Lista 2 - Controle de Sistemas Dinâmicos

Solução Computacional de Equações Diferenciais

Aluno: Frederico José Ribeiro Pelogia

RA: 133619

Turma: N

Departamento de Ciência e Tecnologia
Universidade Federal de São Paulo
Dezembro de 2020

Sumário

Consideração sobre as Simulações

O presente relatório apresenta soluções às questões propostas na segunda lista de exercícios da unidade curricular Controle de Sistemas Dinâmicos, utilizando como material de apoio a apostila [2]. As simulações computacionais utilizaram a biblioteca científica **scipy** [3] da linguagem de programação Python. Para cada sistema simulado será apresentada uma breve discussão sobre as EDOs, um diagrama de blocos, o código na íntegra, os gráficos gerados e uma discussão sobre os resultados.

Sobre a biblioteca Scipy

Para a integração numérica das EDOs e solução dos problemas de valor inicial associados aos modelos simulados, foi utilizada a função `scipy.integrate.solve_ivp()`. Essa função utiliza, por padrão, um método RK45 [1] (Runge-Kutta) e integra um sistema de equações diferenciais ordinárias, dadas condições iniciais, recebendo-o na forma

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

$$y(0) = y0.$$

A variável y pode ser um vetor com diferentes equações diferenciais, que serão integradas simultaneamente. Assim, para resolver sistemas de ordem superior com a função em questão, é recomendado explicitar as relações entre cada derivada. Por exemplo, uma EDO de segunda ordem da forma $\ddot{x} = a\dot{x} + bx$ deveria ser enviado como

$$\frac{d}{dt}\begin{pmatrix}\dot{x}\\ x\end{pmatrix} = \begin{pmatrix}a\dot{x} + bx\\ \dot{x}\end{pmatrix}$$

A Figura 1 apresenta, em detalhe, os argumentos que devem ser passados à função.

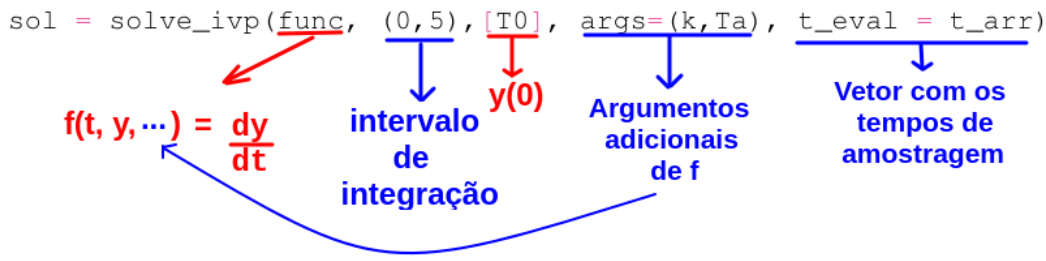

Figura 1: Argumentos da função `solve_ivp`. Fonte: Autor

1 Newton's Law of Cooling

O primero exercício propõe a resolução do seguinte problema:

A temperatura de dentro da sua casa é de **70°F** e a do lado de fora (ambiente) é de **30°F**. Às **1:00** a fornalha parou de funcionar. Às **3:00** a temperatura na casa caiu para **50°F**.

Assumindo que a temperatura de fora da casa é constante e que a Lei do Resfriamento de Newton se aplica, determine quando a temperatura de dentro da sua casa atingirá **40°F.**

A Lei do Resfriamento de Newton é descrita pela EDO

$$\frac{dT}{dt} = -k(T(t) - T_a),$$

que revela que a variação temporal da temperatura de um corpo em um certo instante é proporcional à diferença da temperatura do mesmo com a temperatura dos seus arredores [2]. Assim, pode-se também escrever a lei como

$$T(t) = T_a + (T_0 - T_a)e^{-kt}.$$

A apostila [2] apresenta, na Figura 2, uma representação da lei como um diagrama de blocos.

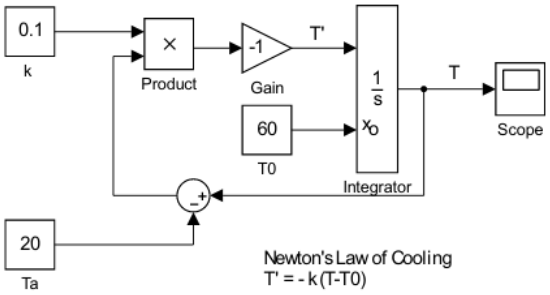

Figura 2: Diagrama de Blocos para a Lei do Resfriamento de Newton. Fonte: [2]

Embora o problema seja simples e o valor da constante k pudesse ser obtido analiticamente, foi optado por descobri-lo por tentativa e erro, através dos resultados das simulações. O código referente ao modelo da Lei do Resfriamento de Newton está apresentado no Listing 1.

```
1  #Imports
2  import numpy as np
3  from scipy.integrate import solve_ivp
4  import matplotlib.pyplot as plt
5
6  #Define constants
7  T0 = 70 # Initial temperature inside (F)
8  Ta = 30 # Initial temperature outside (F)
9  k = 0.35# cooling constant (hours^(-1))
10
11 # Teste 1: k = 0.5
12 # Teste 2: k = 0.25
13 # Teste 3: k = 0.45
14 # Teste 4: k = 0.35
15
16 t_arr = np.linspace(0,5,1000) # array with sampling times
17
```

```
# Newton's Law of Cooling)
# T' = -k(T - Ta)
# dy/dy = func(...)
# y = [T]
def func(t,y,k,Ta):
    T = y
    return -k*(T - Ta)

#Solve the ODE using scipy.integrate.solve_ivp (initial value problem)
sol = solve_ivp(func, (0,5),[T0], args=(k,Ta), t_eval = t_arr)

#Plot Solution
fig, ax = plt.subplots()
ax.plot(sol.t, sol.y[0])
ax.set(xlabel='Elapsed Time (h)', ylabel='Temperature (F)',
       title='Temperature Decay')
ax.grid()

#Search time with T ~= 40 F
diff = (sol.y[0]-40)**2
sample_40f = np.where( diff == np.amin(diff))[0][0]
t_40f = sol.t[sample_40f]
ax.scatter(t_40f, sol.y[0][sample_40f], c = 'red', label =
           f'T({t_40f:.2f} h) = {sol.y[0][sample_40f]:.2f} F')
ax.legend()
plt.show()
print(f'\nThe temperature got closer to 40 F at time t = {t_40f:.2f} h\n')
```

Code Listing 1: Código para a Lei de Resfriamento de Newton

A Figura 3 apresenta os resultados das simulações para diferentes valores de k. A constante k recebeu inicialmente o valor 0.5 e depois foi sendo refinada até que a temperatura após 2 horas fosse de 50^{o}F, como no enunciado.

