Prof. Alexandro Garro Brito
Sistemas de controle - trimestre 2025-3
Programa de Pós-Graduação em Engenharia de Sistemas Eletrônicos / UFSC

Controle de motor CC em espaço de estados

Pedro Henrique Melo Araujo

1. Introdução

Esse trabalho computacional tem o objetivo de modelar e projetar o controle de um motor de corrente contínua (CC) utilizando espaços de estados.

1. Análise do sistema

Modelagem física

A modelagem física do motor conduz ao seguinte sistema de equações

$$\begin{cases} V_t(t) = L_a \frac{di_a(t)}{dt} + R_a i_a(t) + E_a(t) \\ E_a(t) = K_e \omega(t) \\ \omega(t) = \frac{d\theta(t)}{dt} \\ \tau_{ele}(t) - \tau_L(t) = J \frac{d\omega(t)}{dt} + b\omega(t) \\ \tau_{ele} = K_\tau i_a(t) \\ \tau_L(t) = J_L \frac{d\omega(t)}{dt} + b_L \omega(t) \end{cases} \tag{1}$$

Variável	Descrição
$V_t(t)$	tensão terminal (V)
$E_a(t)$	força contra-eletromotriz (V)
$i_a(t)$	corrente de armadura (A)
$\tau_{ele}(t)$	torque da máquina ($N \cdot m$)
$\tau_L(t)$	torque de carga ($N \cdot m$)
$\omega(t)$	velocidade do eixo (rad/s)
$\theta(t)$	posição do eixo (rad)

Parâmetros do motor

Os parâmetros do motor são apresentados na tabela a seguir:

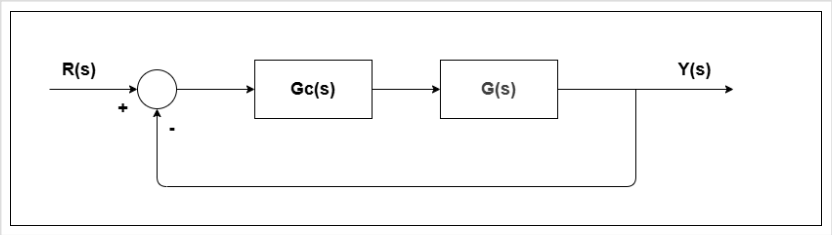
Descrição	Valor
L_a - indut. armadura	1,3 H
R_a - resist. armadura	0,3 Ω
J - mom. inércia MCC	0,0013 $kg \cdot m^2$
b - amort. rotacional MCC	0,00169 $N \cdot m \cdot s/rad$
J_L - mom. inércia de carga	0,036056 $kg \cdot m^2$
K_e - constante construtiva	0,0055678 $V \cdot s/rad$
K_τ - constante construtiva	0,23077 $N \cdot m/A$
b_L - amort. rotacional carga	0,0169 $N \cdot m \cdot s/rad$

```
La = 1.3 # indutância de armadura (H)
Ra = 2.0 # resistência de armadura (ohms)
J = 0.0013 # momento de inércia (kg.m^2)
b = 0.00169 # amortecimento MCC (N.m.s/rad)
Jl = 0.036056 # momento de inércia da carga (kg.m^2)
Ke = 0.0055678 # constante construtiva (N.m/A)
Kt = 0.23077 # constante construtiva (N.m/A)
bl = 0.0169 # amortecimento rotacional da carga (N.m.s/rad)
```

Diagrama de blocos do sistema

```
display_image_with_caption("figs/diagrama_de_blocos.png", 1, "Diagrama de bloc
```

Figura 1 - Diagrama de blocos.



Fonte: Elaborada pelo autor

Resposta ao degrau unitário

a) Função da transferência da tensão do estator para velocidade

A partir das equações usadas para descrever o motor e considerando condições iniciais nulas é possível obter a seguinte função de transferência:

$$G(s) = \frac{\Omega(s)}{V_i(s)} = \frac{K_t}{s^2[L_a(J+J_L)] + s[L_a(b+b_L) + R_a(J+J_L)] + [R_a(b+b_L) \cdot$$

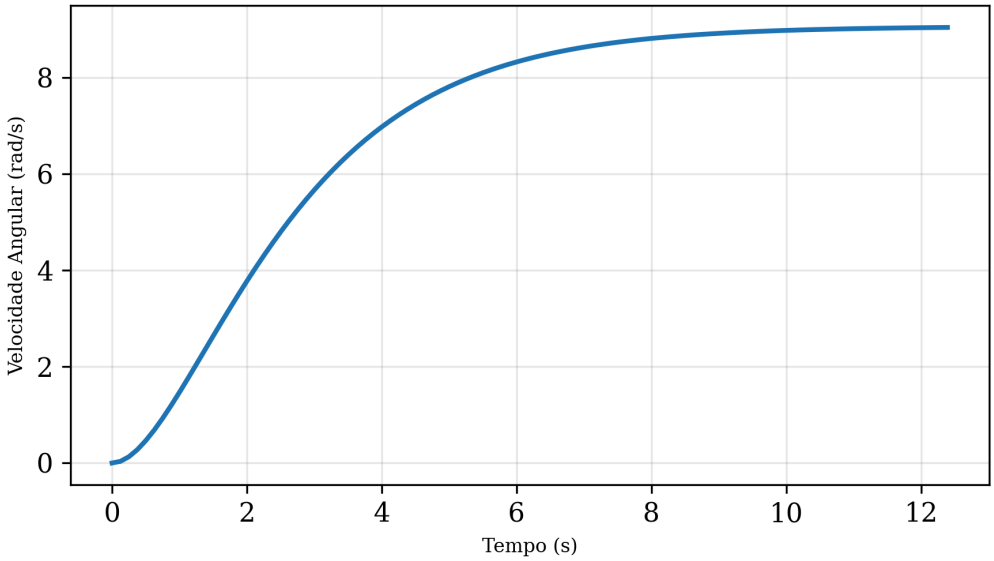
```
coef_num_1 = [Kt]
coef_denom_1 = [La*(J + Jl), La*(b + bl) + La*(J + Jl), La*(b + bl) + Ke*Kt]

print("Coeficientes do numerador: ", coef_num_1)
print("Coeficientes do denominador: ", coef_denom_1)
```

```
Coeficientes do numerador:  [0.23077]
Coeficientes do denominador:  [0.0485628, 0.07272980000000001, 0.02545188120
6]
```

Por fim, a resposta ao degrau unitária desse sistema é apresentada na figura 1

```
G_1 = ct.tf(coef_num_1, coef_denom_1)
# print("Função de transferência G(s):")
# print(G)
t, y =  ct.step_response(G_1)
print_graph(t, y, "Resposta ao degrau unitário da velocidade angular", "Tempo
```

