Modelo Lotka-Volterra: sistema presa-predador

Prof. Doherty Andrade

www.metodosnumericos.com.br

1 Introdução ao sistema presa-predador

Neste post vamos estudar o modelo matemático de Lotka-Volterra.

É também conhecido como sistema de equações diferenciais presa-predador baseado em https://scipy-cookbook.readthedocs.io/items/LoktaVolterraTutorial.html

O sistema é um par de equações diferenciais de primeira ordem, não lineares, frequentemente usado para descrever a dinâmica de interação de sistemas biológicos nos quais duas espécies interagem, um predador e o outro sua presa. Para facilitar a compreensão podemos imaginar que as presas são coelhos e os predadores são raposas.

O modelo foi proposto independentemente por Alfred J. Lotka em 1925 e Vito Volterra em 1926, e pode ser descrito por

$$
\begin{aligned}
\frac{du}{dt} &= au - buv \\
\frac{dv}{dt} &= -cv + dbuv
\end{aligned}
$$

com as seguintes notações:

u: número de presas (por exemplo, coelhos)

v: número de predadores (por exemplo, raposas)

a, b, c, d são parâmetros constantes que definem o comportamento das populações:

a: é a taxa de crescimento natural das presas (os coelhos), quando não há raposa

b: é a taxa natural de morte das presas (coelhos), devido à predação

c: é a taxa natural de morte dos predadores (a raposas), quando não há presas

d: é o fator que descreve quantos presas (coelhos) capturados permitem criar um novo predador (raposa).

Usaremos $X = [u, v]$ para descrever o estado de ambas as populações.

Definição das equações. Primeiramente identificamos e escrevemos a função do sistema.

2 Desenvolvimento

In [1]:
```
#! python
from numpy import *
```

```
import pylab as p
# Definição de parâmetros
a = 1.
b = 0.1
c = 1.5
d = 0.75
def dX_dt ( X , t = 0 ):
    "" "Retorna a taxa de crescimento das populações de raposas e coelhos . "" "
    return array ([ a * X [ 0 ] - b * X [ 0 ] * X [1 ] ,
                    - c * X [ 1 ] + d * b * X [ 0 ] * X [ 1 ] ])
```

3 Equilíbrio da população:

Antes de usar SciPy para integrar este sistema, daremos uma olhada mais de perto na posição de equilíbrio. O equilíbrio ocorre quando a taxa de crescimento é igual a 0, isto é, quando $\frac{dX}{dt} = 0$. Isso nos dá dois pontos:

$$X_{f0} = (0, 0)$$

e

$$X_{f1} = (\frac{c}{bd}, \frac{a}{b}).$$

```
In [2]: #! python
        X_f0 = array ([     0.  ,   0. ])
        X_f1 = array ([ c / ( d * b ), a / b ])
        all ( dX_dt ( X_f0 ) == zeros ( 2 ) ) and all ( dX_dt ( X_f1 ) == zeros (
        # => Verdadeiro

Out[2]: True
```

4 Estabilidade dos pontos fixos

Perto desses dois pontos, o sistema pode ser linearizado:

$$\frac{dX}{dt} = A_f * X$$

onde A é a matriz Jacobiana avaliada no ponto correspondente. Temos que definir a matriz Jacobiana:

```
In [3]: #! python
        def d2X_dt2 ( X , t = 0 ):
            "" "Retorna a matriz Jacobiana avaliada em X." ""
            return array ([[ a - b * X [ 1 ],   - b * X [ 0 ]     ],
                           [ b * d * X [ 1 ] ,   - c + b * d * X [ 0 ]]  ])
```

Perto de X_f0, que representa a extinção de ambas as espécies, temos:

```
In [4]: #! python
        A_f0  =  d2X_dt2 ( X_f0 )
        print('A_f0=',A_f0)
```

```
A_f0= [[ 1.  -0. ]
 [ 0.  -1.5]]
```

Perto de X_f0, o número de coelhos aumenta e a população de raposas diminui. A origem é, portanto, um ponto de sela.

Perto de X_f1, temos:

```
In [5]: #! python
        A_f1  =  d2X_dt2 ( X_f1 )
        # seus autovalores são +/- sqrt (c * a) .j:
        lambda1 ,  lambda2  =  linalg . eigvals ( A_f1 )
```

Como os autovalores são números imaginários, as populações de raposas e coelhos são periódicas, conforme segue a partir de análises adicionais . Seu período é dado por: T_f1 = 2 * pi / abs (lambda1)

```
In [6]: import numpy as np
        import matplotlib.pyplot as plt
        import scipy.linalg as la
        eigvals, eigvecs = la.eig(A_f1)
        print('autovalores:',eigvals)
        print('autovetores:', eigvecs)
```

```
autovalores: [0.+1.22474487j 0.-1.22474487j]
autovetores: [[-0.85280287+0.j          -0.85280287-0.j          ]
 [ 0.        +0.52223297j  0.        -0.52223297j]]
```

```
In [7]: #!Python
        T_f1  =  2 * pi / abs ( lambda1 )
        print('período:', T_f1)
```

```
período: 5.130199320647456
```