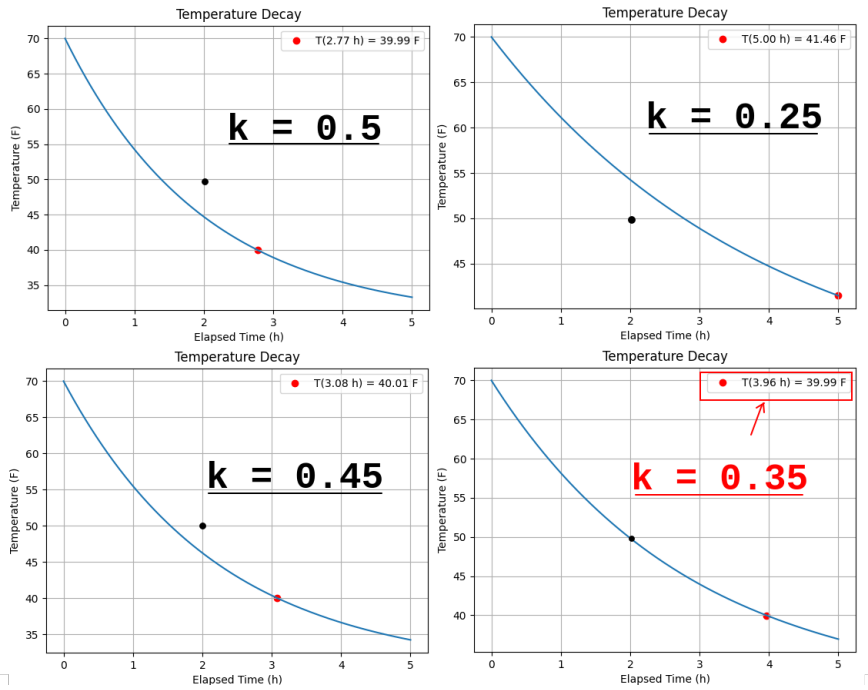

Figura 3: Simulação para a Lei do Resfriamento de Newton, refinando o valor da constante k. Fonte: Autor

Da Figura 3, perceba que o valor que mais satisfez a condição inicial, demarcada pelo círculo preto, foi $k = 0.35$. Para esse teste, temos que o interior da casa atinge uma temperatura próxima de 40°F após 3.96 $h \approx 3$ h 58 min.

Assim, a temperatura de interesse foi atingida às **4:58**.

2 Projectile Motion Model

O segundo exercício envolve a simulação do movimento de um projétil lançado ao ar e que está sujeito ao atrito do ar, conhecido por arrasto (*drag*) e usualmente representado pelo coeficiente k. A Figura 4, retirada da apostila [2], representa o modelo por um diagrama de blocos.

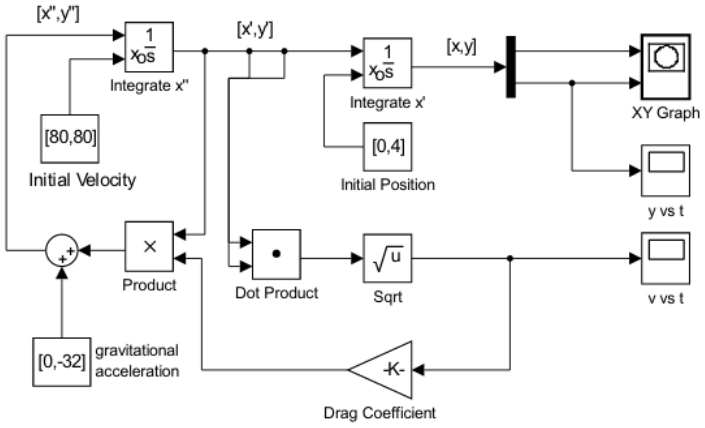

Figura 4: Diagrama de Blocos para o modelo de lançamento de projétil. Fonte: [2]

Note que as integrações são feitas simultaneamente, com a utilização de um vetor com a velocidade em x e em y. Sendo x e y a posição do projétil em x e em y.

$$\begin{pmatrix} \ddot{x} \\ \ddot{y} \end{pmatrix} = - \begin{pmatrix} g_x \\ g_y \end{pmatrix} - k\sqrt{\dot{x}^2 + \dot{y}^2} \begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix}$$

A passagem do sistema de EDOs para a simulação será feita de forma semelhante, mas com a adição da posição em x e y ao vetor, para informar que a derivada desses valores corresponde à velocidade do projétil no respectivo eixo. Portanto, a formulação utilizada para o código será

$$\frac{d}{dt} \begin{pmatrix} \dot{x} \\ \dot{y} \\ x \\ y \end{pmatrix} = \begin{pmatrix} -g_x - k\dot{x}\sqrt{\dot{x}^2 + \dot{y}^2} \\ -g_x - k\dot{y}\sqrt{\dot{x}^2 + \dot{y}^2} \\ \dot{x} \\ \dot{y} \end{pmatrix}.$$

O Listing 2 apresenta o código na íntegra.

```
#Imports
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt

def projectile_motion(V0, neg_g, pos0, drag, t_arr):
    #V0: initial (x, y) velocity (m/s)
    #neg_g: -1*gravitational acceleration (m/s^2)
    #pos0: initial position (m)
    #drag : air drag
    #t_arr : array with sampling times

    # Define the ODE
    # dy/dt = func(...)
    # y = [vx, vy, posx, posy]
    def func(t, y, neg_g, drag):
        acc_x = neg_g[0] - y[0]*drag*np.sqrt(y[0]**2 + y[1]**2)
```

```
        acc_y = neg_g[1] - y[1]*drag*np.sqrt(y[0]**2 + y[1]**2)
        return [acc_x, acc_y, y[0], y[1]]


    #Solve the ODE using scipy.integrate.solve_ivp (initial value problem)
    sol = solve_ivp(func, [0,50],[V0[0], V0[1], pos0[0], pos0[1]],args=(neg_g,
                                            drag)
    , t_eval = t_arr)

    v_mag = np.sqrt(sol.y[0]**2 + sol.y[1]**2)

    # Plotting
    fig, axs = plt.subplots(1, 3, figsize=(14,4))
    fig.suptitle(f'Projectile Motion Model: V0 = {V0}, pos0 = {pos0}, drag = {
                                            drag}, neg_g = {neg_g}')
    above_floor = sol.y[3] >= 0
    axs[1].plot(sol.t[above_floor], sol.y[3][above_floor])
    axs[1].set(xlabel='Elapsed Time(s)', ylabel='Y Position (m)')
    axs[1].grid()
    axs[0].plot(sol.t[above_floor], v_mag[above_floor])
    axs[0].set(xlabel='Elapsed Time (s)', ylabel='Velocity Magnitude (m/s)')
    axs[0].grid()
    axs[2].plot(sol.y[2][above_floor], sol.y[3][above_floor])
    axs[2].set(xlabel='X Position (m)', ylabel='Y Position (m)')
    axs[2].grid()
    plt.show()

# ======================================================================
# Define inputs
V0 = [24.384, 24.384] #(80 ft/s ~ 24.384 m/s)
neg_g = [0, -9.7536] #(32 ft/s^2 ~ 9.7536 m/s^2)
pos0 = [0, 1.2192]#(4ft ~ 1.2192 m)
drag = 0.25
t_arr = np.linspace(0,50,10000)

projectile_motion(V0, neg_g, pos0, drag, t_arr)
```

Code Listing 2: Código para o modelo de lançamento de projétil

Para cada um dos testes realizados, foram gerados 3 gráficos:

- Tempo x Magnitude da Velocidade. Onde a magnitude da velocidade é $\sqrt{\dot{v}^T\dot{v}} = \sqrt{\dot{x}^2 + \dot{y}^2}$
- Tempo x Posição em Y (altitude).
- Posição em X x Posição em Y.