Figura 1 - Resposta ao degrau unitário da velocidade angular



Fonte: Elaborada pelo autor

b) Função do torque de carga para velocidade

A partir das equações usadas para descrever o motor e considerando condições iniciais nulas é possível obter a seguinte função de transferência:

$$G(s) = \frac{\Omega(s)}{\tau_i(s)} = \frac{1}{sJ_a + bl} \tag{3}$$

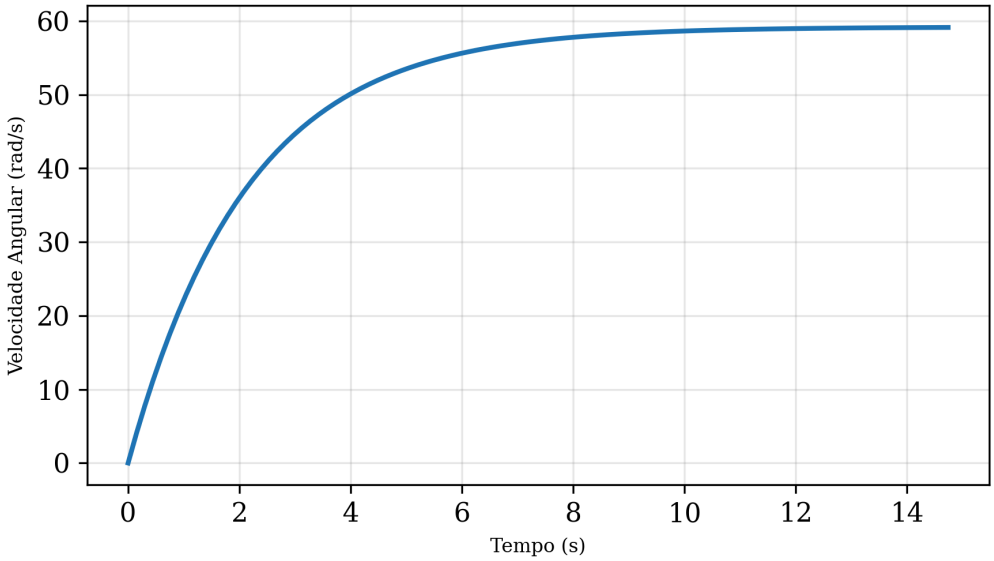
```
coef_num_2 = [1]
coef_denom_2 = [Jl, bl]

print("Coeficientes do numerador: ", coef_num_2)
print("Coeficientes do denominador: ", coef_denom_2)
```

```
Coeficientes do numerador:  [1]
Coeficientes do denominador:  [0.036056, 0.0169]
```

Por fim, a resposta ao degrau unitária desse sistema é apresentada na figura 2

```
G_2 = ct.tf(coef_num_2, coef_denom_2)
# print("Função de transferência G(s):")
# print(G)
t, y = ct.step_response(G_2)
print_graph(t, y, "Resposta ao degrau unitário da velocidade angular", "Tempo
```

Figura 2 - Resposta ao degrau unitário da velocidade angular



Fonte: Elaborada pelo autor

Análise dos resultados

2. Representação em espaços de estados

Inicialmente para representar o sistema em espaços de estados, é necessário definir as variáveis de estado. Analisando as equações que descrevem o motor, fica claro que a corrente de armadura $i_a(t)$ e a velocidade angular $\omega(t)$ estão relacionadas, mas não são interdependentes. Logo, a evolução temporal dessas variáveis descreve bem o comportamento do sistema e elas devem ser escolhidas como variáveis de estado.

O sistema apresenta duas entradas, a tensão do estator $V_t(t)$ e o torque de carga $\tau_L(t)$. A saída do sistema é a velocidade angular $\omega(t)$. Logo, escrevendo as equações do motor em termos das entradas e saídas, temos:

$$\begin{cases} \dfrac{di_a(t)}{dt} = -(R_a/L_a)i_a(t) - (K_e/L_a)\omega(t) + (1/L_a)V_t \\ \dfrac{d\omega(t)}{dt} = (K_T/J)i_a(t) - (b/J)\omega(t) - (1/J)\tau_L(t) \\ y(t) = \omega(t) \end{cases} \tag{4}$$

Colocando essas equações na forma matricial, obtemos a representação em espaços de estados do sistema:

$$\begin{cases} \dot{\vec{x}}(t) = \mathbf{A}\vec{x}(t) + \mathbf{B}\vec{u}(t) \\ \vec{y}(t) = \mathbf{C}\vec{x}(t) + \mathbf{D}\vec{u}(t) \end{cases} \tag{5}$$

$$
\begin{cases}
\begin{bmatrix} \frac{di_a(t)}{dt} \\ \\ \frac{d\omega(t)}{dt} \end{bmatrix} = \begin{bmatrix} -\frac{R_a}{L_a} & -\frac{K_e}{L_a} \\ \\ \frac{K_T}{J} & -\frac{b}{J} \end{bmatrix} \begin{bmatrix} i_a(t) \\ \\ \omega(t) \end{bmatrix} + \begin{bmatrix} \frac{1}{L_a} & 0 \\ \\ 0 & -\frac{1}{J} \end{bmatrix} \begin{bmatrix} V_t(t) \\ \\ \tau_L(t) \end{bmatrix} \\
y(t) = \begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} i_a(t) \\ \\ \omega(t) \end{bmatrix} + \begin{bmatrix} 0 & 0 \end{bmatrix} \begin{bmatrix} V_t(t) \\ \\ \tau_L(t) \end{bmatrix}
\end{cases} \tag{6}
$$

Em seguida, serão analisados os polos e zeros do sistema representado em espaços de estados. Os polos do sistema são os autovalores da matriz A, tomando o caso contínuo podemos calcular os polos como:

$$
det(\nu\mathbf{I} - \mathbf{A}) = 0 \tag{6}
$$

```
# Definir matrizes
A = np.array([
    [-Ra/La, -Ke/La],
    [Kt/J,     -b/J]])
B = np.array([
    [1/La, 0],
    [0, -1/J]])
C = np.array([[0, 1]])
D = np.array([[0, 0]])
# Criar o objeto state space
sys_cont = ct.ss(A, B, C, D)

polos = ct.poles(sys_cont)
print(f"Polos do sistema: {polos}")
```

```
Polos do sistema: [-1.41923077+0.86375272j -1.41923077-0.86375272j]
```

Os zeros podem ser encontrados através da seguinte equação:

$$
\mathbf{C} \cdot \text{adj}(\nu\mathbf{I} - \mathbf{A}) \cdot \mathbf{B} = 0 \tag{7}
$$

