UNIVERSIDADE FEDERAL DE ALAGOAS

Instituto de Computação – IC

Engenharia de Computação

Sistemas de Controle 2

CONTROLE DIGITAL: MODELAGEM E ANÁLISE DE SISTEMAS EM TEMPO DISCRETO

Aldemir Melo Rocha Filho

Sandoval da Silva Almeida Junior

Tayco Murilo Santos Rodrigues

Maceió – 2022

ÍNDICE

1. Introdução

Este relatório tem como objetivo final apresentar a implementação de dois tipos de controladores digitais, PID e $Deadbeat$. O sistema utilizado para estudo neste documento é o mesmo dos relatórios anteriores só que dessa vez em seu modelo discreto. As tarefas realizadas foram implementadas em linguagem $Python$ com auxílio de módulos presentes nas bibliotecas $control$, $scipy$, $numpy$ e $matplotlib$.

O link para acesso ao notebook com todos os algoritmos utilizados pode ser encontrado na área de anexos.

2. Computador digital inserido no ambiente de controle

Do livro: *Engenharia de Sistemas de Controle – 6ª edição*, temos as seguintes funções atribuídas ao computador digital no que diz respeito ao ambiente de controle:

- Supervisão: Externa à malha de realimentação. Exemplos de funções supervisórias consistem de escalonamento de tarefas, monitoramento de parâmetros e variáveis com relação a valores fora de faixa, ou inicialização do desligamento de segurança.
- Controle: Interno à malha de realimentação. Exemplos de funções de controle são as compensações com avanço e com atraso de fase.

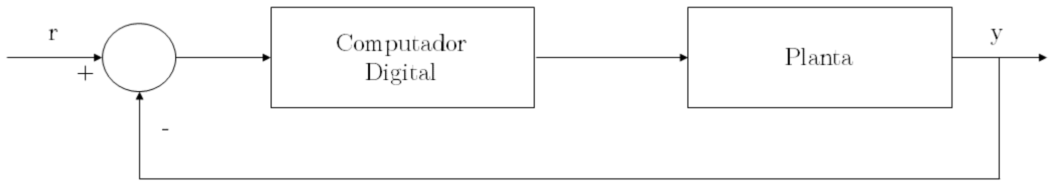


Imagem 1.1: Localização do Computador Digital na malha

3. Representação do sistema em Tempo Contínuo

Como já apresentado nos relatórios das atividades anteriores, temos que nosso sistema em tempo contínuo é descrito pela seguinte função de transferência em S:

$$G(s) = \frac{s+1}{s^3 + s^2 + 2s + 1}$$

Com os seguintes conjuntos de polos e zeros:

$$Polos = \{p_1 \cong -0.5698 + 0j, p_2 \cong -0.2151 \pm 1.3071j\}$$

$$Zeros = \{z_1 = -1\}$$

Logo, temos que nosso sistema em S é *estável* uma vez que todos os seus polos estão contidos no semiplano esquerdo.

4. Discretização do Sistema em tempo contínuo

Se queremos simular o sistema contínuo com um dispositivo digital, precisamos de um método para converter o modelo contínuo em um modelo discreto. Esta conversão é chamada de discretização. Durante a conversão algumas informações do modelo contínuo podem ser perdidas. É importante que essa perda de informação seja minimizada. Aqui, faremos uso de três diferentes técnicas, cada uma com suas vantagens e desvantagens, e todas levando a diferentes representações do mesmo sistema. São elas:

$$\begin{cases} Euler\ com\ T_s = 0.1 \\ Backward\ com\ T_s = 0.1 \\ Tustin\ com\ T_s = 0.1 \end{cases}$$

Onde T_s é a variável que representa o período de amostragem.

Uma vez determinado a técnica de discretização e o valor para T_s, podemos fazer uso do método $cont2discrete(\,)$ presente no módulo $scipy.signal$, a fim de obter o numerador e o denominador de nossa função de transferência em z. O método em questão recebe como entrada o sistema, o valor para T_s e uma $string$ que determina a técnica a ser utilizada. O trecho de código utilizado para este fim pode ser observado abaixo:

```
#Discretizacao utilizando diferentes metodos
metodos = ['bilinear','euler','backward_diff']

#Funcoes de transferencia do sistema em Z
for metodo in metodos:
  sysDisc = signal.cont2discrete((numCont,denCont), 0.1, metodo)
  numD = sysDisc[0][0]
  denD = sysDisc[1]
  print('\n')
  print('Funcao de Transferencia em Z utilizando o metodo ' + metodo +':')
  print('Numerador:  ',numD)
  print('Denominador:',denD)
```

Imagem 4.1: Trecho de código utilizado para discretizar o sistema

4.1. Discretização utilizando o método de *Euler*

O método de Euler faz uso da seguinte abordagem: A área é aproximada pelo retângulo *olhando para a frente* de $(k - 1)$ em direção a k com uma amplitude igual ao valor de a função em $(k - 1)$.

Suprimindo os cálculos envolvidos para simplificar a explicação temos a seguinte relação para discretizar o sistema substituindo a variável s por sua correspondente em z:

$$s = \frac{z - 1}{T_s}$$

O algoritmo presente na **Imagem 4.1** apresentou a seguinte saída quando os parâmetros fornecidos eram aqueles para se realizar a discretização utilizando o método de Euler:

```
Funcao de Transferencia em Z utilizando o metodo euler:
Numerador:   [ 0.00000000e+00  3.10862447e-15  1.00000000e-02  -9.00000000e-03]
Denominador: [ 1.      -2.9      2.82     -0.919]
```

Imagem 4.2: Função de transferência em z utilizando o método de *Euler*

Realizando os devidos arredondamentos, podemos interpretar a função presente na **Imagem 4.2** como a seguinte função de transferência em z:

$$G(z)_E = \frac{10^{-2}z - 9 \cdot 10^{-3}}{z^3 - 2.9\,z^2 + 2.82z - 0.919}$$

Que possui o seguinte conjunto de polos e zeros:

$$Polos_E = \{p_1 \cong 0.9430 + 0j\,, p_2 \cong 0.9785 \pm 0.1307j\}$$

$$Zeros_E = \{z_1 = 0.9\}$$

E a seguinte resposta ao degrau unitário comparada à $G(s)$:

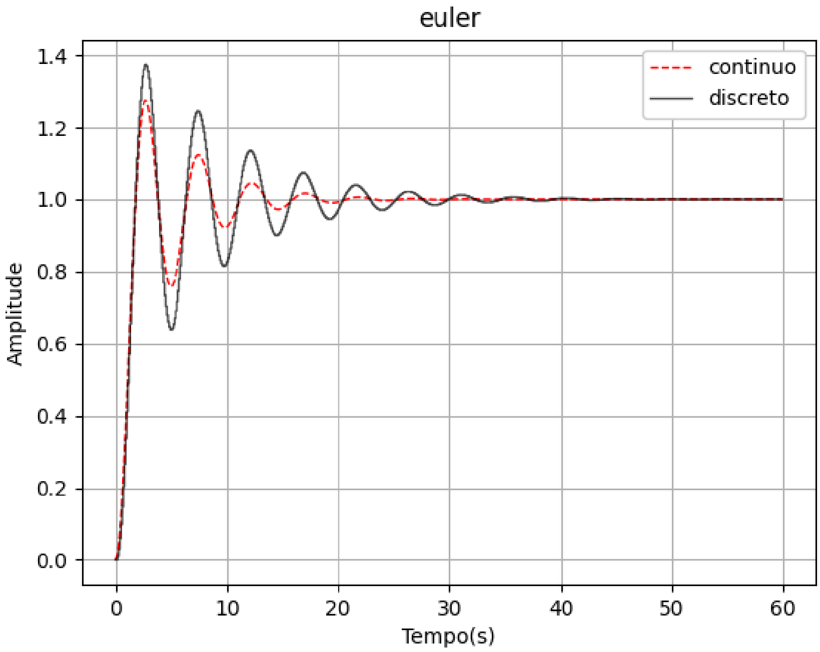


Imagem 4.3: Comparação entre as respostas de $G(s)$ e $G(z)_E$ ao degrau unitário