A Figura 5 apresenta testes com um lançamento oblíquo utilizando diferentes valores para o coeficiente de arrasto.

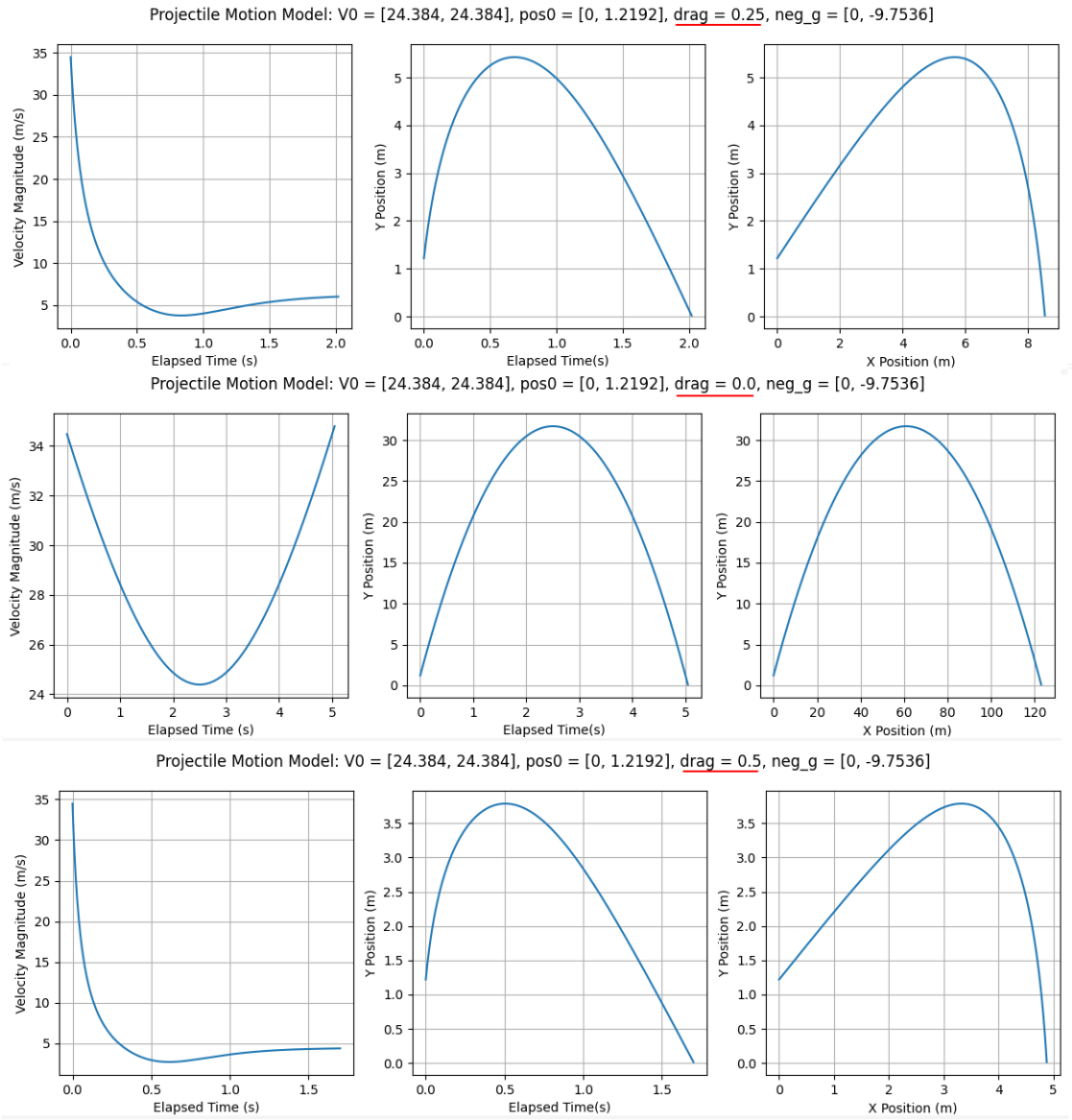

Figura 5: Simulações com lançamento oblíquo do projétil. Fonte: Autor

Note que o caso com coeficiente de arrasto nulo apresenta tempo de subida igual ao de queda. O módulo da velocidade, nesse caso, tem o comportamento de uma parábola com concavidade para cima, descrescendo até o instante em que o projétil atinge a altura máxima e aumentando na mesma taxa após isso. O cenário sem arrasto foi o que obteve a maior distância até a posição do pouso, de pouco mais de 120 metros, atingindo a altura máxima na metade dessa distância.

Os lançamentos feitos com arrasto não nulo obtiveram um tempo de subida menor do que o de descida. A força de arrasto também proporcionou ao projétil uma queda mais íngreme, embora mais lenta. Pode-se perceber, também, que a inserção do atrito do ar no modelo reduz drasticamente a distância horizontal total do lançamento. Os testes com

`drag =0.25` e `drag =0.5` obtiveram distância total próxima de 8 metros e 5 metros, respectivamente.

Por fim, a Figura 6 exibe simulações que envolvem lançamentos horizontais, isto é, zerando a componente vertical da velocidade inicial do projétil. Também foi definida uma altura maior, de $10m$, para a posição inicial do lançamento, além de uma velocidade de 80 m/s.