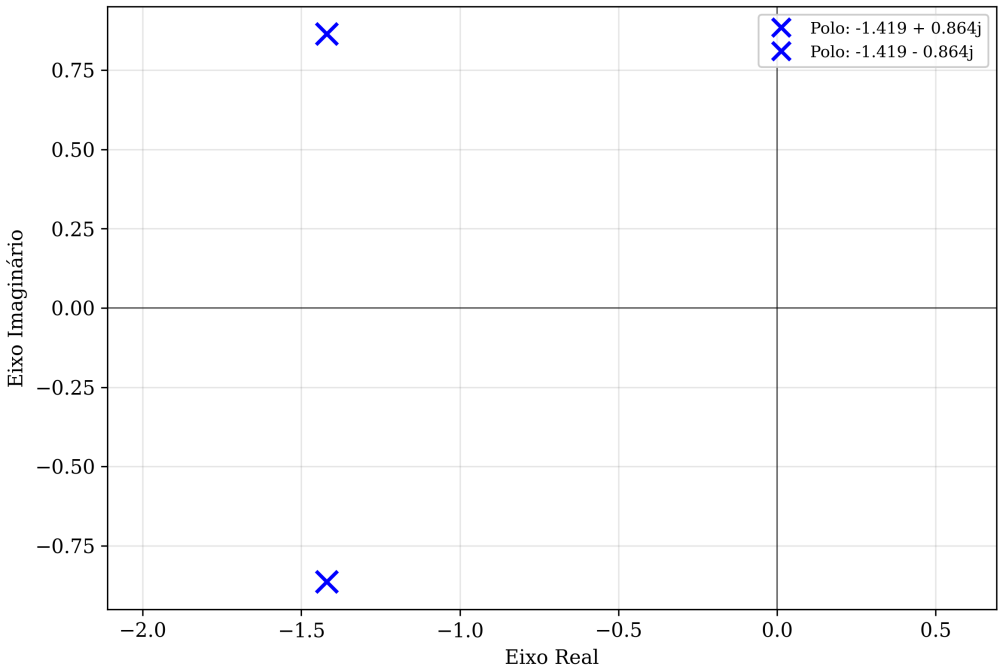
```
zeros = ct.zeros(sys_cont)
print(f"Zeros do sistema: {zeros}")
```

```
Zeros do sistema: []
```

Logo, com esses resultados pode-se plotar o mapa de polos e zeros do sistema.

```
print_pzmap(sys_cont, fig_number=3)
```

Figura 3 - Mapa de Polos e Zeros



Fonte: Elaborado pelo autor

A partir da análise do sistema, obteve-se dos polos em malha aberta, localizados em $s = -1.42 \pm 0,86j$, e nenhum zero. Logo, o sistema é estável, pois a parte real de ambos os polos é negativa, situando-se no semiplano esquerdo. Além disso, como os polos são complexos conjugados, o sistema é classificado como sub-amortecido. Isso implica que sua resposta transitória natural apresentará oscilaÇões antes de atingir o estado estacionário.

3. Discretização do sistema

Utilizando a seguinte a aproximação para a derivada e substituindo na definição de espaço de estado para sistemas contínuos, obtêm-se.

$$\dot{x}(t) \approx \frac{x[k+1] - x[k]}{T_s} \tag{8}$$

$$\begin{cases} \vec{x}[k+1] = \mathbf{A}_d\vec{x}[k] + \mathbf{B}_d\vec{u}[k] \\ \vec{y}[k] = \mathbf{C}_d\vec{x}[k] + \mathbf{D}_d\vec{u}[k] \end{cases} \tag{9}$$

Onde:

- $\mathbf{A}_d = T_s\mathbf{A} + \mathbf{I}$
- $\mathbf{B}_d = T_s\mathbf{B}$
- $\mathbf{C}_d = \mathbf{C}$

- $\mathbf{D}_d = \mathbf{D}$

Logo, o sistema em análise pode ser discretizado como:

$$\begin{cases} \begin{bmatrix} i_a[k+1] \\ \\ \omega[k+1] \end{bmatrix} = \begin{bmatrix} 1 - \frac{T_s R_a}{L_a} & -\frac{T_s K_e}{L_a} \\ \\ \frac{T_s K_T}{J} & 1 - \frac{T_s b}{J} \end{bmatrix} \begin{bmatrix} i_a[k] \\ \\ \omega[k] \end{bmatrix} + \begin{bmatrix} \frac{T_s}{L_a} & 0 \\ \\ 0 & -\frac{T_s}{J} \end{bmatrix} \begin{bmatrix} V_t[k] \\ \\ \tau_L[k] \end{bmatrix} \\ \\ y[k] = \begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} i_a[k] \\ \\ \omega[k] \end{bmatrix} + \begin{bmatrix} 0 & 0 \end{bmatrix} \begin{bmatrix} V_t[k] \\ \\ \tau_L[k] \end{bmatrix} \end{cases}$$

Adotando um tempo de amostragem de $T_s = 0.005$, obtém-se:

```
# Sampling time
Ts = 0.005

sys_dis = sys_cont.sample(Ts, method="euler")

print("Matriz Ad discreta:")
print(sys_dis.A)

print("\nMatriz Bd discreta:")
print(sys_dis.B)

print("\nMatriz Cd discreta:")
print(sys_dis.C)

print("\nMatriz Dd discreta:")
print(sys_dis.D)
```

```
Matriz Ad discreta:
[[ 9.92307692e-01 -2.14146154e-05]
 [ 8.87576923e-01  9.93500000e-01]]

Matriz Bd discreta:
[[ 3.84615385e-03  0.00000000e+00]
 [ 0.00000000e+00 -3.84615385e+00]]

Matriz Cd discreta:
[[0. 1.]]

Matriz Dd discreta:
[[0. 0.]]
```

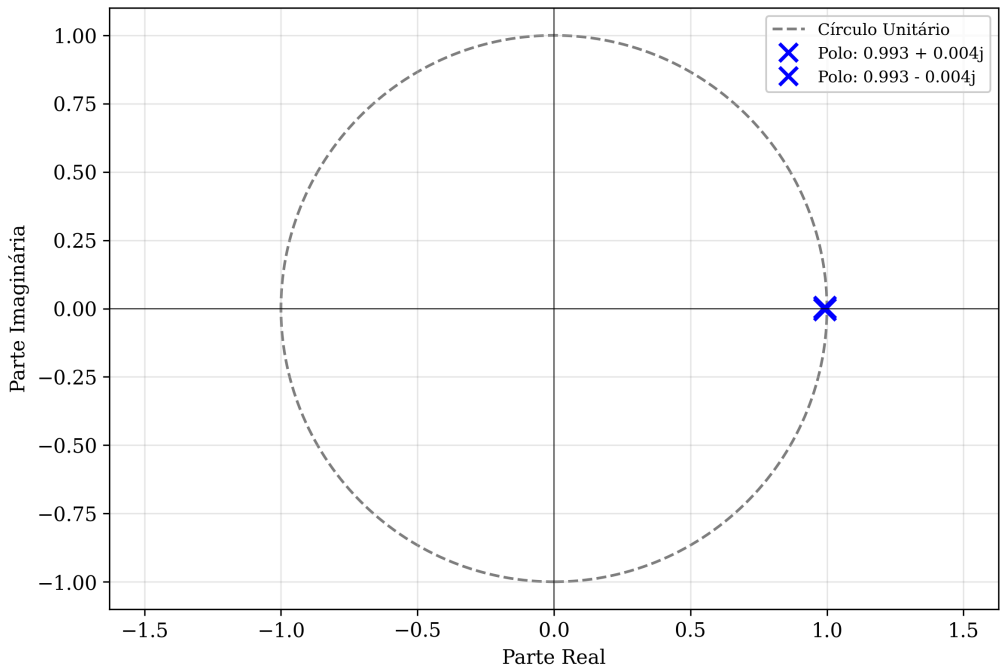
$$\mathbf{x}[k+1] = \begin{bmatrix} 9.9231 \times 10^{-1} & -2.1415 \times 10^{-5} \\ 8.8758 \times 10^{-1} & 9.9350 \times 10^{-1} \end{bmatrix} \mathbf{x}[k] + \begin{bmatrix} 3.8462 \times 10^{-3} & (\\ 0 & -3.8 \end{bmatrix}$$

$$\mathbf{y}[k] = \begin{bmatrix} 0 & 1 \end{bmatrix} \mathbf{x}[k]$$

Analisando a estabilidade do sistema obtido tem-se

```
print_pzmap(sys_dis)
```

Figura 3 - Mapa de Polos e Zeros (Sistema Discreto)