4.2. Discretização utilizando o método *Backward*

O método de Euler Backward faz uso da seguinte abordagem: A área é aproximada pelo retângulo *olhando para trás* de k em direção a $(k - 1)$ com uma amplitude igual ao valor da função em k.
Suprimindo os cálculos envolvidos para simplificar a explicação temos a seguinte relação para discretizar o sistema substituindo a variável s por sua correspondente em z:

$$s = \frac{z - 1}{z \cdot T_s}$$

O algoritmo presente na **Imagem 4.1** apresentou a seguinte saída quando os parâmetros fornecidos eram aqueles para se realizar a discretização utilizando o método de Euler Backward:

```
Funcao de Transferencia em Z utilizando o metodo backward_diff:
Numerador:    [ 9.81266726e-03 -8.92060660e-03 -4.44089210e-16  4.44089210e-16]
Denominador:  [ 1.           -2.87243533   2.76538805  -0.89206066]
```

Imagem 4.4: Função de transferência em z utilizando o método de *Euler Backward*

Realizando os devidos arredondamentos, podemos interpretar a função presente na **Imagem 4.4** como a seguinte função de transferência em z:

$$G(z)_B = \frac{9.813 \cdot 10^{-3} z^3 - 8.923 \cdot 10^{-3}\, z^2}{z^3 - 2.872z^2 + 2.765z - 0.892}$$

Que possui o seguinte conjunto de polos e zeros:

$$Polos_B = \{p_1 \cong 0.9430 + 0j\,, p_2 \cong 0.9785 \pm 0.1307j\}$$

$$Zeros_B = \{z_1 = 0.9\}$$

E a seguinte resposta ao degrau unitário comparada à $G(s)$:

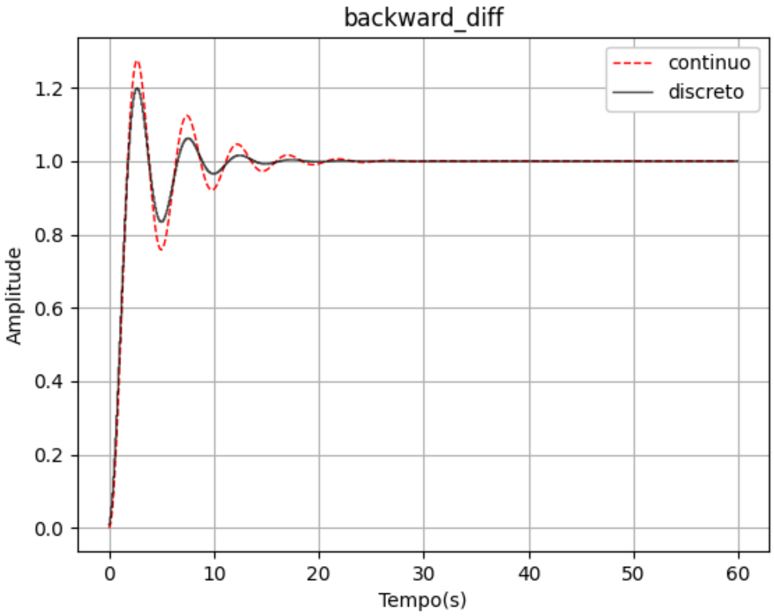


Imagem 4.5: Comparação entre as respostas de $G(s)$ e $G(z)_B$ ao degrau unitário

4.3. Discretização utilizando o método de *Tustin*

O método Trapezóidal($Tustin$) faz uso da seguinte abordagem: Podemos ligar com uma reta o valor de amplitude da função em $(k - 1)$ e em k, e assim, calcular a área do trapézio formado pelos retângulos obtidos partindo das abordagens descritas nos tópicos **4.1** e **4.2**.
Suprimindo os cálculos envolvidos para simplificar a explicação temos a seguinte relação para discretizar o sistema substituindo a variável s por sua correspondente em z:

$$s = \frac{2}{T_s} \cdot \frac{z-1}{z+1}$$

O algoritmo presente na **Imagem 4.1** apresentou a seguinte saída quando os parâmetros fornecidos eram aqueles para se realizar a discretização utilizando o método de Tustin:

```
Funcao de Transferencia em Z utilizando o metodo bilinear:
Numerador:   [ 0.00248786   0.0027248   -0.00201398  -0.00225092]
Denominador: [ 1.          -2.88555858   2.7914939   -0.90498756]
```

Imagem 4.6: Função de transferência em z utilizando o método de *Tustin*

Realizando os devidos arredondamentos, podemos interpretar a função presente na **Imagem 4.6** como a seguinte função de transferência em z:

$$G(z)_T = \frac{0.0025\, z^3 + \ 0.0027z^2 - 0.002z - \ 0.0023}{z^3 - \ -2.8856z^2 + 2.7915\, z - \ 0.905}$$

Que possui o seguinte conjunto de polos e zeros:

$$Polos_T = \{p_1 \ \cong 0.9446 + 0j \,, p_2 \ \cong \ 0.9705 \ \pm 0.1274j\}$$

$$Zeros_T = \{z_1 = \ -1 \,, z_2 \cong 0.9048\}$$

E a seguinte resposta ao degrau unitário comparada à $G(s)$:

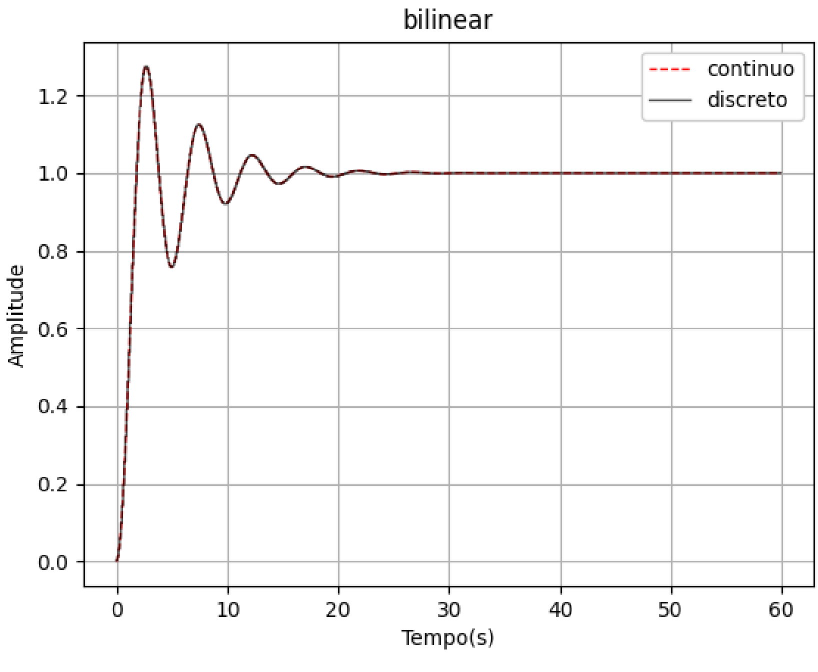


Imagem 3.7: Comparação entre as respostas de $G(s)$ e $G(z)_T$ ao degrau unitário

Partindo dos resultados apresentados nos itens **4.1**, **4.2** e **4.3** podemos concluir que o método de discretização, com $T_s = 0.1$, que apresenta o resultado mais próximo ao sistema em tempo contínuo, é o de $Tustin$, uma vez que houve pouca distorção na saída $Y(z)$ quando comparado ao método de $Euler$ e $Backward$, assim havendo um menor comprometimento no comportamento do sistema.