Integrando o ODE usando scipy.integrate Agora usaremos o módulo scipy.integrate para integrar os ODEs. Este módulo oferece um método denominado odeint, que é muito fácil de usar para integrar EDOs:

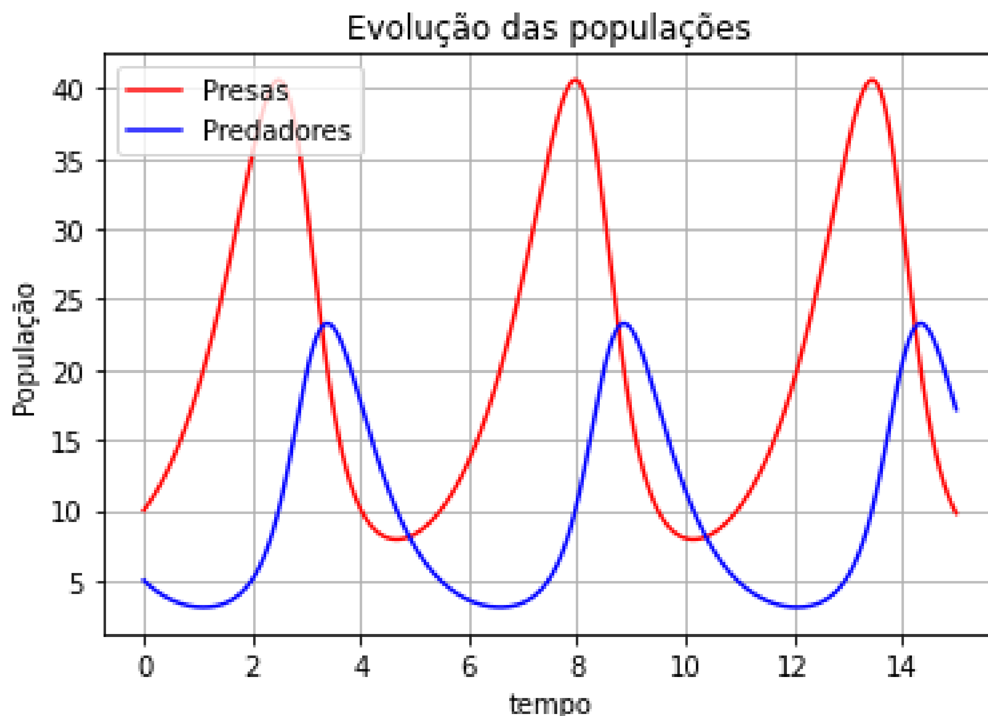
```
In [8]: #! python
        from scipy  import  integrate
        t  =  linspace ( 0 ,  15 ,   1000 )
        # tempo
        X0  =  array ([ 10 ,  5 ])
        # condições iniciais: 10 coelhos e 5 raposas
        X ,  infodict  =  integrate. odeint ( dX_dt ,  X0 ,  t ,  full_output = True )
        infodict [ 'message' ]
        # >>> 'Integração bem-sucedida.'
```

```
Out[8]: 'Integration successful.'
```

Infodict é opcional, e você pode omitir o argumento `full_output` se não quiser. Digite "info (odeint)" se quiser mais informações sobre entradas e saídas de odeint.

Agora podemos usar Matplotlib para traçar a evolução de ambas as populações:

```
In [9]: #!python
        rabbits, foxes = X.T
        f1 = p.figure()
        p.plot(t, rabbits, 'r-', label='Presas')
        p.plot(t, foxes  , 'b-', label='Predadores')
        p.grid()
        p.legend(loc='best')
        p.xlabel('tempo')
        p.ylabel('População')
        p.title('Evolução das populações')
        f1.savefig('presas_e_predadores_1.png')
```