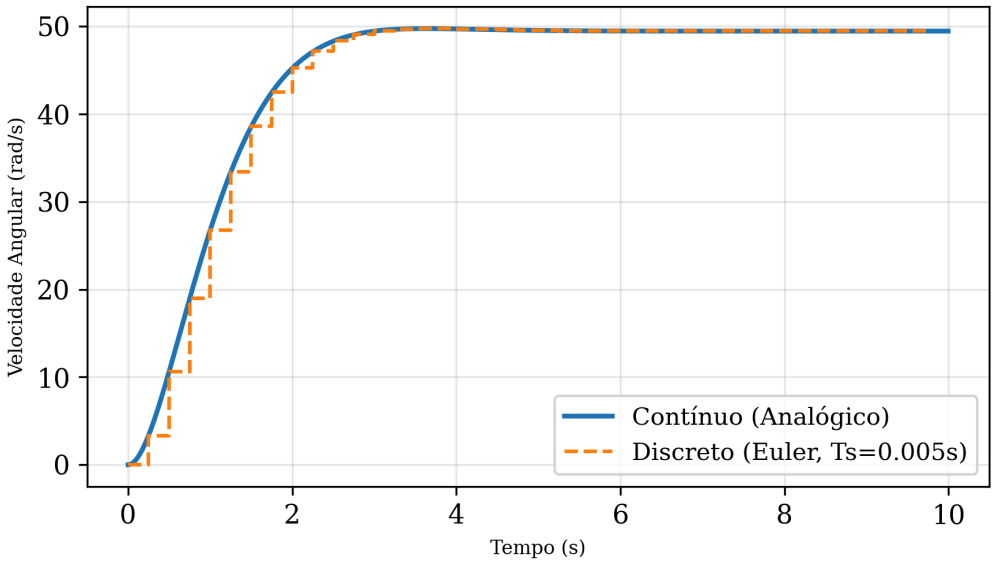


Fonte: Elaborado pelo autor

Logo, a partir da figura 3 pode se observar que o sistema discreto é estável, uma vez que os autovalores da matriz A_d estão dentro do círculo unitário. Comparando ambas as respostas dos sistemas ao degrau unitário de tensão do estator:

```
t_final = 10.0
t = np.arange(0, t_final, Ts)

# Construir entrada em degrau
U = np.zeros((2, len(t)))
U[0, :] = 1.0   # Degrau em Vt

t_c, y_c = ct.forced_response(sys_cont, T=t, U=U)
t_d, y_d = ct.forced_response(sys_dis, T=t, U=U)

print_graph(t_c, y_c[0],
            "Comparação: Resposta ao degrau unitário de tensão",
            "Tempo (s)",
            "Velocidade Angular (rad/s)",
            fig_number=4,
            t2=t_d,
            y2=y_d[0],
            label1="Contínuo (Analógico)",
            label2=f"Discreto (Euler, Ts={Ts}s)")
```

Figura 4 - Comparação: Resposta ao degrau unitário de tensão



Fonte: Elaborada pelo autor

Analisando a figura 4 que mostra a comparação das respostas ao degrau unitário, pode-se concluir que a discretização manteve as características dinâmicas do sistema contínuo. Além disso, a escolha da taxa de amostragem de $T_s = 1ms$ mostrou-se adequada ao manter a fidelidade dinâmica entre os sistemas e não introduziu instabilidades.

4. Análise da controlabilidade e observabilidade

A partir do sistema discreto obtido na seção anterior e apresentado abaixo a seguir, omitindo a matriz D_d uma vez que ela é nula:

a. Controlabilidade

Para analisar a controlabilidade do sistema, ou seja, se ele pode levar os estados, i_a e w, de um ponto inicial para um ponto final qualquer, a matriz de controlabilidade $\mathcal{C}$ deve ter posto cheio. A matriz de controlabilidade pode ser obtida a partir da fórmula a seguir.

$$\mathcal{C} = \begin{bmatrix} \mathbf{B}_d & \mathbf{A}_d\mathbf{B}_d & \dots & \mathbf{A}_d^{n-1}\mathbf{B}_d \end{bmatrix} \tag{12}$$

Uma vez que o sistema em questão possui duas entradas na matriz $\mathbf{B}$: V_t relacionada a entrada de controle ou do atuador e τ_L relacionada ao distúrbio ou pertubação. Como não comandos a carga devemos analisar a controlabilidade apenas do ponto

de vista da tensão que pode ser comandada. Logo, verificando a controlabilidade usando a matriz $\mathbf{B}$ apenas cm a coluna relacionada a tensão terminal V_t, obtemos:

```
Ad = sys_dis.A
Bd = sys_dis.B

# Bd tem shape (2,2). A coluna 0 é Vt e a coluna 1 é Tau_l
# Precisamos fazer um reshape para garantir que continue sendo uma matriz colu
Bd_atuador = Bd[:, 0].reshape(-1, 1)

# Matriz de controlabilidade
C = ct.ctrb(Ad, Bd_atuador)

# Verificar o posto da matriz
posto = np.linalg.matrix_rank(C)
num_states = Ad.shape[0]

print(f"Matriz de Controlabilidade (Atuador):\n{C}")
print("-" * 30)
print(f"Número de Estados (n): {num_states}")
print(f"Posto da Matriz: {posto}")
```

```
Matriz de Controlabilidade (Atuador):
[[0.00384615 0.00381657]
 [0.         0.00341376]]
------------------------------
Número de Estados (n): 2
Posto da Matriz: 2
```

$$\mathcal{C} = \begin{bmatrix} 3.8462 \times 10^{-3} & 3.8166 \times 10^{-3} \\ 0 & 3.4138 \times 10^{-3} \end{bmatrix} \tag{13}$$

$$\begin{cases} n = 2 \\ \text{posto}(\mathcal{C}) = 2 \end{cases} \therefore \text{Sistema Controlável}$$

Visto que pelos resultados a matriz tem posto cheio, ou seja, o número de colunas linearmente independentes é igual a dimensão da matriz, o sistema é controlável através do comando do atuador através da tensão terminal.

b. Observabilidade

A observabilidade do sistema é análoga a controlabilidade, porém analisando a saída do sistema. Para o sistema ser observável a partir da saída $y(t)$ e entrada $u(t)$ durante uma intervalo de tempo, deve-se ser capaz de estimar os estados dados pela corrente i_a e pela velocidade w. Para determinar a observabilidade devemos analisar a matriz de observabilidade que pode ser obtido como:

$$\mathcal{O} = \begin{bmatrix} \mathbf{C}_d \\ \mathbf{C}_d\mathbf{A}_d \\ \vdots \\ \mathbf{C}_d\mathbf{A}_d^{n-1} \end{bmatrix} \tag{14}$$