5. Método de discretização de *Tustin* para T_s variável

Como visto no tópico **4**, o método de *Tustin* é aquele que apresenta menor distorção na saída quando comparado ao sistema em tempo contínuo. Agora, o objetivo é observar o comportamento do sistema quando discretizado com este método, mas levando em consideração diferentes valores para T_s. Os valores que serão utilizados estão presentes no conjunto abaixo:

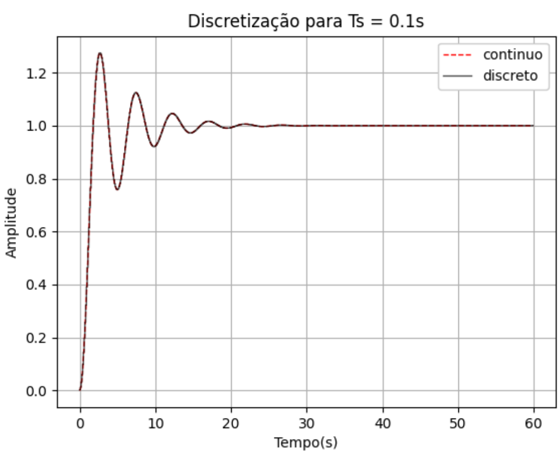
$$Taxa\ de\ amostragem = \{0.1, 0.2, 0.4, 0.6, 0.8, 1\}$$

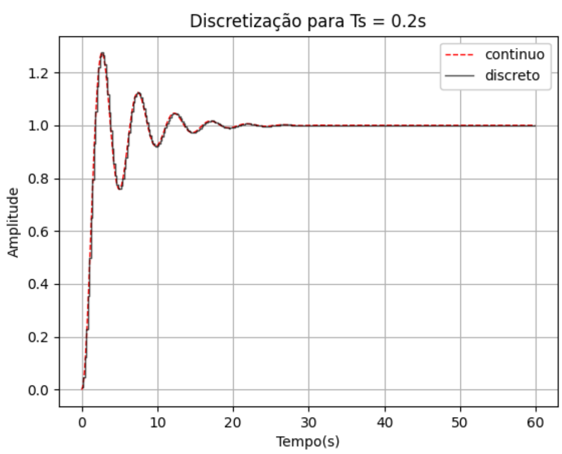
Para realizar a tarefa descrita no parágrafo anterior, faremos uso de dois métodos presentes no módulo *scipy.signal*, são eles: *cont2discrete*( ), para realizar a discretização do sistema e o método *dlsim*( ), para avaliar a resposta do sistema à um sinal de entrada, neste caso, um degrau unitário. A biblioteca *matplotlib* também foi utilizada para gerar os gráficos com os resultados. O trecho de código utilizado junto das saídas obtidas pode ser observado abaixo:

```
periodos = [0.1, 0.2, 0.4, 0.6, 0.8, 1]

#Definicao do vetor de tempo e da entrada
tempo = np.arange(0,60,0.1)
u = np.full(len(tempo),1)

#Resposta do sistema em tempo continuo
tOutCont, yOutCont, x = signal.lsim(sysCont, u, tempo)

#Resposta do sistema discreto com diferentes valores de Ts
for periodo in periodos:
  sysDisc = signal.cont2discrete((numCont,denCont), periodo, 'bilinear')
  OutDisc, yOutDisc = signal.dlsim(sysDisc, u, tempo)
  plt.plot(tOutCont, yOutCont,'r--', linewidth = 1, label = 'continuo')
  plt.step(OutDisc, yOutDisc, 'k', alpha = 0.7, where='post', linewidth=1, label='discreto')
  plt.legend(loc='best', framealpha=1)
  plt.title('Discretizacao para Ts = '+ str(periodo) + 's')
  plt.grid(0.5)
  plt.xlabel('Tempo(s)')
  plt.ylabel('Amplitude')
  plt.show()
  print('\n')
```

Imagem 5.1: Trecho de código utilizado para comparar o sistema discretizado com diferentes valores de T_s

Os gráficos com as diferentes respostas do sistema podem ser visualizados a seguir:





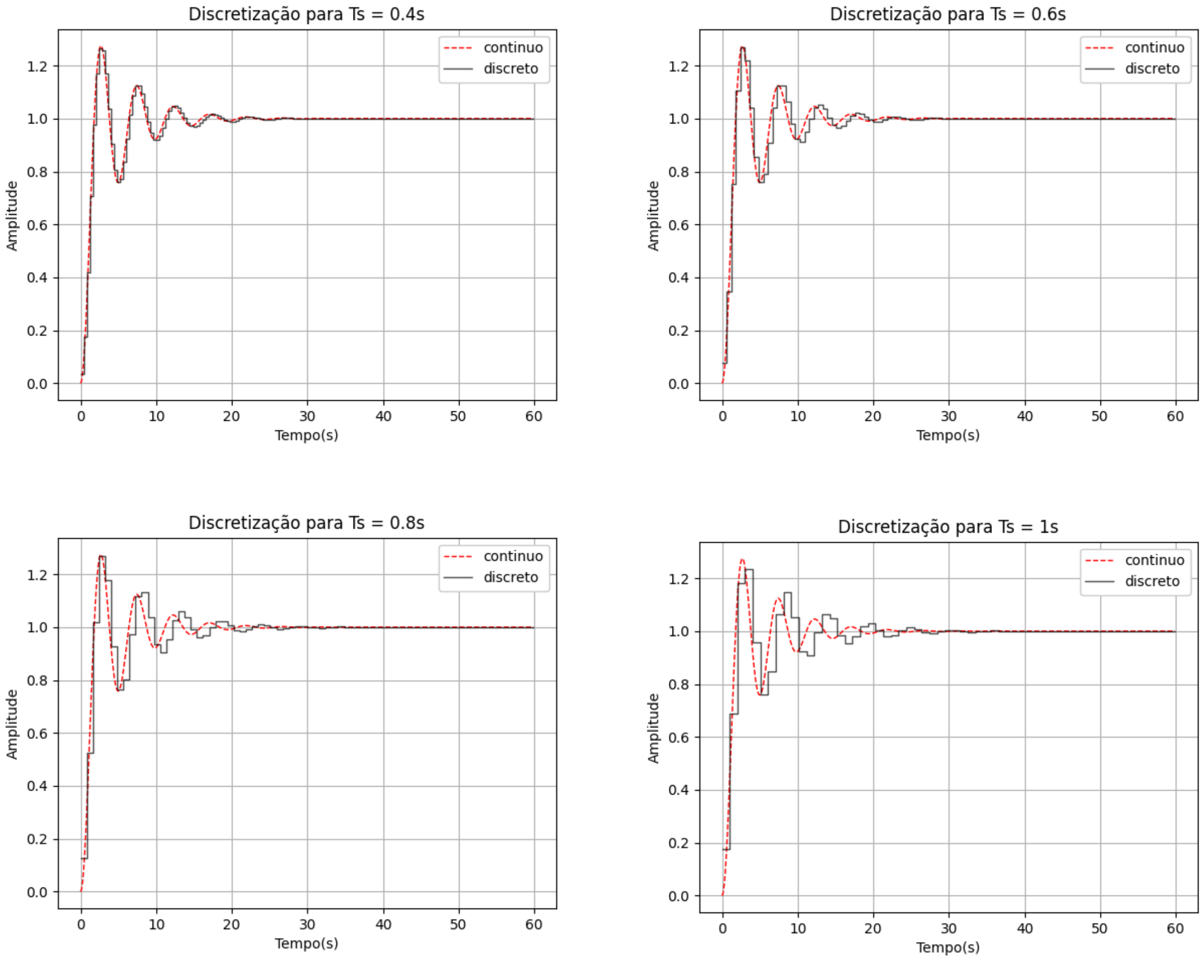


Imagem 5.2: Conjunto de plots que representam as diferentes respostas ao degrau do sistema $G(z)$ quando discretizado utilizando o método de *Tustin* com T_s variável

Podemos observar que a medida em que o valor de T_s varia, temos uma maior perda de informação na resposta do sistema discretizado, quando comparado ao sistema em tempo contínuo. Este comportamento é apresentado pois o "espaçamento" que existe entre as amostras não é suficiente para manter a informação do sinal preservada nos primeiros segundos de simulação, contudo, o mesmo é praticamente indiferente quando o sistema ultrapassa o tempo de acomodação t_s. Portanto, para um projeto de um sistema de controle em z, temos que o valor escolhido para T_s pode variar a depender de quantidade de informação que se deseja obter, de forma que o sinal gerado seja suficientemente equivalente ao seu correspondente em s.