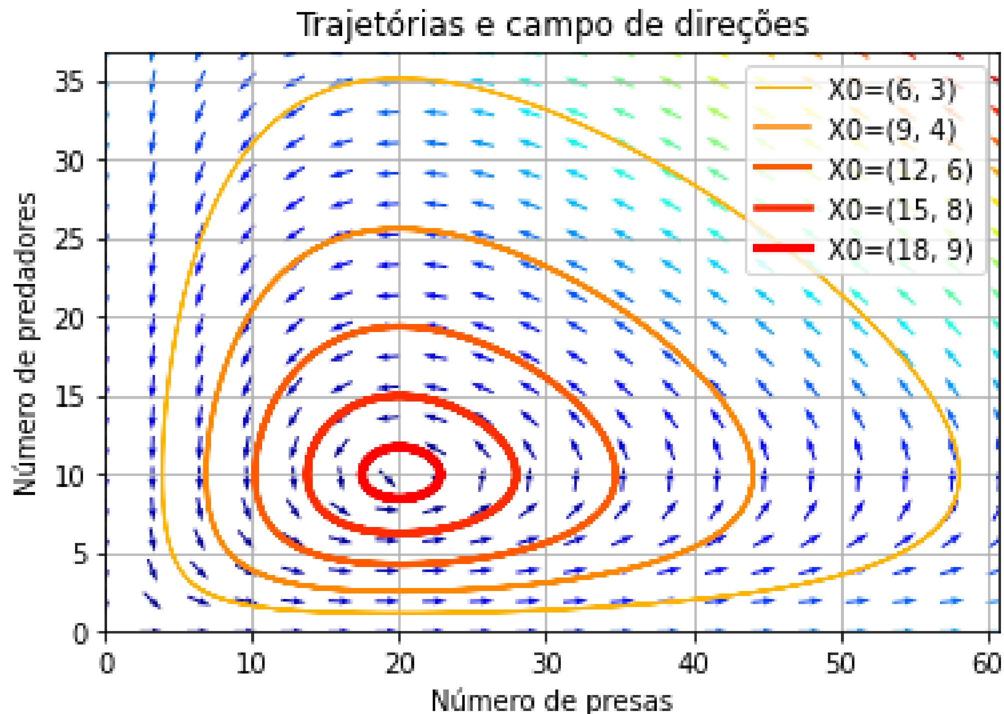
As populações são de fato periódicas e seu período é próximo ao valor T_f1 que calculamos.

Plotando campos de direção e trajetórias no plano de fase Plotaremos algumas trajetórias em um plano de fase para diferentes pontos de partida entre X_f0 e X_f1.

Usaremos o mapa de cores do Matplotlib para definir as cores das trajetórias. Esses mapas de cores são muito úteis para fazer gráficos bonitos. Dê uma olhada em ShowColormaps se quiser mais informações.

```
In [10]:  # valores de X0 entre  X_f0 e X_f1
         values  = linspace(0.3, 0.9, 5)
         vcolors = p.cm.autumn_r(linspace(0.3, 1., len(values)))  # cores de cada trajet

         f2 = p.figure()

         # plotando as  trajetorias
         for v, col in zip(values, vcolors):
             X0 = v * X_f1                               # ponto inicial
             X = integrate.odeint( dX_dt, X0, t)
             p.plot( X[:,0], X[:,1], lw=3.5*v, color=col, label='X0=(%.f, %.f)' % ( X0[0

         # definindo uma malha e calculando o campo de direções em cada ponto
         ymax = p.ylim(ymin=0)[1]
         xmax = p.xlim(xmin=0)[1]
         nb_points   = 20

         x = linspace(0, xmax, nb_points)
         y = linspace(0, ymax, nb_points)

         X1 , Y1  = meshgrid(x, y)                       # cria a malha
         DX1, DY1 = dX_dt([X1, Y1])                      # calcula a taxa de crescimento
         M = (hypot(DX1, DY1))                           # norma da taxa de crescimento
         M[ M == 0] = 1.                                 # evita erro de divisão por zer
         DX1 /= M                                        # Normaliza cada feclha do camp
         DY1 /= M

         #-------------------------------------------------------
         # Desenhando o campo de direções.
         # Usamos cores para dar informações sobre a velocidade de crescimento
         p.title('Trajetórias e campo de direções')
         Q = p.quiver(X1, Y1, DX1, DY1, M, pivot='mid', cmap=p.cm.jet)
         p.xlabel('Número de presas')
         p.ylabel('Número de predadores')
         p.legend()
```

```
p.grid()
p.xlim(0, xmax)
p.ylim(0, ymax)
f2.savefig('presas_e_predadores_2.png')
```



Este gráfico nos mostra que mudar a população de raposas ou coelhos pode ter um efeito não intuitivo. Se, para diminuir o número de coelhos, introduzirmos raposas, isso pode levar a um aumento de coelhos a longo prazo, dependendo do tempo de intervenção.