Utilizando os valores das matrizes $\mathbf{C}_d$ e $\mathbf{A}_d$ obtidas anteriormente.

```
Ad = sys_dis.A
Cd = sys_dis.C

# Calcular matriz de observabilidade
O = ct.obsv(Ad, Cd)

# Verificar o posto
posto_obs = np.linalg.matrix_rank(O)
num_states = Ad.shape[0]

print(f"Matriz de Saída Cd: {Cd}")
print(f"Matriz de Observabilidade:\n{O}")
print("-" * 30)
print(f"Posto da Matriz: {posto_obs}")
```

```
Matriz de Saída Cd: [[0. 1.]]
Matriz de Observabilidade:
[[0.         1.        ]
 [0.88757692 0.9935    ]]
------------------------------
Posto da Matriz: 2
```

$$\mathcal{O} = \begin{bmatrix} 0 & 1 \\ 8.8758 \times 10^{-1} & 9.9350 \times 10^{-1} \end{bmatrix} \tag{15}$$

$$\begin{cases} n = 2 \\ \text{posto}(\mathcal{O}) = 2 \end{cases} \therefore \text{Sistema Observável}$$

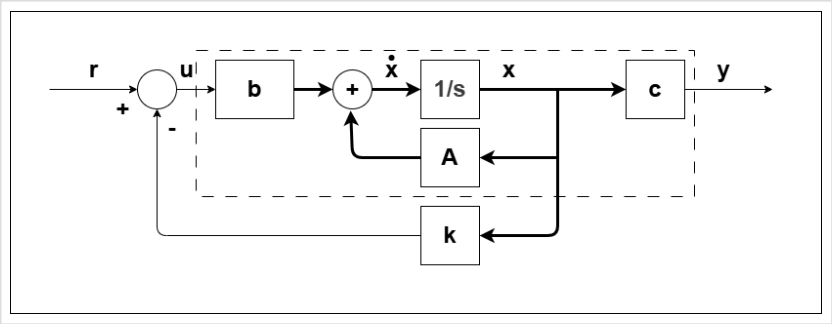
Logo, de forma análoga para o caso da controlabilidade a matriz de observabilidade tem posto cheio, e portanto, o sistema é observável. Dessa forma, as variáveis de estado podem ser estimadas tomando a entrada e a saída do sistema em um intervalo de tempo

5. Controlador por alimentação estática de estados

A partir do do sistema discretizado (equação 11) propõe-se o seguinte controlador por realimentação estática de estados.

```
display_image_with_caption("figs/diagrama_de_blocos_se.png", 5, "Diagrama de b
```

Figura 5 - Diagrama de blocos do controlador por realimentação de estados com integrador.



Fonte: Elaborada pelo autor

Como discutido na seção anterior através do resultado da equação 13, o sistema em malha aberta é controlável. Portanto, segundo o teorema da controlabilidade o sistema em malha fechada é controlável.

Para cálculo do controlador vamos usar apenas a coluna da matriz $\mathbf{B}$ relacionada a tensão, uma vez que é o parâmetro que pode-se atuar. O torque de carga nesse cenário é um distúrbio, e portanto, não é passível de ser controlado.

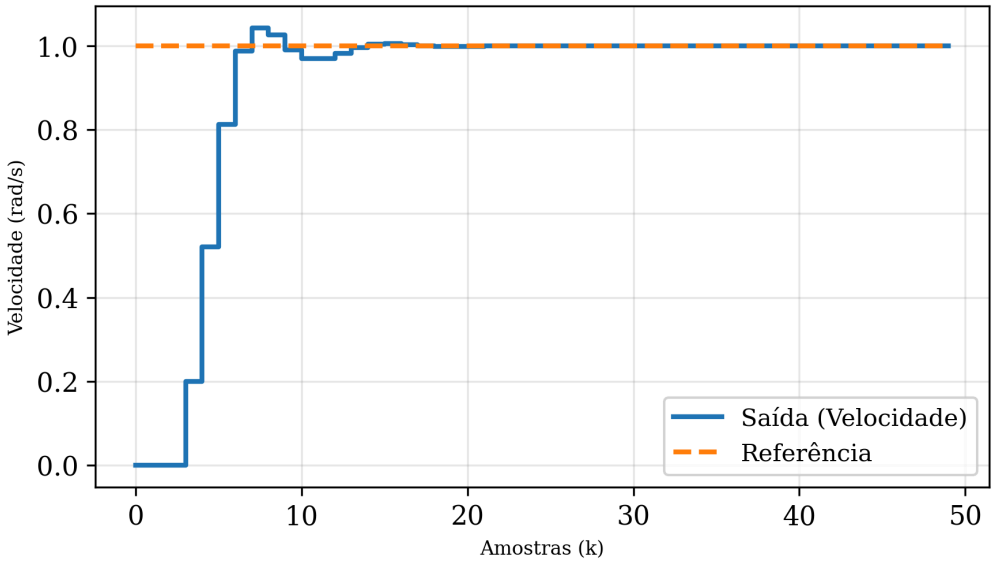
Visto que os polos de malha aberta são: $polos_{ma} = 0,994 \pm 0.004j$. Escolhermos os polos de malha fechada para serem um pouco mais rápidos evitando saturar o controlador com uma resposta muito rápida. Dessa forma, os polos escolhidos vão ser $polos_{mf} = 0,5 \pm 0,5j$. Logo, utilizando a implementação relativa a fórmula de Ackermann, obtém-se:

```
Cd = sys_dis.C
Dd = sys_dis.D

# --- 2. Construção das Matrizes Aumentadas ---
# Equação do integrador: xi[k+1] = xi[k] - C*x[k] + r[k]
# A_aum = [ A   0 ]
#         [ -C  1 ]
zeros_col = np.zeros((2, 1))
linha_integrador = np.hstack((-Cd, [[1]])) # [0, -1, 1]
A_aum = np.vstack((np.hstack((Ad, zeros_col)), linha_integrador))
Bd_controle = Bd[:, 0].reshape(-1, 1)
B_aum = np.vstack((Bd_controle, [[0]]))

# --- 3. Cálculo do Ganho K Aumentado ---
# Precisamos de 3 polos agora. Vamos manter os originais e adicionar um integr
# O polo do integrador deve ser um pouco mais rápido ou próximo dos dominantes
polos_desejados = np.array([0.5 + 0.5j, 0.5 - 0.5j, 0.6])

K_aum = ct.place(A_aum, B_aum, polos_desejados)
```

```python
print(f"Ganhos K (Estados): {K_aum[0, :2]}")
print(f"Ganho Ki (Integrador): {K_aum[0, 2]}")

B_ref = np.array([[0], [0], [1]])

A_fechada = A_aum - B_aum @ K_aum

C_aum = np.array([[0, 1, 0]])

sys_ci = ct.StateSpace(A_fechada, B_ref, C_aum, 0, dt=1)

# --- 5. Simulação com forced_response ---
t = np.arange(0, 50, 1)
U = np.ones_like(t) # Degrau de referência 1 rad/s

t_out, y_out = ct.forced_response(sys_ci, T=t, U=U)

# --- 6. Plotagem ---
print_graph(
    t=t_out,
    y=y_out.flatten(),
    title="Controle com Ação Integral (Erro Zero)",
    xlabel="Amostras (k)",
    ylabel="Velocidade (rad/s)",
    fig_number=6,
    t2=t_out,
    y2=U,
    label1="Saída (Velocidade)",
    label2="Referência",
    use_step=True  # Novo parâmetro para usar step plot em ambas as curvas
)
```

```
Ganhos K (Estados): [360.31       260.98024535]
Ganho Ki (Integrador): -58.58647137842874
```