6. Prova de estabilidade do sistema utilizando o critério de *Jury*

6.1. Estabilidade de Sistema Digital via plano z

Uma vez que podemos realizar uma mudança de variável em nossa função de transferência $G(s)$ para obtermos a sua correspondente $G(z)$, também podemos realizar um mapeamento dos pontos presentes no plano s para o plano z. Em específico, se busca mapear os *polos* e *zeros* do sistema entre esses planos para se determinar os critérios de estabilidade e desempenho do sistema.

Do livro: *Engenharia de Sistemas de Controle – 6ª edição*, temos a seguinte assertiva: Cada região do plano s pode ser mapeada em uma região correspondente no plano z. Dessa forma, um sistema de controle digital é estável se todos os polos da função de transferência em malha fechada, $T(z)$, estão dentro do círculo unitário no plano z, instável se algum polo está fora do círculo unitário e/ou se existem polos de multiplicidade maior que um sobre o círculo unitário, e marginalmente estável se polos de multiplicidade um estão sobre o círculo unitário e todos os demais polos estão dentro do círculo unitário.

A representação do mapeamento pode ser observada abaixo.

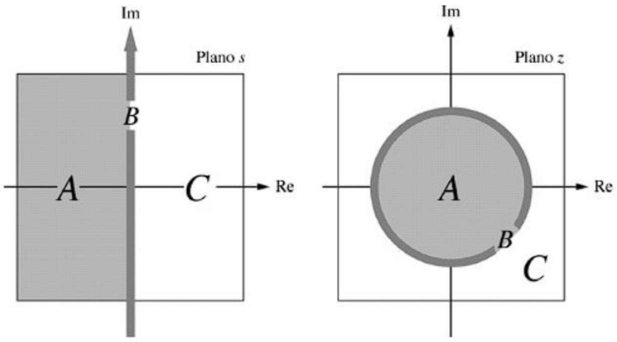


Imagem 6.1: Mapeamento de regiões entre o plano s e o plano z. Fonte: NISE, Norman. Engenharia de Sistemas de Controle. 6ª ED. (2012, p. 1059)

O critério de $Jury$ é a versão discreta do critério de $Routh - Hurwitz$. Permite que seja verificado se todas as raízes de um polinômio estão dentro do círculo unitário sem que se precise calcular explicitamente as raízes.

O primeiro passo para se observar a estabilidade partindo deste critério é montar a matriz de $Jury$, partindo dos coeficientes do denominador de nossa função de transferência $G(z)$.

Seja:

$$G(z) = \frac{b(z)}{a(z)} = \frac{b(z)}{a_0 z^n + a_1 z^{n-1} + \cdots + a^n}$$

Com $a_0 > 0$, a matriz de $Jury$ é formada por:

a_0	a_1	$\cdots$	a_n
a_n	a_{n-1}	$\cdots$	a_0
b_0	b_1	$\cdots$	
b_{n-1}	b_{n-2}	$\cdots$	
c_0	c_1	$\cdots$	
$\vdots$	$\vdots$		

De forma que:

$$b_0 = a_0 - \frac{a_n}{a_0} a_n$$

$$b_1 = a_1 - \frac{a_n}{a_0} a_{n-1}$$

$$\vdots$$

$$b_k = a_k - \frac{a_n}{a_0} a_{n-k}$$

$$\vdots$$

$$c_k = b_k - \frac{b_{n-1}}{b_0} b_{n-1-k}$$

$$\vdots$$

Por fim, uma vez que a matriz está completamente montada, temos que se $a_0 > 0$, então todas as raízes estarão dentro do círculo unitário, sse, todos os termos da primeira coluna das linhas ímpares forem positivos. Se nenhum elemento da primeira coluna das linhas ímpares for nulo, o número de raízes fora do círculo unitário é igual ao número de elementos negativos.
Para o nosso sistema, a fim de manter a maior coerência dos dados, vamos tomar como modelo a função de transferência discretizada utilizando o método de $Tustin$, com um $T_s = 0.1$. O trecho de código utilizado e as saídas obtidas podem ser observadas abaixo.

Primeiro, definimos um método para inverter uma linha qualquer com o propósito de montar as linhas pares de nossa matriz.

```python
#Metodo para inverter um vetor qualquer

def inverseVector(vector):
  aux = vector[::-1]
  return aux
```

Imagem 6.2: Trecho de código utilizado para inverter um vetor qualquer

Depois montamos as linhas subsequentes da matriz.

```python
#Criamos as duas primeiras linhas da matriz de Jury
coefficient = [1,-2.88556,2.79149,-0.90499]
jury = []
#Linha 1 com os coeficiente
jury.append(coefficient)
#Linha 2 com os coeficientes invertidos
jury.append(inverseVector(coefficient))

#Setup
line = 2
odd_line = 0
column = len(coefficient)

#Montagem da tabela
while(column != 1):

  #vetor auxiliar para guardar os resultados da linha atual
  aux_vector = []
  # calculamos an/a0 atual
  mult = round((jury[line-2][column-1])/(jury[line-2][0]),5)
  #Calculando a linha atual
  for aux in range(column-1):
    aux_vector.append(round(jury[line-2][aux] - (jury[line-1][aux] * mult),5))

  #Armazenando a linha atual na matriz de Jury
  jury.append(aux_vector)
  #Armazenando o inverso da linha atual na matriz de Jury
  jury.append(inverseVector(aux_vector))
  #Decrementando o valor que representa a coluna
  column = column - 1
  #Incrementando o valor que representa a linha
  line = line + 2
```

Imagem 6.3: Trecho de código utilizado para montar toda a matriz de *Jury*

Podemos usar o trecho de código abaixo para mostrar a matriz de *Jury* montada no passo anterior.

```
#Matriz de Jury do sistema
for i in range(len(jury)):
  print(jury[i])
```

Imagem 6.4: Trecho de código utilizado para exibir a matriz de *Jury*

```
[1, -2.88556, 2.79149, -0.90499]
[-0.90499, 2.79149, -2.88556, 1]
[0.18099, -0.35929, 0.18009]
[0.18009, -0.35929, 0.18099]
[0.0018, -0.00179]
[-0.00179, 0.0018]
[2e-05]
[2e-05]
```

Imagem 6.5: Matriz de *Jury* do sistema

Podemos interpretar a saída da **Imagem 5.4** como a seguinte matriz:

1	-2.89	2.79	-0.90
-0.90	2.79	-2.89	1
0.18	-0.36	0.18	
0.18	-0.36	0.18	
0.0018	-0.00179		
-0.00179	0.0018		
$2 \cdot 10^{-5}$			

Por fim, verificamos a condição de estabilidade do sistema. O trecho de código utilizado para este fim e a saída obtida podem ser observados abaixo.

```
#Verificando o criterio de Jury
while(1):

  #Se algum elemento presente em uma linha impar da primeira coluna for <=0
  if( jury[odd_line][0] <= 0 ):
    print('Existe raiz do polinomio fora do circulo unitario, logo o sistema e instavel.')
    break

  #Incremento na variavel que representa a linha impar atual
  odd_line = odd_line + 2

  #Se a linha impar == ultima linha --> atende ao criterio de estabilidade
  if(odd_line == line):
    print('As raizes do polinomio estao todas dentro do circulo unitario, logo o sistema e
      estavel.')
    break
```

Imagem 6.6: Trecho de código utilizado para verificar o critério de *Jury*

```
As raizes do polinomio estao todas dentro do circulo unitario, logo o sistema e estavel.
```

