73684211 2.94736842 3.15789474 3.36842105 3.57894737
 3.78947368 4.        ]


In [ ]:
```