Figura 6 - Controle com Ação Integral (Erro Zero)



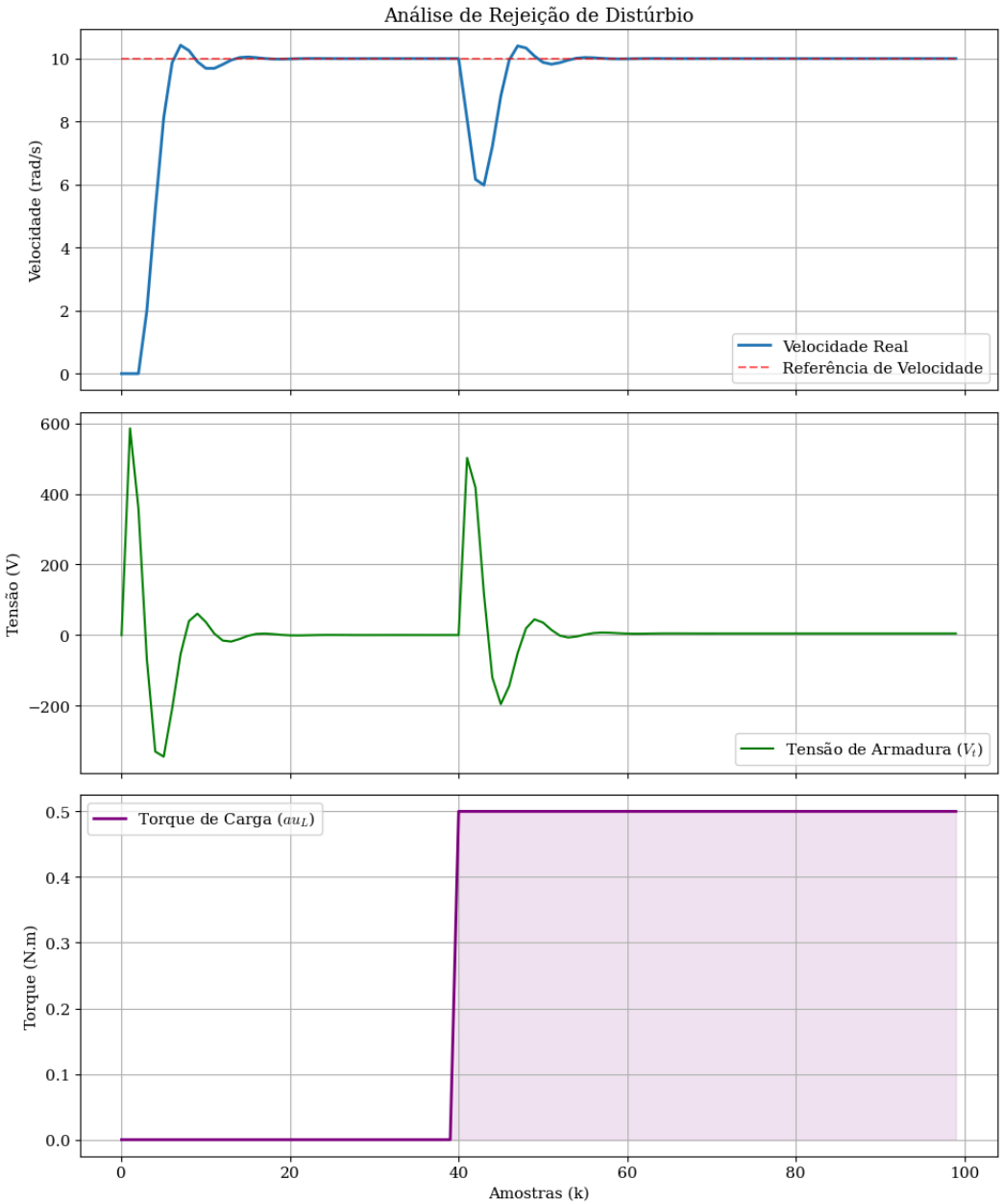
Fonte: Elaborada pelo autor

Resposta do sistema a um distúrbio de tensão de carga

```
Bd_dist = Bd[:, 1].reshape(-1, 1)   # B de distúrbio
K_x = K_aum[:, :2]  # Ganhos para os estados da planta
K_i = K_aum[:, 2:]  # Ganho do integrador

# --- 3. Montagem da Malha Fechada com DISTÚRBIO ---
# Estados: [x_planta(2), x_integrador(1)]
# Entradas da simulação: [Referencia(1), Distúrbio_Torque(1)]

# Matriz A_fechada (3x3)
# Dinâmica da planta: A - B_ctrl*K_x
# Efeito do integrador na planta: -B_ctrl*K_i
top_rows = np.hstack((Ad - Bd_controle @ K_x, -Bd_controle @ K_i))
bot_row = np.hstack((-Cd, [[1]])) # Dinâmica do integrador
A_cl = np.vstack((top_rows, bot_row))

# Matriz B_fechada (3x2) -> DUAS ENTRADAS
# Coluna 1: Referência (Afeta só o integrador)
# Coluna 2: Distúrbio (Afeta a planta via Bd_dist)
b_integrador_ref = np.array([[1]]) # O '1' que multiplica a referência
b_integrador_dist = np.array([[0]]) # O '0' para o distúrbio no integrador

B_cl = np.block([
    [np.zeros((2,1)),  Bd_dist],           # Linha de cima (Planta): 2x2
    [b_integrador_ref, b_integrador_dist]  # Linha de baixo (Integrador): 1x2
])

# Matriz C (Para ver Velocidade e Tensão Aplicada)
# Queremos ver: Velocidade Real e Tensão de Controle (reconstruída)
# Como state space só mostra estados, vamos ver a Velocidade primeiro.
```

```python
C_cl = np.array([[0, 1, 0]]) # Velocidade
D_cl = np.zeros((1, 2))

sys_disturbio = ct.StateSpace(A_cl, B_cl, C_cl, D_cl, dt=1)

# --- 4. Simulação do Cenário ---
t = np.arange(0, 100, 1)
inputs = np.zeros((2, len(t)))

# Cenário:
# k=0 a k=100: Referência de velocidade = 10 rad/s
inputs[0, :] = 10
# k=40 em diante: Aplica torque de carga de 0.5 N.m (Degrau de Distúrbio)
inputs[1, 40:] = 0.5

t, y = ct.forced_response(sys_disturbio, T=t, U=inputs)

# Reconstrução manual da Tensão de Controle para plotagem
# u[k] = -Kx * x - Ki * xi
# Precisamos dos estados x (ia, w) e xi
# O forced_response retorna os estados em "x" (não confundir com o eixo x)
states = ct.forced_response(sys_disturbio, T=t, U=inputs).states
x_planta = states[:2, :]
x_int = states[2, :]

# Calculando u para cada ponto
u_control = - (K_x @ x_planta) - (K_i * x_int)
u_control = u_control.flatten()

# --- 5. Plotagem ---
# Vamos criar 3 subplots compartilhando o eixo X (tempo/amostras)
fig, (ax1, ax2, ax3) = plt.subplots(3, 1, figsize=(10, 12), sharex=True)

# --- Gráfico 1: Velocidade (Saída) ---
# Usamos flatten() para garantir que y não tenha dimensões extras
ax1.plot(t, y.flatten(), label='Velocidade Real', linewidth=2)
ax1.plot(t, inputs[0, :], 'r--', alpha=0.6, label='Referência de Velocidade')
ax1.set_ylabel('Velocidade (rad/s)')
ax1.set_title('Análise de Rejeição de Distúrbio')
ax1.grid(True)
ax1.legend(loc='lower right')

# --- Gráfico 2: Tensão (Esforço de Controle) ---
ax2.plot(t, u_control, color='green', label='Tensão de Armadura ($V_t$)')
ax2.set_ylabel('Tensão (V)')
ax2.grid(True)
ax2.legend(loc='lower right')

# --- Gráfico 3: Torque de Carga (O Distúrbio) ---
# AQUI está o que você pediu: inputs[1, :] é o vetor do torque
ax3.plot(t, inputs[1, :], color='purple', linewidth=2, label='Torque de Carga

ax3.set_ylabel('Torque (N.m)')
ax3.set_xlabel('Amostras (k)')
```

```python
# Vamos pintar a área sob o torque para destacar quando ele está ativo
ax3.fill_between(t, inputs[1, :], color='purple', alpha=0.1)

ax3.grid(True)
ax3.legend(loc='upper left')

plt.tight_layout()
plt.show()
```