Imagem 6.7: Saída do critério de *Jury*

Podemos atestar a afirmação da **Imagem 6.7** realizando uma *plotagem* do lugar geométrico das raízes do sistema $G(z)$. Para este fim, foi feito uso do método $rlocus(\,)$, presente no módulo *matlab* da biblioteca *control*. O trecho de código utilizado junto de sua saída pode ser observado abaixo.

```
#Plot do lugar geometrico das raizes do sistema

sysCont = matlab.TransferFunction([0,0,1,1], [1,1,2,1])
sysDisc = matlab.c2d(sysCont, 0.1, 'bilinear')

rlocus = matlab.rlocus(sysDisc)
plt.grid(alpha = 0.5)
plt.show()
```

Imagem 6.8: Trecho de código utilizado para exibir o lugar geométrico das raízes

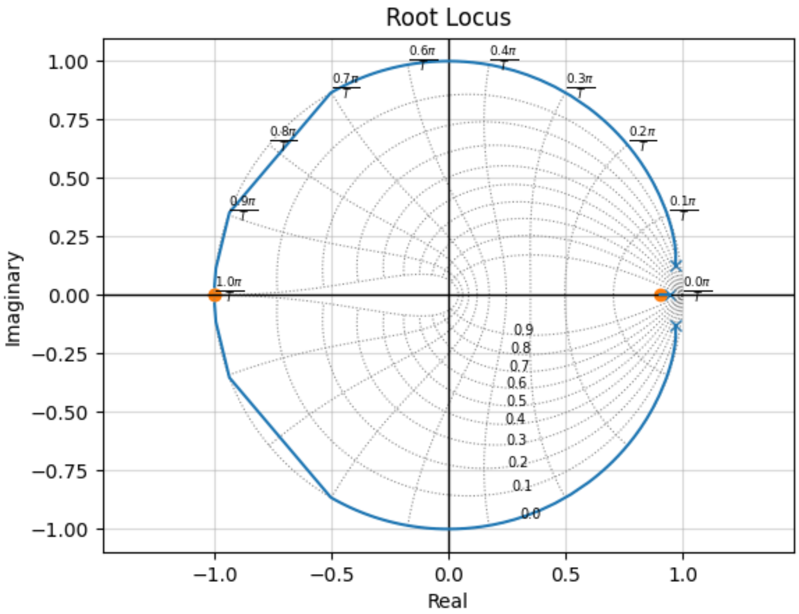


Imagem 6.9: Lugar geométrico das raízes

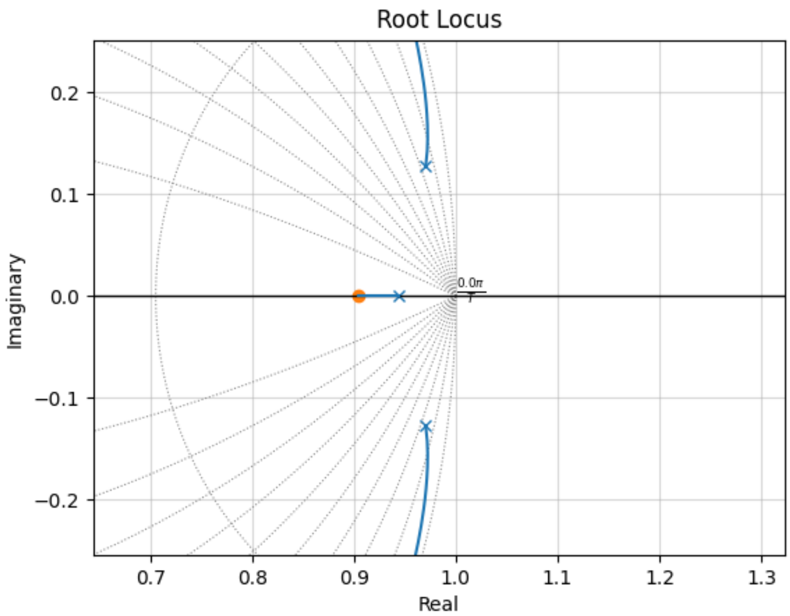


Imagem 6.10: Lugar geométrico das raízes ampliado

Portanto, partindo da **Imagem 6.10**, podemos atestar que o algoritmo responsável por gerar e verificar o critério de $Jury$ gerou a saída correta para o sistema em questão. Dessa forma, temos que $o\ sistema\ G(z)\ \acute{e}\ est\acute{a}vel$.

7. Implementação de um controlador PID digital

Esse tipo de controlador possui características interessantes para manter sob controle a saída de um determinado sistema. Utilizado em sistemas de malha fechada, um controle desse tipo pode ser ajustado para oferecer a resposta desejada e manter a saída em um valor estável com um mínimo de erro possível, apenas com o ajuste de três parâmetros.

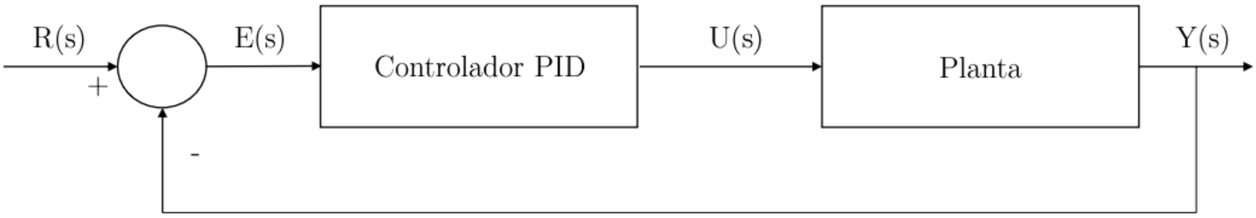


Imagem 7.1: Posição do controlador em malha fechada

Em tempo contínuo o Controlador $P.I.D.$ pode ser descrito com o seguinte modelo matemático:

$$u(t) = k_p e(t) + k_i \int e(t)\,dt + k_d \frac{d}{dt} e(t)$$

Agora, escrevendo uma aproximação para cada componente, temos:

$$u[k] = k_p e[k] + k_i h \sum_{i}^{k-1} e[i] + k_d \frac{e[k] - e[k-1]}{h}$$

Perceba que a representação acima é uma aproximação direta do modelo matemático de um controlador $P.I.D.$ digital. Temos que essa representação apesar de direta é ineficiente, uma vez que, do ponto de vista computacional, temos um uso excessivo de memória e aumento exponencial do tempo que se leva para calcular a nova saída $u[k]$, devido ao somatório existente na parcela integrativa. Sendo assim, vamos partir de um outro modelo em tempo contínuo. Considere o seguinte:

$$\frac{d}{dt} u(t) = k_p \frac{d}{dt} e(t) + k_i e(t) + k_d \frac{d^2}{dt^2} e(t)$$

O modelo matemático acima é equivalente ao apresentado anteriormente, uma vez que, apenas derivamos a equação no tempo. Agora, escrevendo uma aproximação para cada componente, temos:

$$\frac{u[k] - u[k-1]}{h} = k_p \frac{e[k] - e[k-1]}{h} + k_i e[k] + k_d \frac{e[k] - e[k-1] - e[k-2]}{h}$$

Fazendo os devidos ajustes na equação temos a seguinte representação para um controlador $P.I.D.$ digital:

$$u[k] = u[k-1] + k_p (e[k] - e[k-1]) + k_i h e[k] + \frac{k_d}{h} (e[k] - e[k-1] - e[k-2])$$

Onde:

- k_p é o coeficiente de ação proporcional
- k_i é o coeficiente de ação integrativa
- k_d é o coeficiente de ação derivativa
- $u[k]$ é a próxima entrada para o sistema
- $u[k-1]$ é a entrada anterior do sistema
- $e[k]$ é o valor de erro atual do sistema
- $e[k-1]$ é o valor de erro anterior ao instante k
- $e[k-2]$ é o valor de erro anterior ao instante $k-1$
- $h = T_s$ que para o nosso caso vale 0.1

Para a implementação computacional do controlador partimos da implementação individual de cada bloco do sistema em malha fechada. Garantindo assim o comportamento de cada componente junto de suas entradas e saídas corretas.