Traçando contornos Podemos verificar que a função IF definida abaixo permanece constante ao longo de uma trajetória:

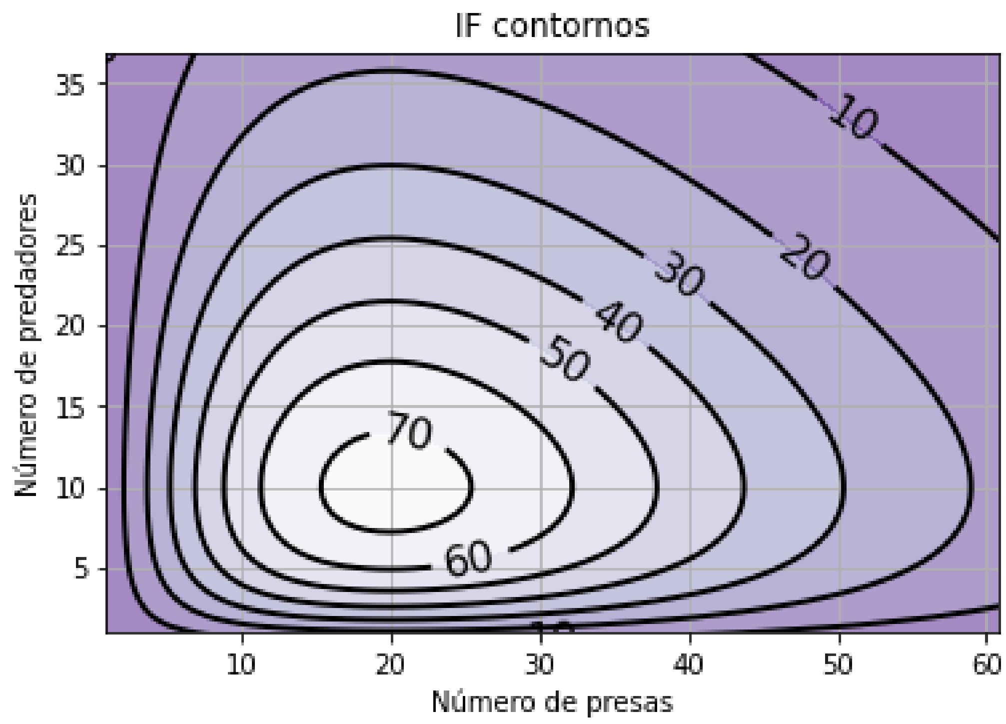
```
In [11]: #!python
         def IF(X):
             u, v = X
             return u**(c/a) * v * exp( -(b/a)*(d*u+v) )
         #Vamos verificar que  IF permanece constante para diferentes trajetórias
         for v in values:
             X0 = v * X_f1                              # ponto incial
             X = integrate.odeint( dX_dt, X0, t)
             I = IF(X.T)                                # calcula IF ao longo da trajet
             I_mean = I.mean()
             delta = 100 * (I.max()-I.min())/I_mean
```

```
    print ('X0=(%2.f,%2.f) => I ~ %.1f |delta = %.3G %%' % (X0[0], X0[1], I_mea
#     X0=( 6, 3) => I ~ 20.8 |delta = 6.19E-05 %
#     X0=( 9, 4) => I ~ 39.4 |delta = 2.67E-05 %
#     X0=(12, 6) => I ~ 55.7 |delta = 1.82E-05 %
#     X0=(15, 8) => I ~ 66.8 |delta = 1.12E-05 %
#     X0=(18, 9) => I ~ 72.4 |delta = 4.68E-06 %
```

```
X0=( 6, 3) => I ~ 20.8 |delta = 6.19E-05 %
X0=( 9, 4) => I ~ 39.4 |delta = 2.67E-05 %
X0=(12, 6) => I ~ 55.7 |delta = 1.82E-05 %
X0=(15, 8) => I ~ 66.8 |delta = 1.12E-05 %
X0=(18, 9) => I ~ 72.4 |delta = 4.68E-06 %
```

Traçar isocontornos de IF pode ser uma boa representação de trajetórias, sem ter que integrar a ODE.

```
In [12]: # plot iso contours
         nb_points = 80                              # grid size
         x = linspace(0, xmax, nb_points)
         y = linspace(0, ymax+2, nb_points)
         X2 , Y2  = meshgrid(x, y)                   # create the grid
         Z2 = IF([X2, Y2])                           # compute IF on each point
         f3 = p.figure()
         CS = p.contourf(X2, Y2, Z2, cmap=p.cm.Purples_r, alpha=0.5)
         CS2 = p.contour(X2, Y2, Z2, colors='black', linewidths=2. )
         p.clabel(CS2, inline=1, fontsize=16, fmt='%.f')
         p.grid()
         p.xlabel('Número de presas')
         p.ylabel('Número de predadores')
         p.ylim(1, ymax)
         p.xlim(1, xmax)
         p.title('IF contornos')
         f3.savefig('presas_e_predadores_3.png')
         p.show()
```


IF contornos
Número de predadores
Número de presas
10
20
30
40
50
60
70
35
30
25
20
15
10
5
10
20
30
40
50
60