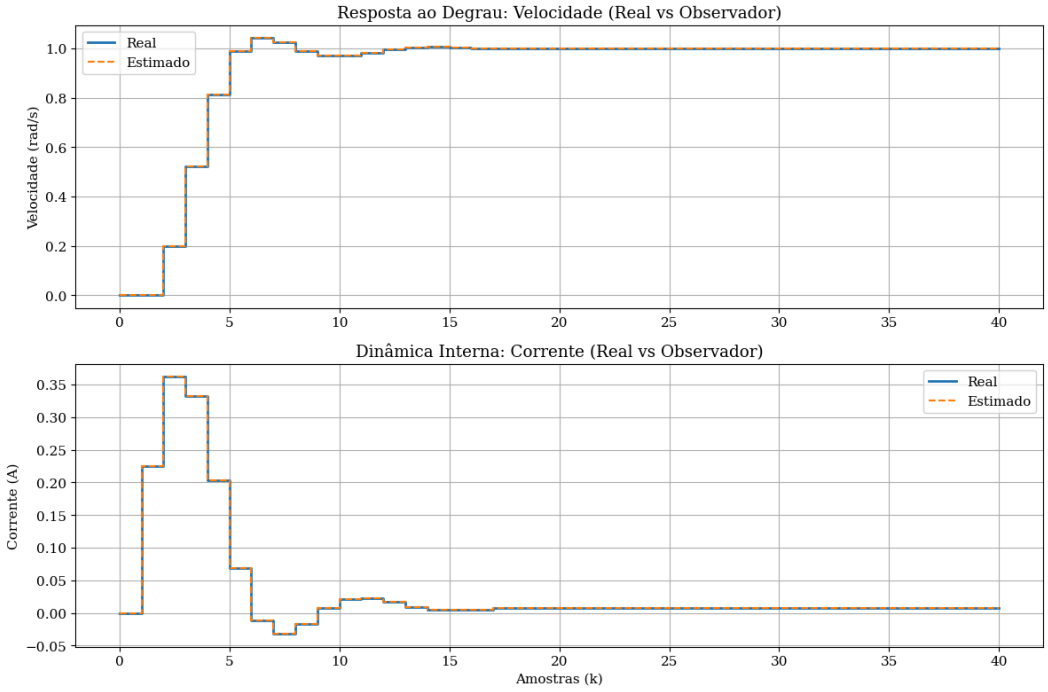


6. Observador de estados

```python
polos_observador = np.array([0.2 + 0.2j, 0.2 - 0.2j])
K_x = K_aum[:, :2]  # Ganhos para os estados da planta
K_i = K_aum[:, 2:]  # Ganho do integrador

print(f"Ganho k_x{K_x}")
print(f"Ganho k_i{K_i}")

# Cálculo do ganho L
L = ct.place(Ad.T, Cd.T, polos_observador).T

print(f"Matriz de ganho do observador L: \n{L}")

# --- Construindo a Matriz A Grande (5x5) ---
# Linha 1: Planta
row1 = np.hstack((Ad, -Bd_controle @ K_i, -Bd_controle @ K_x))
# Linha 2: Integrador
row2 = np.hstack((-Cd, [[1]], np.zeros((1, 2))))
# Linha 3: Observador
row3 = np.hstack((L @ Cd, -Bd_controle @ K_i, Ad - Bd_controle @ K_x - L @ Cd)

A_big = np.vstack((row1, row2, row3))

# --- Construindo a Matriz B Grande (5x1) ---
# A entrada é apenas a Referência (r)
# Planta: 0, Integrador: 1, Observador: 0
B_big = np.array([[0], [0], [1], [0], [0]])

# --- Construindo a Matriz C Grande ---
# Queremos ver: [Velocidade Real, Velocidade Estimada, Corrente Real, Corrente
# x = [ia, w, xi, ia_est, w_est]
C_big = np.array([
    [0, 1, 0, 0, 0], # y1: Velocidade Real
    [0, 0, 0, 0, 1], # y2: Velocidade Estimada
    [1, 0, 0, 0, 0], # y3: Corrente Real
    [0, 0, 0, 1, 0]  # y4: Corrente Estimada
])

D_big = np.zeros((4, 1))

# --- 4. Criando o Sistema e Simulando ---
sys_completo = ct.StateSpace(A_big, B_big, C_big, D_big, dt=1)

# Simulação de Degrau
t, y = ct.step_response(sys_completo, T=40)

y = np.squeeze(y)

# --- 5. Plotagem ---
plt.figure(figsize=(12, 8))

# Subplot Velocidade
plt.subplot(2, 1, 1)
plt.step(t, y[0], label='Real', linewidth=2)      # Linha 0 do C_big
plt.step(t, y[1], '--', label='Estimado')         # Linha 1 do C_big
```

```
plt.title('Resposta ao Degrau: Velocidade (Real vs Observador)')
plt.ylabel('Velocidade (rad/s)')
plt.legend()
plt.grid(True)

# Subplot Corrente
plt.subplot(2, 1, 2)
plt.step(t, y[2], label='Real', linewidth=2)      # Linha 2 do C_big
plt.step(t, y[3], '--', label='Estimado')         # Linha 3 do C_big
plt.title('Dinâmica Interna: Corrente (Real vs Observador)')
plt.ylabel('Corrente (A)')
plt.xlabel('Amostras (k)')
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()
```

```
Ganho k_x[[360.31       260.98024535]]
Ganho k_i[[-58.58647138]]
Matriz de ganho do observador L:
[[0.75230941]
 [1.58580769]]
```