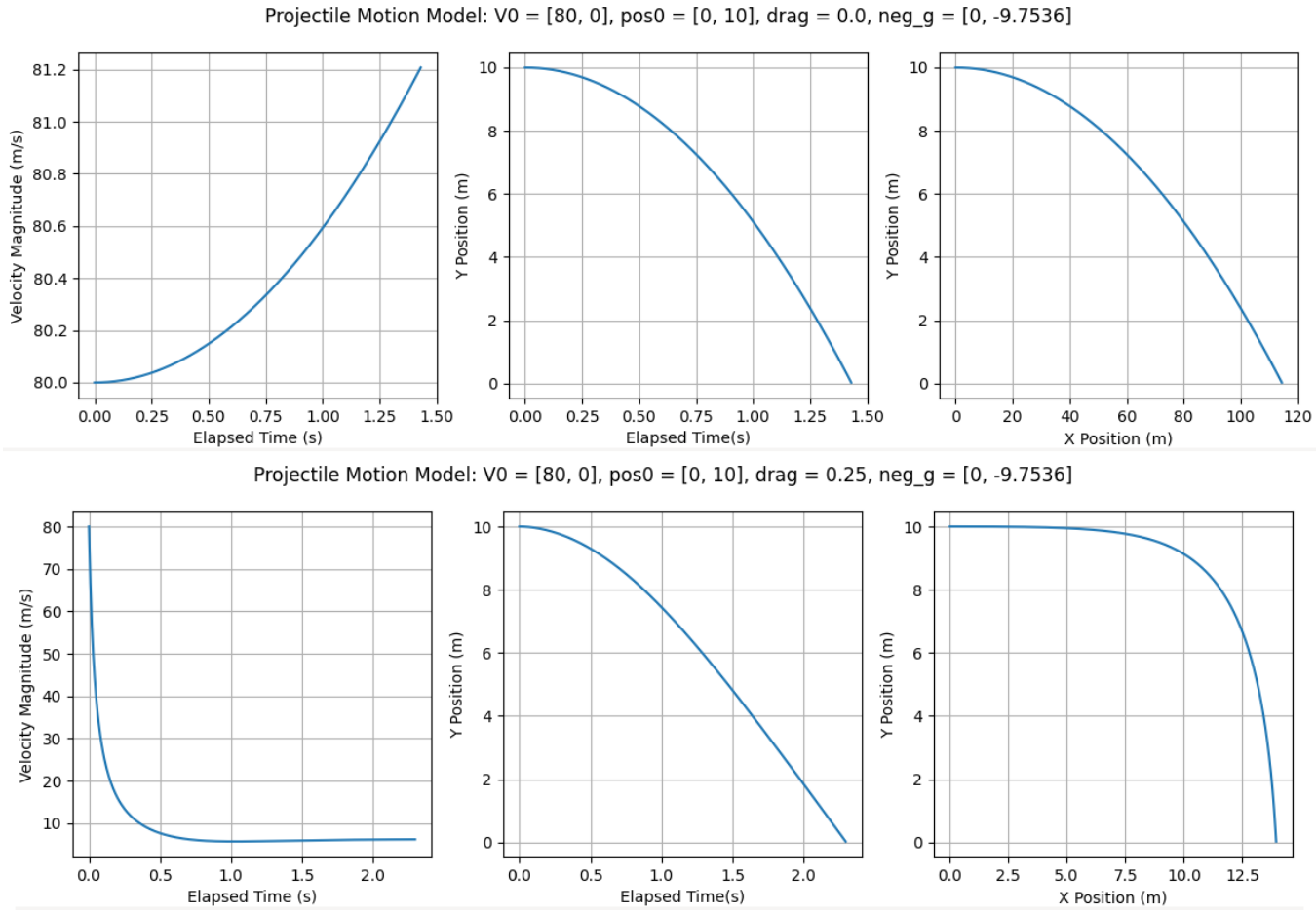

Figura 6: Simulações com lançamento horizontal do projétil. Fonte: Autor

Houveram duas principais diferenças perceptíveis pelos gráficos da Figura 6. A primeira é que, como no caso sem atrito a única força agindo sobre o projétil é a força peso, a magnitude da velocidade tende a aumentar, afinal o projétil está sendo acelerado verticalmente para baixo. Enquanto isso, no caso em que considera-se o atrito do ar, a força de arrasto aumenta proporcionalmente à velocidade, até que iguala-se à força peso, fazendo com que a velocidade do objeto convirja para um valor que mantém-se constante até o pouso.

3 LRC Circuit

Considere um circuito RLC com $L = 1.00\ H$, $R = 1.00 \times 10^2\ \Omega$, $C = 1.00 \times 10^{-4}\ F$, e $V = 1.00 \times 10^3\ V$. Suponha que não há carga no capacitor e nem corrente circulando no

sistema quando uma bateria é inserida no instante $t = 0$. Use um modelo computacional para descobrir a corrente e a carga no capacitor em função do tempo. Descreva como o sistema se comporta com decorrer do tempo.

Considerando a Lei de Ohm

$$v_R(t) = Ri(t),$$

e algumas relações básicas de circuitos elétricos

$$v_L(t) = L\,\frac{di(t)}{dt}, \quad v_L(t) = L\,\frac{di(t)}{dt},$$

$$i(t) = C\frac{dv_C(t)}{dt}, \qquad v_c(t) = \frac{1}{C}\,q(t),$$

podemos, a partir da Lei das Tensões de Kirchoff, obter

$$v(t) = R\,i(t) + L\,\frac{di(t)}{dt} + \frac{1}{C}\,q(t)$$

$$\implies v(t) = \frac{1}{C}\,q(t) + R\,\dot{q}(t) + L\,\ddot{q}(t)$$

$$\implies \ddot{q}(t) = \frac{1}{L}\left(v(t) - \frac{1}{C}\,q(t) - R\,\dot{q}(t)\right).$$

A Figura 7 apresenta um diagrama de blocos desenhado para a EDO de 2ª ordem obtida.

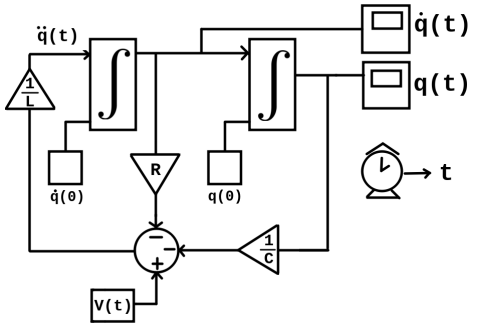

Figura 7: Diagrama de blocos para o Circuito RLC. Fonte: Autor.

Assim, podemos montar a função

$$\implies \begin{pmatrix}\dot{q}(t)\\ q(t)\end{pmatrix} = \begin{pmatrix}\frac{1}{L}\left(v(t) - \frac{1}{C}\,q(t) - R\,\dot{q}(t)\right)\\ q(t)\end{pmatrix},$$

que pode ser introduzida no modelo computacional, cujo código está disposto no Listing 3.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from scipy.integrate import solve_ivp
4
5 # define constants
6 L = 1.00 # Inductance (Henry)
```