Para o componente responsável por calcular e atualizar o erro, temos:

```
## Bloco de erro

def erro(e0, e1, e2, setpoint, output):

  eAtual = setpoint - output
  e2 = e1
  e1 = e0
  e0 = eAtual

  return e0, e1, e2
```

Imagem 7.2: Trecho de código responsável por atualizar o erro

O trecho de código acima recebe os últimos três valores de erro, o sinal de referência e a última saída do sistema para calcular e gerar a saída $E(z)$ com os três valores de erro mais recentes.

Para o componente que desempenha o papel do controlador $P.I.D.$ temos:

```
#Bloco PID

def PID(e0, e1, e2, uAnterior):

  kp = 5
  ki = 2.8
  kd = 0.000001

  u = uAnterior + kp*(e0 - e1) + (ki*0.1)*(e0) + (kd/0.1)*(e0 - e1 - e2)
  return u
```

Imagem 7.3: Trecho de código responsável por gerar o sinal de controle para o sistema

O trecho de código acima recebe os três valores de erro mais recentes junto do último sinal de controle gerado para calcular e gerar na saída a próxima entrada do sistema $U(z)$. Para se alcançar o comportamento desejado em malha fechada, os parâmetros k_p, k_i e k_d foram definidos respectivamente como 5, 2.8 e 0.000001. A escolha desses valores foi realizada de maneira arbitrária.

Para o componente que representa o sistema temos:

```
#Bloco do sistema

def system(discreteSystem, inputSignal, timeStep):

  tOutSystem, yOutSystem = signal.dlsim(discreteSystem, inputSignal, timeStep)
  return tOutSystem, yOutSystem
```

Imagem 7.4: Trecho de código responsável por gerar a resposta do sistema ao sinal de controle

O trecho de código acima recebe o sistema discretizado junto dos vetores que representam o sinal de entrada e o tempo de simulação, apresentando na saída a resposta $Y(s)$.

Agora, definimos as condições iniciais do sistema em malha fechada:

```
#Set up das condicoes iniciais do sistema

#Valores de erro acumulados com valores inicias nulos
e0 = 0
e1 = 0
e2 = 0

#Valores iniciais para setpoint e saida
setPoint = 1

#Valor do time step (fixo)
Ts = 0.1

#Instantes de tempo (0s —> 60s com passo = Ts —> 600 instantes de tempo)
instanteTempo = 0

#Vetor com os intantes de tempo (tamanho 600)
tempo = np.arange(0, 60, Ts)

#Vetor com as entradas
u = np.zeros(len(tempo))
u[0] = 1

#Vetor com as saidas
ysOut = np.zeros(len(tempo))
tsOut = np.zeros(len(tempo))

#Modelagem do sistema discreto
num = [0,0,1,1]
den = [1,1,2,1]
dt = 0.1
discreteSystem = signal.cont2discrete((num,den), dt, 'bilinear')
```

Imagem 6.5: Trecho de código responsável por definir as condições iniciais do sistema em malha fechada

O trecho de código acima garante os erros iniciais do sistema iguais a zero, o valor do sinal de referência como um e o valor da taxa de amostragem como 0.1. Garante também a criação dos vetores que representam a entrada e a saída da planta, assim como o tempo de simulação, todos preenchidos também com zeros, pôr fim a discretização é realizada utilizando o método de $Tustin$ com $T_s = 0.1$.

Uma vez que a lógica dos blocos individuais foi montada, suas entradas e saídas modeladas de maneira correta e as condições iniciais do sistema estabelecidas, resta apenas simular a malha fechada garantindo a sequência lógica de execução correta. O trecho de código responsável por simular a lógica do sistema em malha fechada pode ser observado abaixo.

```
#Sistema em malha fechada

while(instanteTempo < len(tempo)-1):

  #Resposta do sistema
  tsOut, ysOut = system(discreteSystem, u, tempo)
  output = ysOut[instanteTempo]

  #Valor de erro para o instante atual
  e0, e1, e2 = erro(e0, e1, e2, setPoint, output)

  #Erro + entrada anterior usadas no controlador para gerar a proxima entrada do sistema
  u[instanteTempo + 1] = PID(e0, e1, e2, output)

  instanteTempo += 1
```

Imagem 6.6: Simulação do sistema em malha fechada

Ao fim da execução do algoritmo presente na **Imagem 6.6** temos as seguintes saídas:

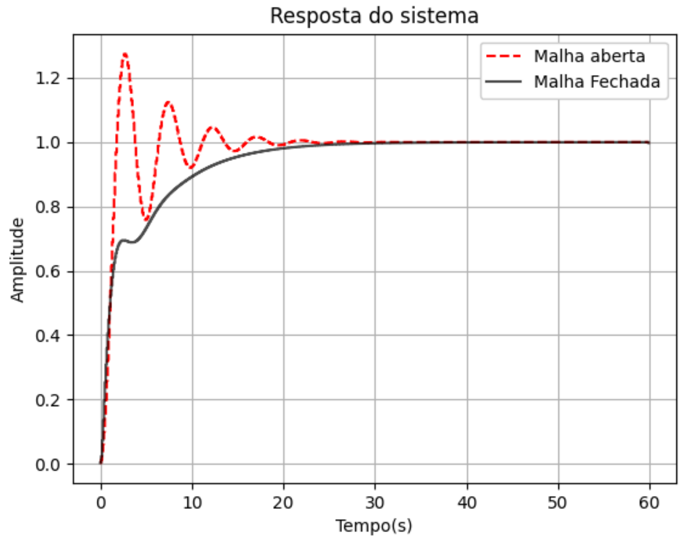


Imagem 6.7: Comparação entre as respostas do sistema em Malha aberta e Malha Fechada

Partindo da **Imagem 6.7,** podemos observar que o comportamento desejado para o sistema foi alcançado fazendo uso do controlador em questão, com os parâmetros escolhidos de maneira apropriada.

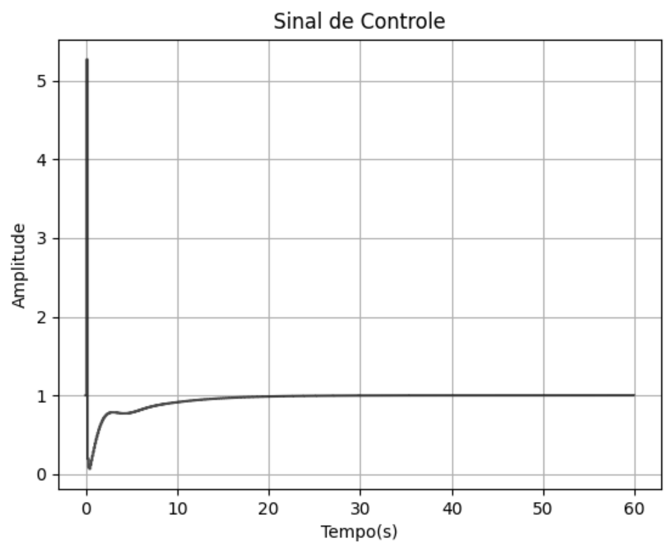


Imagem 6.8: Sinal de controle gerado no bloco $P.I.D.$ utilizado como entrada $U(z)$ no sistema

8. Implementação de um controlador *DEADBEAT*

Temos que este tipo de controlador é projetado com o objetivo de fazer com que a saída do sistema vá para o estado estacionário no menor tempo de *time steps* possível. Para o sistema em questão, onde temos $T_s = 0.1$ é desejado que após o instante de tempo $t = 0.1s$ o sistema estabilize sobre o sinal de entrada.