7. Dependências necessárias e funções auxiliares

```
# dependências necessárias
import control as ct
import matplotlib.pyplot as plt
import numpy as np
from IPython.display import Image, HTML, display
import io, base64
```

```
import warnings

warnings.filterwarnings('ignore', category=FutureWarning)

# Configuração global de fontes do matplotlib
plt.rcParams['font.family'] = 'serif'
plt.rcParams['font.serif'] = ['DejaVu Serif', 'Times New Roman', 'Computer Mod
plt.rcParams['mathtext.fontset'] = 'cm'  # Computer Modern para fórmulas matem
plt.rcParams['font.size'] = 11

# funções auxiliares
def print_graph(t, y, title, xlabel, ylabel, fig_number=1, source="Elaborada p
    plt.figure(figsize=(6, 4))

    # Plotar primeira curva
    if use_step:
        plt.step(t, y, where='post', linewidth=2, label=label1)
    else:
        plt.plot(t, y, linewidth=2, label=label1)

    # Se houver segunda curva, plotá-la
    if t2 is not None and y2 is not None:
        if use_step:
            plt.step(t2, y2, where='post', linestyle='--', linewidth=2, label=
        else:
            step_plot = 50  # Plotar a cada 50 pontos para visualização mais c
            plt.step(t2[::step_plot], y2[::step_plot], label=label2, where='po
                    linestyle='--')

    plt.title(f"Figura {fig_number} - {title}", fontsize=9, fontweight='bold',
    plt.xlabel(xlabel, fontsize=8)
    plt.ylabel(ylabel, fontsize=8)
    plt.grid(True, alpha=0.3)

    # Adicionar legenda se houver duas curvas
    if t2 is not None and y2 is not None:
        plt.legend(fontsize=10)

    fig = plt.gcf()
    fig.text(0.5, 0.01, f"Fonte: {source}", fontsize=9, style='italic', ha='ce
    plt.tight_layout(rect=[0, 0.03, 1, 1])
    buf = io.BytesIO()
    fig.savefig(buf, format='png', dpi=300, bbox_inches='tight')
    buf.seek(0)
    img_b64 = base64.b64encode(buf.read()).decode('utf-8')
    buf.close()
    plt.close(fig)
    display(HTML(f'<div style="text-align:center"><img src="data:image/png;bas

def print_pzmap(system, fig_number=3, source="Elaborado pelo autor"):
    fig = plt.figure(figsize=(8, 6))
    poles = ct.poles(system)
    zeros = ct.zeros(system)
    is_discrete = system.dt is not None and system.dt > 0
```

```
    # Lista para armazenar as labels dos polos e zeros
    legend_labels = []

    if is_discrete:
        circle = plt.Circle((0, 0), 1, color='gray', fill=False,
                            linestyle='--', linewidth=1.5)
        plt.gca().add_patch(circle)
        legend_labels.append(('Círculo Unitário', None))

    # Plotar polos sem anotações individuais
    for i, p in enumerate(poles):
        plt.plot(np.real(p), np.imag(p), 'x', markersize=12, color='blue',
                 markeredgewidth=2)

        # Adicionar à lista de legendas
        if np.imag(p) >= 0:
            pole_label = f'{np.real(p):.3f} + {np.imag(p):.3f}j' if np.imag(p)
        else:
            pole_label = f'{np.real(p):.3f} - {abs(np.imag(p)):.3f}j'
        legend_labels.append((f'Polo: {pole_label}', 'blue'))

    # Plotar zeros sem anotações individuais
    if len(zeros) > 0:
        for i, z in enumerate(zeros):
            plt.plot(np.real(z), np.imag(z), 'o', markersize=10, color='red',
                     markerfacecolor='none', markeredgewidth=2)

            # Adicionar à lista de legendas
            if np.imag(z) >= 0:
                zero_label = f'{np.real(z):.3f} + {np.imag(z):.3f}j' if np.ima
            else:
                zero_label = f'{np.real(z):.3f} - {abs(np.imag(z)):.3f}j'
            legend_labels.append((f'Zero: {zero_label}', 'red'))

    plt.grid(True, alpha=0.3)

    if is_discrete:
        plt.xlabel('Parte Real', fontsize=11)
        plt.ylabel('Parte Imaginária', fontsize=11)
        plt.title(f'Figura {fig_number} - Mapa de Polos e Zeros (Sistema Discr
                 fontsize=10, fontweight='bold', pad=20)
    else:
        plt.xlabel('Eixo Real', fontsize=11)
        plt.ylabel('Eixo Imaginário', fontsize=11)
        plt.title(f'Figura {fig_number} - Mapa de Polos e Zeros',
                 fontsize=9, fontweight='bold', pad=20)

    plt.axhline(0, color='black', lw=0.5)
    plt.axvline(0, color='black', lw=0.5)
    plt.axis('equal')

    # Criar elementos de legenda customizados
    from matplotlib.lines import Line2D
```

```
    custom_lines = []
    custom_labels = []

    for label, color in legend_labels:
        if color == 'blue':
            custom_lines.append(Line2D([0], [0], marker='x', color='w', marker
                                       markeredgecolor='blue', markersize=10, m
        elif color == 'red':
            custom_lines.append(Line2D([0], [0], marker='o', color='w', marker
                                       markeredgecolor='red', markersize=10, ma
        else:
            custom_lines.append(Line2D([0], [0], color='gray', linestyle='--',
        custom_labels.append(label)

    # Posicionar legenda dentro do gráfico
    plt.legend(custom_lines, custom_labels, loc='upper right', fontsize=9, fra

    fig.text(0.5, 0.01, f'Fonte: {source}', fontsize=9, style='italic', ha='ce
    plt.tight_layout(rect=[0, 0.03, 1, 1])
    buf = io.BytesIO()
    fig.savefig(buf, format='png', dpi=300, bbox_inches='tight')
    buf.seek(0)
    img_b64 = base64.b64encode(buf.read()).decode('utf-8')
    buf.close()
    plt.close(fig)

    display(HTML(f'<div style="text-align:center"><img src="data:image/png;bas



def display_image_with_caption(image_path, figure_number, caption_text, width=
    try:
        with open(image_path, "rb") as image_file:
            encoded_string = base64.b64encode(image_file.read()).decode('utf-8

        html_code = f'''
        <div style="text-align: center; margin: 20px 0;">
            <p style="font-size: 11px; font-weight: bold; margin-bottom: 15px;
                Figura {figure_number} - {caption_text}
            </p>
            <img src="{encoded_string}"
                 alt="Figura {figure_number}"
                 width="{width}"
                 style="border: 1px solid #ddd; padding: 5px; display: block;
            <p style="font-size: 9px; font-style: italic; margin-top: 10px; te
                Fonte: Elaborada pelo autor
            </p>
        </div>
        '''

        display(HTML(html_code))

    except FileNotFoundError:
        print(f"Erro: Arquivo '{image_path}' não encontrado.")
```

```
    except Exception as e:
        print(f"Erro ao carregar imagem: {str(e)}")
```