```
7  R = 100 # Resistance (Ohm)
8  C = 0.0001 # Capacitance (Farad)
9
10 V = lambda t: 1000 # test 1 : constant Voltage (Volt)
11 #V = lambda t: 5*np.cos(t) # test 2 : sinusoidal
12
13 t_arr = np.linspace(0,0.5,200)# array with sampling times
14
15 # Define the ODE
16 # dy/dt = func(...)
17 # y = [i(t), q(t)]
18 def func(t, y, L, R, C, V):
19     return [(V(t) - y[1]/C - R*y[0])/L, y[0]]
20
21 #Solve the ODE using scipy.integrate.solve_ivp (initial value problem)
22 sol = solve_ivp(func, [0,0.5], [0.0, 0.0], args=(L,R,C,V), t_eval = t_arr)
23
24 fig, axs = plt.subplots(1, 2, figsize=(12,4))
25 fig.suptitle('Electric Current and Charge on the RLC System')
26 axs[0].plot(sol.t, sol.y[0])
27 axs[0].set(xlabel='Elapsed Time(s)', ylabel='Electric Current (A)')
28 axs[0].grid()
29 axs[1].plot(sol.t, sol.y[1])
30 axs[1].set(xlabel='Elapsed Time(s)', ylabel='Electric Charge (C)')
31 axs[1].grid()
32 plt.show()
```

Code Listing 3: Código para a simulação do circuito RLC

O primeiro teste realizado para este sistema foi utilizando os valores fornecidos no enunciado, com uma fonte de tensão constante de 1000 V. O resultado para esse caso é apresentado na Figura 8.

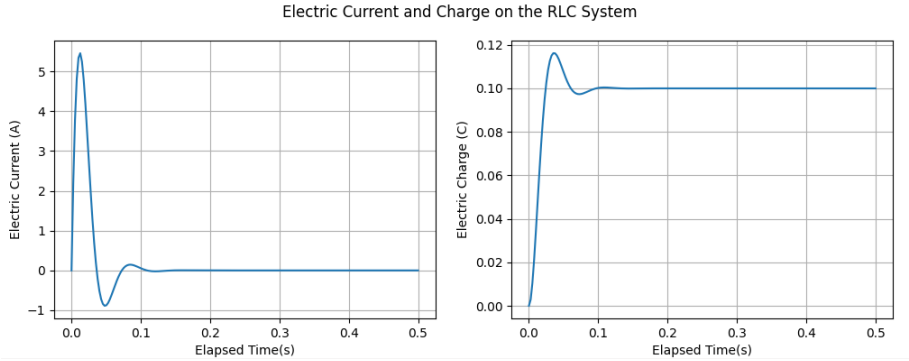

Figura 8: Simulação computacional do Circuito RLC com tensão elétrica constante. Fonte: Autor

Pode-se notar que a tensão constante faz com que a corrente e a carga tenham comportamento constante com o decorrer do tempo, com exceção dos primeiros 0.1 segundos. Nestes primeiros instantes, a variação na corrente e na carga do sistema ocorre como reação à inserção repentina da fonte de tensão (bateria), sendo este comportamento conhecido por

transiente ou resposta ao degrau. O comportamento é coerente com as condições inicais nulas definidas.

Foi também feito um experimento inserindo-se uma tensão oscilatória no sistema. Isso poderia ser feito, por exemplo, substituindo a bateria por um gerador de funções. Assim, tomando $V(t) = 1000cos(t)$, temos os resultados apresentados na Figura 9. Note que os gráficos da parte de cima abordam os 0.5 segundos iniciais de simulação e os de baixo abordam os primeiros 10 segundos, tendo sido alterado devidamente o intervalo de integração no código.

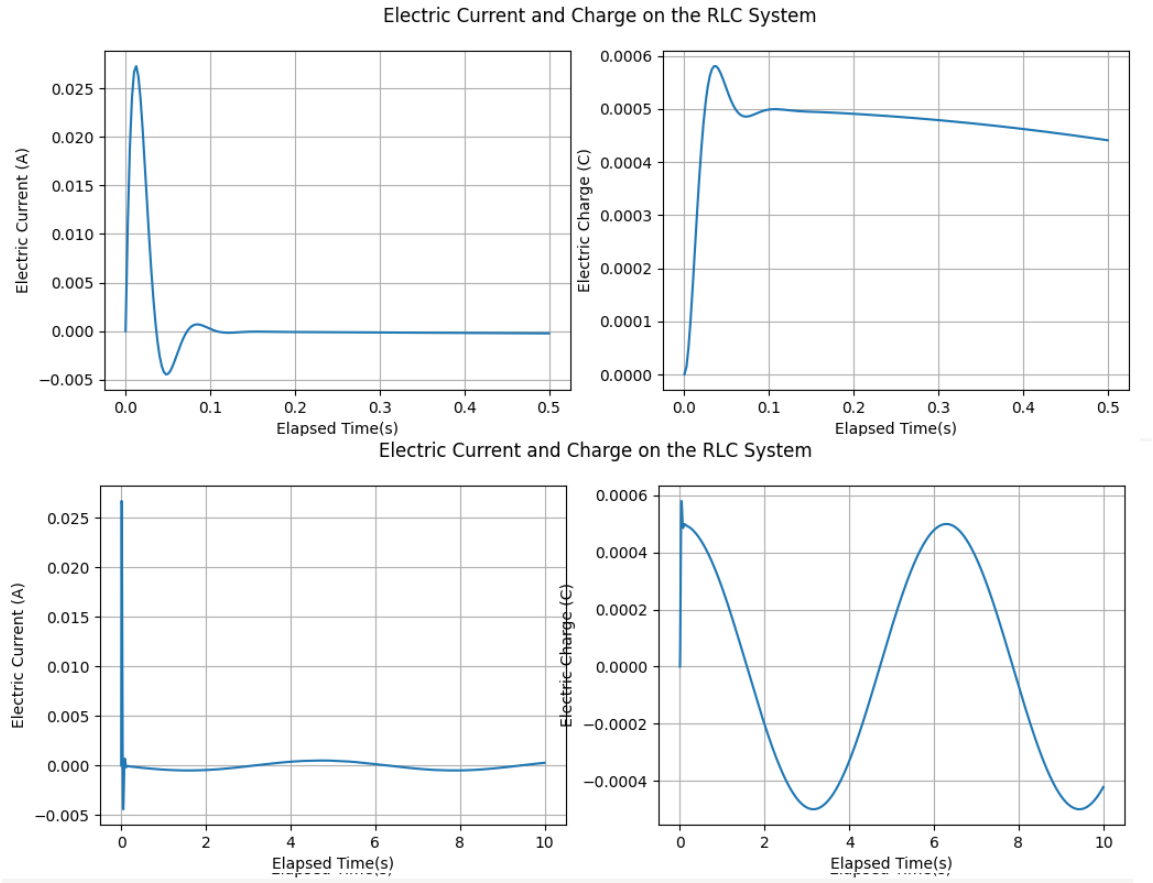

Figura 9: Simulação computacional do Circuito RLC com tensão de entrada $V(t) = 1000cos(t)$. Fonte: Autor

Perceba que o mesmo comportamento inicial, referente à resposta ao degrau ocorre quando a fonte é ligada. Entretanto, com a entrada senoidal, a corrente e a carga tendem a oscilar com período constante, o que é conhecido por regime permanente senoidal. O comportamento também está coerente com as condições iniciais nulas e o funcionamento real desse tipo de circuito.

4 Plastic Ball Motion Model

Um certo modelo do movimento de uma leve bola de plástico arremessada verticalmente no ar é dado por

$$m\ddot{x} + c\dot{x} + mg = 0, \quad x(0) = 0, \quad \dot{x}(0) = v_0.$$

Aqui, m é a massa da bola, $g = 9.8\ m\ s^{-2}$ é a aceleração da gravidade e c representa o arrasto proveniente do atrito com o ar. Como não há a presença do termo x (altura), pode-se traduzir o modelo para uma equação diferencial de primeira ordem para a velocidade $v(t) = x'(t)$:

$$m\dot{v} + cv + mg = 0.$$

$$\implies \dot{v} = \frac{1}{m}(-cv - mg)$$

Um desenho do diagrama de blocos para essa EDO está apresentada na Figura 10.

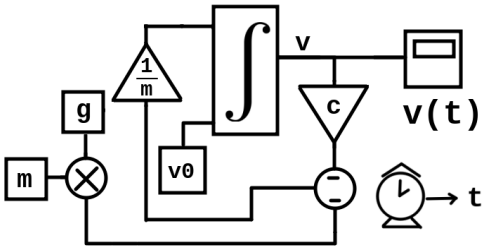

Figura 10: Diagrama de blocos para o modelo de primeira ordem do lançamento da bola de plástico. Fonte: Autor

O código para o modelo de primeira ordem está apresentado no Listing 4.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp

def plastic_ball(m_in,c_in,v0_in):

    # Define constants
    m = m_in # mass (kg)
    g = 9.8 # gravitational acceleration (m / s^2)
    c = c_in # damping constant (kg/s)
    v0 = v0_in # velocity (m/s)
    t_arr = np.linspace(0,10,1000) # array with sampling times (s)

    #dy/dt = func(...)
    # y = [v]
    def func(t, y, m, g, c):
        v = y
        return (-c*v - m*g)/m

    # Solve the ODE using scipy.integrate.solve_ivp (initial value problem)
    sol = solve_ivp(func, [0,10],[v0],args=(m, g, c), t_eval=t_arr)

    # Find maximum height
    max_h_sample = np.where(sol.y[0] == min(sol.y[0], key=abs))
```

```
25
26      # Plotting
27      fig, ax = plt.subplots()
28      ax.plot(sol.t, sol.y[0], zorder=1)
29      ax.scatter(sol.t[max_h_sample], 0,c ='red' , label = f'T( {sol.t[
                                                  max_h_sample][0]:.3f} s) ~= 0',
                                                  zorder=2)
30      ax.set(xlabel='Elapsed Time(s)', ylabel='Velocity (m/s)',
31         title=f'Plastic Ball Motion: m = {m}, g = {g}, c = {c}, v0 = {v0}')
32      ax.legend()
33      ax.grid()
34      plt.show()
35
36  #================================================================
37  #Define inputs
38  m = 0.25
39  g = 9.8
40  c = 0.10  #damping
41  v0 = 50
42  pos0 = 0.0 # initial position (m)
43
44  plastic_ball(m,c,v0)
```