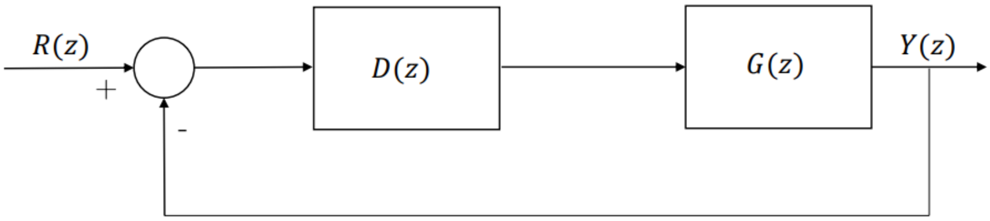


Imagem 8.1: Posição do controlador em malha fechada

Partindo da **Imagem 8.1,** temos que o sistema em malha fechada é o seguinte formato:

$$\overline{G}(z) = \frac{Y(z)}{R(z)} = \frac{G(z)D(z)}{1 + G(z)D(z)} \; (eq. 8.1)$$

Partindo do desempenho desejado para o sistema descrito anteriormente, buscamos que o sistema em malha fechada seja do seguinte formato:

$$\overline{G}(z) = z^{-k}, k \geq 1 \; (eq. 8.2)$$

De forma que o parâmetro k é definido a partir do tempo morto do sistema, que é o tempo em que o mesmo leva para apresentar uma resposta na saída, quando alimentado na entrada com um sinal qualquer.

Uma vez que sabemos a função de transferência do sistema em malha fechada, podemos criar uma relação de igualde entre as $eq. 8.1$ e $eq. 8.2$ para que seja possível determinar a função de transferência de $D(z)$ em malha fechada, que é o bloco do nosso controlador. Assim:

$$\overline{G}(z) = \frac{Y(z)}{R(z)} = \frac{G(z)D(z)}{1 + G(z)D(z)} \rightarrow D(z) = \frac{1}{G(z)} \cdot \frac{\overline{G}(z)}{1 - \overline{G}(z)} \rightarrow D(z) = \frac{1}{G(z)} \cdot \frac{z^{-k}}{1 - z^{-k}}$$

Portanto:

$$D(z) = \frac{1}{G(z)} \cdot \frac{1}{z^k - 1} \; (eq. 8.3)$$

É importante notar que o projeto do controlador *DeadBeat* consiste em posicionar todos os polos do sistema em malha fechada, em $z = 0$. Esses polos correspondem a resposta mais rápida possível do sistema. Agora, partimos para a modelagem do controlador *DeadBeat* para o nosso caso.

Partindo da **Imagem 3.7**, temos que o tempo morto do sistema é nulo, uma vez que o mesmo apresenta algum tipo de resposta quase que imediatamente no mesmo instante de tempo em que é alimentando com um sinal na entrada. Uma vez que precisamos que o valor seja $k \geq 1$, não podemos tomar $k = 0$, dessa forma vamos usar $k = 1$.

Observando a **Imagem 4.6,** temos que em z o sistema em malha aberta pode ser descrito utilizando a seguinte função de transferência:

$$G(z) = \frac{0.00248786z^3 + 0.0027248z^2 - 0.00101398z - 0.00225092}{z^3 - 2.88555858z^2 + 2.7914939z - 0.90498756}$$

Dessa forma, temos que o inverso de $G(z)$ tem o seguinte formato:

$$\frac{1}{G(z)} = \frac{z^3 - 2.88555858z^2 + 2.7914939z - 0.90498756}{0.00248786z^3 + 0.0027248z^2 - 0.00101398z - 0.00225092}$$

Logo, a função de transferência $D(z)$ do controlador tem o seguinte formato:

$$D(z) = \frac{z^3 - 2.88555858z^2 + 2.7914939z - 0.90498756}{0.00248786z^3 + 0.0027248z^2 - 0.00101398z - 0.00225092} \cdot \frac{1}{(z-1)} =$$

$$= \frac{z^3 - 2.88555858z^2 + 2.7914939z - 0.90498756}{0.00248786z^4 + 0.00023694z^3 - 0.00473878z^2 - 0.00023694z + 0.00225092}$$

Uma vez que temos o nosso controlador modelado, resta apenas realizar a implementação. De forma análoga ao controlador $P.I.D.$, partimos da implementação individual de cada bloco do sistema em malha fechada, garantindo assim o comportamento de cada componente junto de suas entradas e saídas corretas.

Para o componente responsável por calcular e atualizar o erro, temos:

```
## Bloco de erro

def erroDB(setpoint, output):
  e = setpoint - output
  return e
```

Imagem 8.2: Trecho de código responsável por atualizar o erro

O trecho de código acima recebe como parâmetro o valor do sinal de referência junto da última saída do sistema e calcula o valor de erro $E(z)$ para ser usado como entrada no controlador.

Para o componente responsável por desempenhar o papel do controlador, temos:

```
#Bloco que representa o controlador

def DeadBeat(controllerDbTf, erro, tempo):
  tOutController, yOutController = signal.dlsim(controllerDbTf, erro, tempo)
  return tOutController,yOutController
```

Imagem 8.3: Trecho de código responsável por calcular a resposta do controlador

O código acima recebe como parâmetro a função de transferência que descreve o sistema do controlador em z, o valor de erro mais atual e o vetor de tempo para poder calcular o valor do sinal de controle $U(z)$, que vai servir de entrada para a planta.

Para o componente responsável por desempenhar o papel da planta do sistema, temos:

```
#Bloco que representa o Sistema

def system(discreteSystem, inputSignal, timeStep):
  tOutSystem, yOutSystem = signal.dlsim(discreteSystem, inputSignal, timeStep)
  return tOutSystem, yOutSystem
```

Imagem 8.4: Trecho de código responsável por calcular a resposta da planta do sistema

O trecho de código acima, recebe o sistema discretizado junto dos vetores que representam o sinal de controle e o tempo de simulação, apresentando na saída a resposta $Y(s)$.
Agora, definimos as condições iniciais do sistema em Malha Fechada:

```
#Set up das condiCOes iniciais do sistema

#Valores iniciais para setpoint e saida
setPoint = 1

#Valor do time step
Ts = 0.1

#Instantes de tempo (0s —> 60s com passo = Ts —> 600 instantes de tempo)
instanteTempo = 0

#Vetor com os intantes de tempo (tamanho 600)
tempo = np.arange(0, 30, Ts)

#Valores de erro acumulados com valores inicias nulos
e = np.zeros(len(tempo))
e[0] = setPoint
#Vetor com o sinal de controle
u = np.zeros(len(tempo))

#Vetor com as saidas
ysOut = np.zeros(len(tempo)) #Saida do sistema
ycOut = np.zeros(len(tempo)) #Saida do controlador
tsOut = np.zeros(len(tempo)) #Instantes de tempo

#Modelagem da funcao de transferencia do controlador em Z
numDb = [ 1.          , −2.88555858,  2.7914939 , −0.90498756]
denDb = [0.00248786, 0.00023694, −0.00473878, −0.00023694, 0.00225092]
controllerDbTf = signal.TransferFunction(numDb,denDb, dt = 0.1)

#Modelagem da funcao de transferencia do sistema em Z
numSDb = [ 0.00248786,  0.0027248 , −0.00201398, −0.00225092]
denSDb = [ 1.          , −2.88555858,  2.7914939 , −0.90498756]
systemDbTf = signal.TransferFunction(numSDb, denSDb, dt = 0.1)
```

Imagem 8.5: Trecho de código responsável por definir as condições iniciais do sistema em malha fechada