Code Listing 4: Código para o modelo de lançamento vertical de uma bola leve de plástico

A Figura 11 apresenta uma comparação de dois testes feitos: um considerando o atrito do ar, controlado pela constante c, e outro desprezando-o.

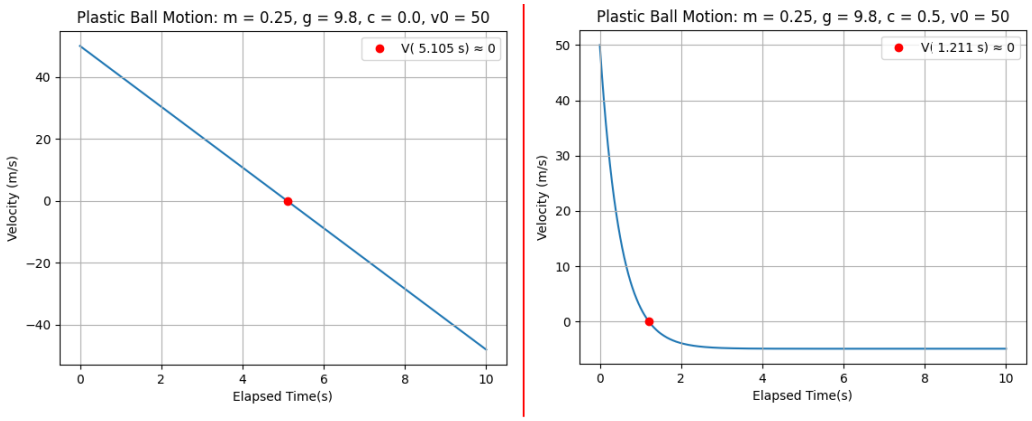

Figura 11: Simulação computacional do lançamento vertical de uma bola de plástico com e sem arrasto. Fonte: Autor

Perceba que o teste sem arrasto apresenta tempo de subida igual ao de descida, pois a velocidade troca de sinal exatamente na metade do tempo em que ela retorna (em módulo) ao seu valor inicial. Neste cenário, a velocidade decresce linearmente na subida e cresce linearmente na descida. Já no caso com atrito do ar, a bola, enquanto está caindo, para de acelerar e tende a uma velocidade constante, conhecida por **velocidade terminal**. Esse fenômeno ocorre justamente pois o termo de atrito é proporcional à velocidade e, ao mesmo tempo, controla sua variação.

A questão também pede os gráficos da posição para o caso em que $\frac{c}{m} = 5s^{-1}$. Para isso, modificaremos o código para simular o modelo com a EDO de segunda ordem

$$m\ddot{x} + c\dot{x} + mg = 0, \quad x(0) = 0, \quad \dot{x}(0) = v_0.$$

As principais modificações necessárias no código estão apresentadas no Listing 5, sem considerar as alterações na montagem dos gráficos.

```
#dy/dt = func(...)
# y = [v,pos]
def func(t, y, m, g, c):
    v = y[0]
    return [(-c*v - m*g)/m, v]

# Solve the ODE using scipy.integrate.solve_ivp (initial value problem)
sol = solve_ivp(func, [0,3],[v0,0.0],args=(m, g, c), t_eval=t_arr)
```

Code Listing 5: Modificações no código do lançamento da bola de plástico

Com o novo modelo, podemos acompanhar a velocidade e a altura da bola nos testes com $\frac{c}{m} = 5s^{-1}$, apresentados na Figura 12.

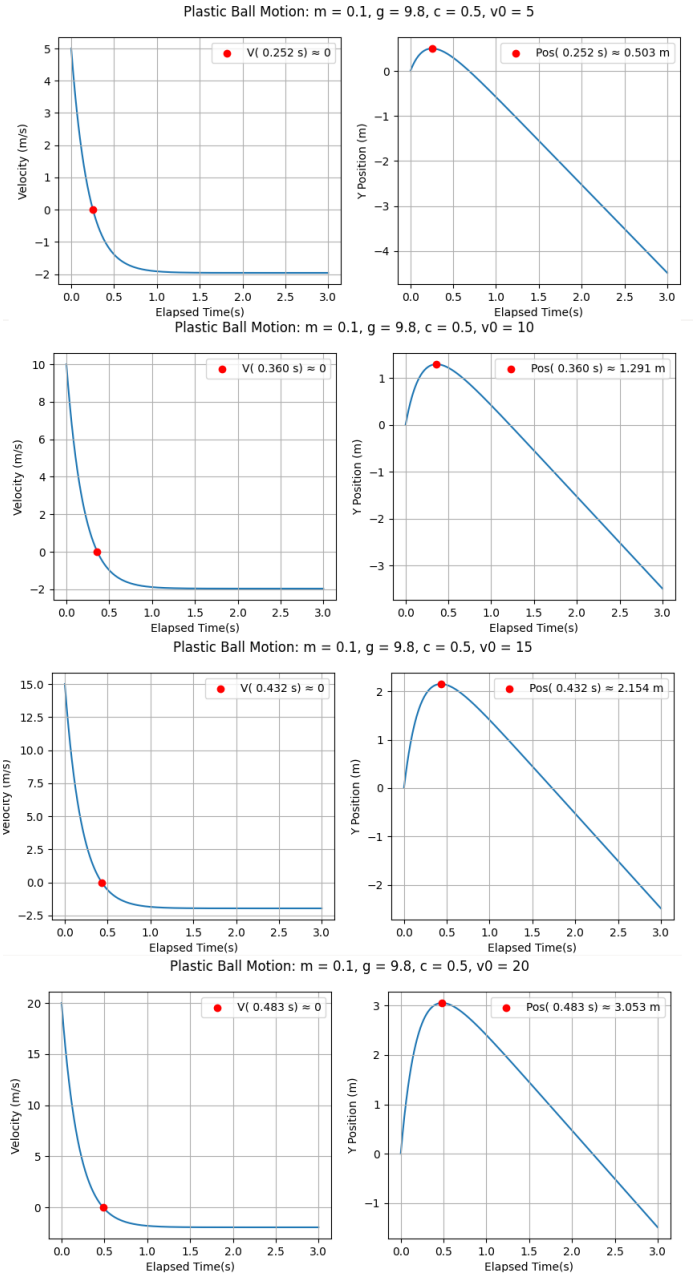

Figura 12: Simulação computacional do lançamento vertical de uma bola de plástico com $\frac{c}{m} = 5s^{-1}$ sujeita a diferentes valores de v_0. Fonte: Autor

Pode-se notar que maiores velocidades inicias fazem com que a bola atinja maior altura máxima. Entretanto, a velocidade terminal e a proporção entre o tempo de subida e descida não parecem sofrer muitas alterações. Isso indica que essas medidas estão fortemente associadas com a razão entre a constante de arrasto e a massa, que foi mantida constante durante as simulações.

5 Predator-Prey Model

O modelo Predador-Presa é um modelo de dinâmica populacional que descreve a interação entre duas espécies. As equações de Lotka-Volterra

$$\dot{x} = x - axy,$$

$$\dot{y} = -y + bxy$$

são uma simplificação razoável para estudos introdutórios do modelo em questão. Nelas, x representa a população de presa e y, de predador. Além disso, a e b representam taxas de interação, ou contato, entre as espécies. A Figura 13, retirada de [2], apresenta um diagrama de blocos para o modelo Predador-Presa, utilizando como exemplo coelhos e raposas.

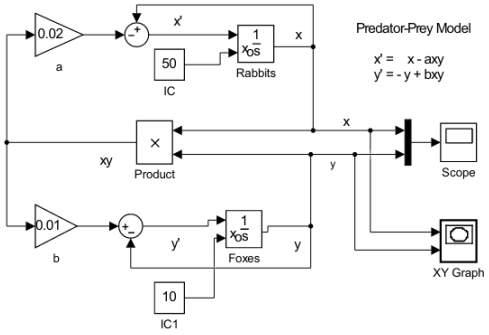

Figura 13: Diagrama de blocos para o modelo Predador-Presa. Fonte: [2]

O código para o modelo Predador-Presa é apresentado no Listing 6

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp

def predator_prey(x0,y0,a,b):
    #x0: inital prey population (e.g Rabbits)
    #y0: initial predator population (e.g Foxes)
    #a: contact coefficient of preys with predators
    #b: contact coefficient of predators with preys

    # array with sampling times
    t_arr = np.linspace(0,50,10000)

    # dy/dt = func(...)
    # y = [x,y]
    def func(t,y,a,b):
        return [y[0] - a*y[0]*y[1],
                -y[1] + b*y[0]*y[1],]

    sol = solve_ivp(func, [0,50],[x0,y0], args=(a,b), t_eval = t_arr)

    fig, axs = plt.subplots(1, 2, figsize=(12,4))
    fig.suptitle(f'Predator-Prey Model: x0 = {x0}, y0 = {y0}, a = {a}, b = {b}')
    axs[0].plot(sol.t, sol.y[0], label = 'Prey')
    axs[0].set(xlabel='Elapsed Time (days)', ylabel='Population')
```