O trecho de código acima garante o erro inicial do sistema igual a zero, o valor do sinal de referência igual a um e o valor da taxa de amostragem em 0.1. Garante também a criação dos vetores que representam a entrada e a saída do sistema, assim como o tempo de simulação, todos preenchidos também com zeros, pôr fim, a representação do controlador e da planta é montada utilizando $T_s = 0.1$.
Uma vez que a lógica dos blocos individuais foi montada, suas entradas e saídas modeladas de maneira correta e as condições iniciais do sistema estabelecidas, resta apenas simular a malha fechada garantindo a sequência lógica de execução correta. O trecho de código responsável por simular a lógica do sistema em malha fechada pode ser observado abaixo.

```
#Sistema em malha fechada

while(instanteTempo < len(tempo)):

  #Resposta do controlador
  tsOut, ycOut = DeadBeat(controllerDbTf, e, tempo)
  output = ycOut[instanteTempo]
  u[instanteTempo] = output

  #Resposta do sistema
  tsOut, ysOut = system(systemDbTf, u, tempo)
  output = ysOut[instanteTempo]

  #Valor de erro para o instante atual
  e[instanteTempo] = erroDB(setPoint, output)

  instanteTempo += 1
```

Imagem 8.6: Simulação do sistema em malha fechada

Ao fim da execução do algoritmo, presente na **Imagem 8.6,** temos as seguintes saídas:

Imagem 8.7: Comparação entre as respostas do sistema em Malha aberta e Malha fechada

Agora, para avaliação do critério de desempenho, precisamos verificar se o sistema apresenta a melhor resposta possível. A resposta em questão aconteceria no instante de tempo $t = 0.1s$, que é o momento que ocorre a primeira amostragem. Diminuindo o intervalo de simulação para apenas $0.2s$, podemos observar com mais clareza o eixo de tempo do nosso $plot$.

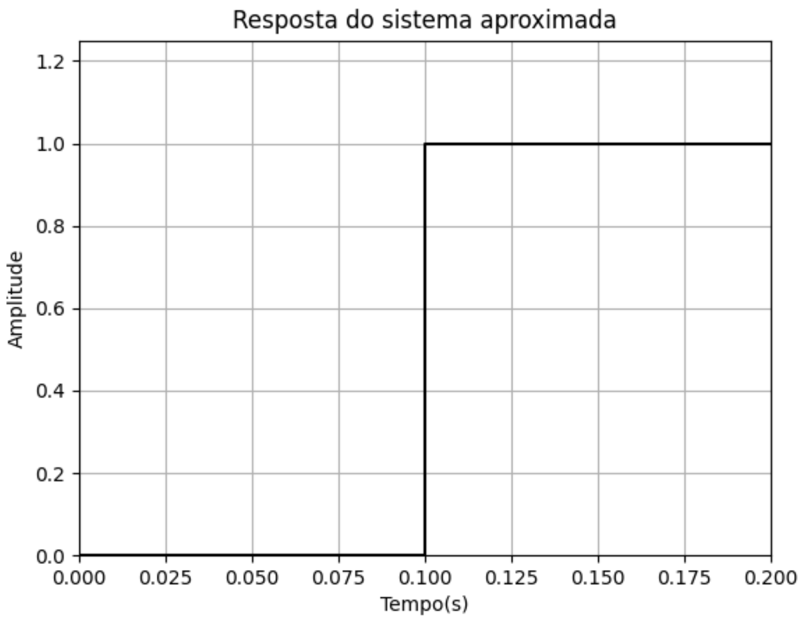


Imagem 8.8: Observação do critério de desempenho do controlador

Podemos perceber que a resposta do sistema acontece em $t = 0.1s$, portanto, o controlador projetado está dentro do parâmetro de desempenho desejado.

As imagens a seguir apresentam o sinal de controle gerado por $D(z)$

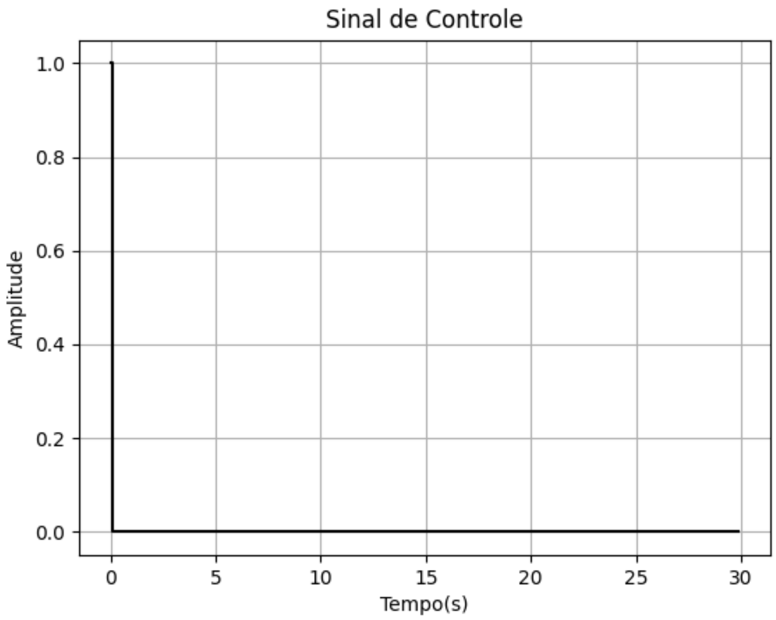


Imagem 8.9: Sinal de controle gerado

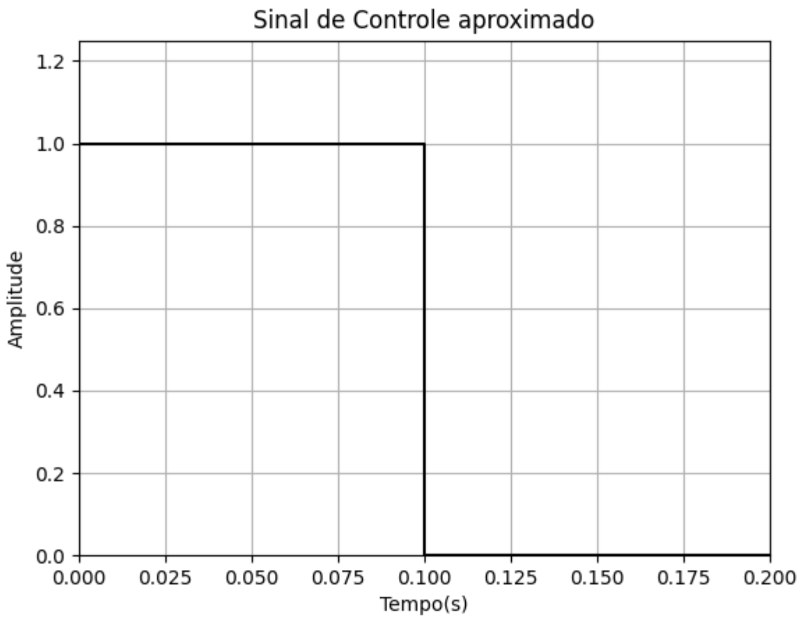


Imagem 8.10: Observação do critério de desempenho do controlador (sinal de controle)

9. Anexos

ANEXO A – Notebook com as implementações realizadas para a análise deste documento:

<[Sistemas de Controle 2 - AB2]>

10. Bibliografia

NISE, Norman. Engenharia de Sistemas de Controle. 6ª ED. Rio de Janeiro: LTC - Livros Técnicos e Científicos Editora Ltda, 2012.

OGATA, Katsushiro. Engenharia de controle Moderno. 5ª ED. São Paulo: Pearson Education do Brasil, 2011.

KLUEVER, Craig. Sistemas Dinâmicos – Modelagem, Simulação e Controle. 1ª ED. Rio de Janeiro: LTC - Livros Técnicos e Científicos Editora Ltda, 2017.

PYTHON SOFTWARE FOUNDATION. Python Language Site: Python Control Systems Library, 2022. Página de documentação. Disponível em: <https://python-control.readthedocs.io/en/0.9.1/>. Acesso em: 06 de Maio. de 2022.

PYTHON SOFTWARE FOUNDATION. Python Language Site: Signal processing, 2022. Página de documentação. Disponível em: < https://docs.scipy.org/doc/scipy/reference/signal.html>. Acesso em: 06 de Maio. de 2022.