```
26      axs[0].plot(sol.t, sol.y[1], label = 'Predator')
27      axs[0].grid()
28      axs[0].legend()
29
30      axs[1].plot(sol.y[0], sol.y[1])
31      axs[1].set(xlabel='Prey Population', ylabel='Predator Population')
32      axs[1].grid()
33      plt.show()
34
35      if 0 == any(sol.y[0]):
36          print('Prey was Extinguished')
37      if 0 == any(sol.y[1]):
38          print('Predator was Extinguished')
39
40  #=====================================================================
41  #Define inputs
42  x0 = 150
43  y0 = 100
44  a = 0.01
45  b = 0.01
46
47  predator_prey(x0,y0,a,b)
```

Code Listing 6: Código do Modelo Predador-Presa

Primeiramente foram feitos alguns testes com valores típicos, utilizando uma população inicial próxima e experimentando diferentes taxas de contato entre as espécies. O resultado desses testes estão apresentados na Figura 14.

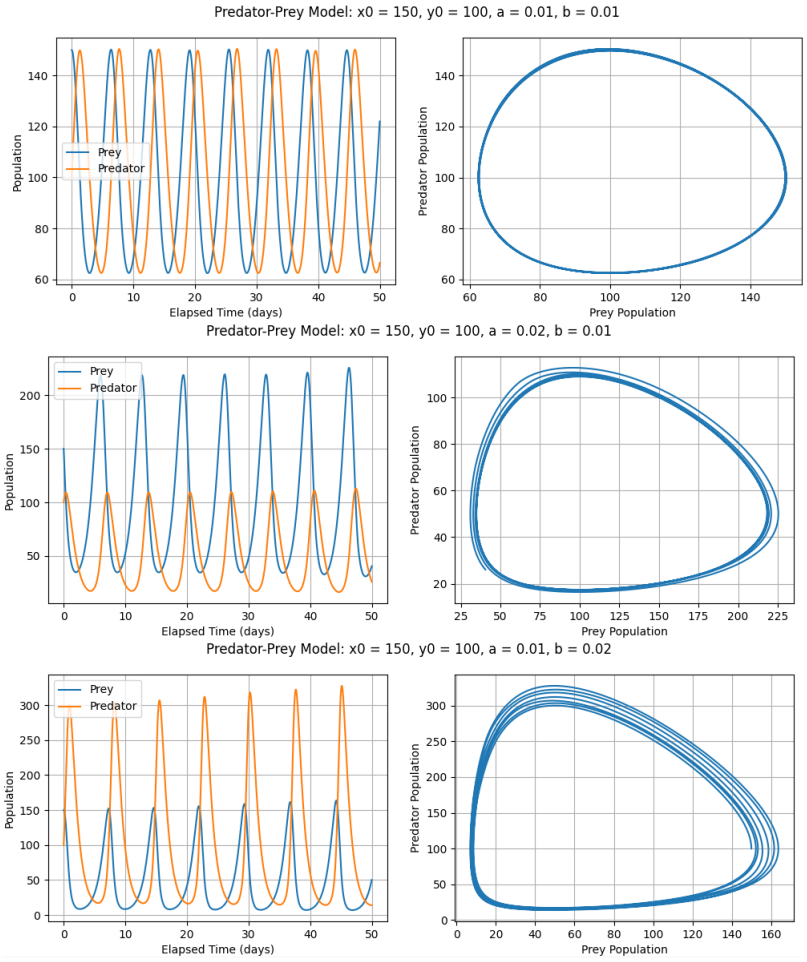

Figura 14: Simulação do modelo Predador-Presa com diferentes taxas de contato. Fonte: Autor

Note que a população das espécies oscila de forma levemente defasada, mas em equilíbrio. Quando as taxas de contato são iguais, a amplitude máxima da oscilação é próxima da população inicial de presa (150) e o centro da oscilação é próximo da população inical de predadores (100). Nos casos em que a ou b são maiores, a amplitude máxima de x e y, respectivamente, seguem a mesma tendência. O gráfico da população de presa pela população de predadores apresenta diversas linhas, majoritariamente sobrepostas, formando um formato ovalado.

O próximo conjunto de testes, abordados pela Figura 15, foi feito com população inicial de presa 5 vezes maior do que a de predador, também experimentando com os valores de a e b.

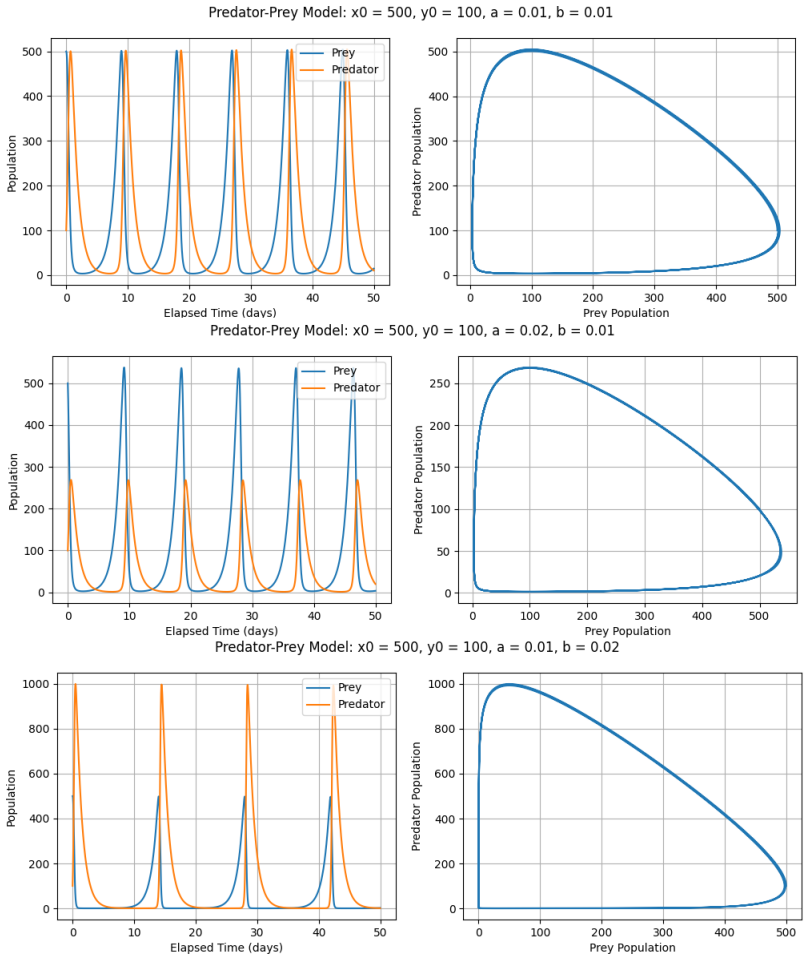

Figura 15: Simulação do modelo Predador-Presa com maior população inicial de Presa. Fonte: Autor

Note que o equilíbrio oscilatório segue ocorrendo, mas com uma defasagem maior. Também é interessante notar que os picos são mais agudos e os vales mais largos. Quando as taxas de contato são iguais, a amplitude máxima da oscilação segue próxima da população inicial de presa (500). Nos casos em que a ou b são maiores, a amplitude máxima de x e y, respectivamente, seguem a mesma tendência. O gráfico da população de presa pela população de predadores apresenta um formato menos ovalado e mais triangular, se aproximando da origem.

Por fim, a Figura 16 apresenta testes com populacão inicial de predadores 5 vezes superior à de presas, também experimentando com os valores de a e b.

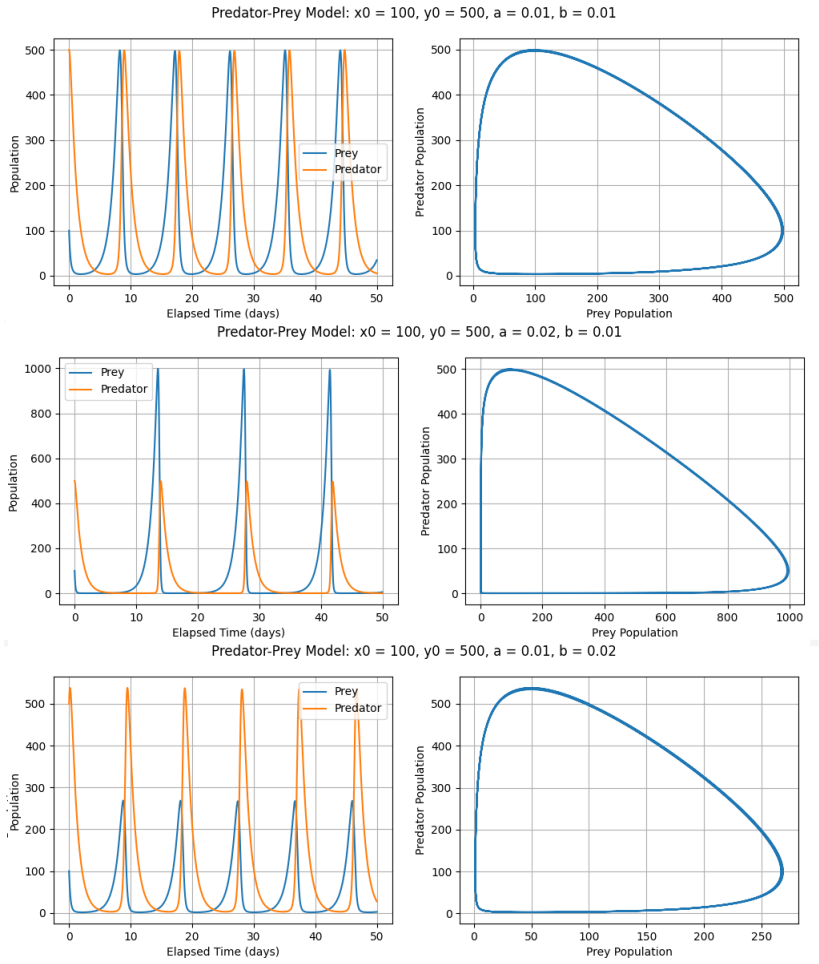

Figura 16: Simulação do modelo Predador-Presa com maior população inicial de Predador. Fonte: Autor

Percebe-se que o equilíbrio oscilatório segue ocorrendo, com os picos agudos e vales largos, assim como no teste com excedente de presa. Uma diferença notável é que, agora, quando as taxas de contato são iguais, a amplitude máxima da oscilação é próxima da população inicial de **predadores** (500). Nos casos em que a ou b são maiores, a amplitude máxima de x e y, respectivamente, seguem a mesma tendência. O gráfico da população de presa pela população de predador tem o mesmo comportamento triangular com pontas arredondadas dos testes com excedente de presa.

Referências

[1] J.R. Dormand e P.J. Prince. "A family of embedded Runge-Kutta formulae". Em: *Journal of Computational and Applied Mathematics* 6.1 (1980), pp. 19 –26. DOI: https://doi.org/10.1016/0771-050X(80)90013-3. URL: http://www.sciencedirect.com/science/article/pii/0771050X80900133.

[2] R. Herman. *Solving Differential Equations Using Simulink*. 2017. URL: http://people.uncw.edu/hermanr/mat361/Simulink/ODE_Simulink.pdf.

[3] *SciPy library*. Disponível em https://www.scipy.org/scipylib/index.html.