A MATLAB Tutorial For Signals and Systems and Related Subjects (Portuguese)

Book · June 2015

DOI: 10.13140/RG.2.1.2320.9443

1 author:

Gabriel Haberfeld

Pontifícia Universidade Católica do Rio de Janeiro

1 PUBLICATION **0** CITATIONS

SEE PROFILE

Tutorial de MATLAB para Sinais e Sistemas e Matérias Relacionadas

Departamento de Engenharia Elétrica

Gabriel Barsi Haberfeld

15 de Junho de 2015

Conteúdo

Parte I
Comandos Básicos

1 Introdução

O MATLAB é uma ferramenta muito poderosa para automatizar qualquer tipo de cálculo numérico extenso, ele se especializa em contas com ponto flutuante (variável tipo float, double ou de precisão estendida, como visto nos cursos de programação do Ciclo Básico). Vamos começar identificando as janelas do MATLAB. A principal é a Command Window, lá você pode digitar e executar comandos e chamadas de função. A janela Workspace mostra todas variáveis existentes do seu programa e seus tipos e conteúdos. A Janela Editor pode ser aberta criando-se um novo Script (Botão New Script) e é onde você vai montar seu programa.

2 Command Window e Workspace

Comece digitando o comando `x = 2` na Command Window e veja o que acontece.

```
x = 2
```

```
x =

     2
```

A Command Window mostrou o valor associado a x, e no Workspace vemos a variável e seu valor associado. Dando dois cliques em x no Workspace pode-se explorar mais a variável, isso pode ser feito com qualquer tipo de variável (float, vetor, matriz, simbólicas).

Tente agora o a sequência de comandos abaixo:

```
x = 3;
y = 2;
c = x - y
```

```
c =

     1
```

A adição do ponto e vírgula omite o resultado, e as variáveis podem ser diretamente utilizadas em outras contas e expressões. Vendo seu Workspace pode reparar que tudo foi atualizado e o valor anterior perdido.

Tente agora os comandos:

```
clc
clear all
```

`clc` limpa a Command Window e `clear all` limpa todas as variáveis para evitar erros em programas longos e/ou repetitivos. Um programa bem escrito não requer uma limpeza de variáveis no meio dele, porém existem casos especiais.

3 Editor - Utilizando Scripts

Tente realizar novamente a sequência de comandos:

```
x = 3;
y = 2;
c = x - y
%
```

```
c =

     1
```

Agora tente mudar o valor de x de 3 para 4. Como pode ter percebido não há como editar uma linha já executada na Command Window. Para poder ter mais liberdade de edição temos que criar um script, que é como se fosse um programa a ser executado e que podemos modificar a vontade.

Tente criar um script clicando em "New Script", no canto superior esquerdo. Uma nova janela se abriu com o nome Editor. É boa prática sempre associar um nome ao seu programa antes de qualquer outra etapa. Salve o script em branco com qualquer nome que desejar, **observando que não se pode salvar com espaços no nome do arquivo** .

É boa prática iniciar os scripts com:

```
clc
clear all
close all
```

`close all` é uma adição aos outros dois, e fecha todas as janelas auxiliares de imagem.

4 Vetores e Matrizes

O MAT em MATLAB não é de Math, e sim de Matrix. O grande poder do MATLAB está em toda a liberdade que trabalhar com sistemas multidimensionais oferece. É interessante procurar no help do MATLAB métodos de como declarar vetores e matrizes, pois existem diversos e cada um é otimizado para um certo tipo de tarefa, os que serão demostrar são os mais fáceis de entender porém não necessariamente os mais rápidos computacionalmente.

Para declarar um vetor basta o comando:

```
x = [1 2 3 4 5]
```

```
x =

     1     2     3     4     5
```

Pode-se concatenar vetores a vontade, respeitando-se as dimensões:

```
x = [0 x 6]
```

```
x =

     0     1     2     3     4     5     6
```

```
y = [x 7]
```

```
y =

     0     1     2     3     4     5     6     7
```

Matrizes podem ser declaradas da mesma forma, utilizando-se ponto e vírgula para separar as linhas:

```
A = [1 2 3 ; 4 5 6]
```

```
A =

     1     2     3
     4     5     6
```

Algumas operações podem ser feitas diretamente:

```
A-1
```

```
ans =

     0     1     2
     3     4     5
```

```
A*2
```

```
ans =

     2     4     6
     8    10    12
```

Algumas operações, porém, devem ser feitas elemento a elemento, no caso de querer elevar todos os elementos de A ao quadrado, por exemplo. É importante sempre observar as dimensões das matrizes, pois o MATLAB é muito restrito e requer coerência total em qualquer conta matricial ou vetorial.

Para operações elemento a elemento basta utilizar o operador ".", conforme abaixo:

```
A.^2
```

```
ans =

     1     4     9
    16    25    36
```

Tente escrever e executar o seguinte Script, clicando em Run ou apertando F5 no Editor:

```
clc
clear all
A = [1 2 3 ; 4 5 6]
A^2
```

Como podemos ver o próprio MATLAB sugeriu a forma correta. Vamos fazer mais operações com matrizes, observe os resultados:

```
clear all

A = [1 2 1 ; 1 1 0; 1 1 1]
```

```
A =

     1     2     1
     1     1     0
     1     1     1
```

```
B = [1 2 3]
```

```
B =

     1     2     3
```

Como a matiz é quadrada podemos eleva-la a potências:

```
A = A^3
```

```
A =

    11    15     6
     6     8     3
     9    12     5
```

Para transpor basta utilizar o plick:

```
B'
```

```
ans =

     1
     2
     3
```

Matrizes de dimensões compatíveis podem ser multiplicadas (observe que B está transposto):

```
C = A * B'
```

```
C =

    59
    31
    48
```

Soma e inversão também são feitas diretamente:

```
C + B'*(-5)
```

```
ans =

    54
    21
    33
```

```
A^-1
```

```
ans =

   -4.0000    3.0000    3.0000
    3.0000   -1.0000   -3.0000
    0.0000   -3.0000    2.0000
```

Acessar elementos dos vetores é feito por meio de parêntesis, na forma (linha,coluna)

```
A(1,1)
```

```
ans =

    11
```

```
A(3,2)
```

```
ans =

    12
```

Vetores também podem ser declarados automaticamente pelo comando:

```
x = a:b:c
```

onde a é o valor do primeiro elemento, b é o passo e c o valor final. Por exemplo:

```
x = 0:0.1:10;
```

Gera um vetor começando em 0, incrementando de 0.1 em 0.1 e terminando em 10. Este vetor irá possuir 101 elementos (`(10*10)+1`) por causa do zero. Podemos conferir verificando o comprimento:

```
length(x)
```

```
ans =

   101
```

Vetores declarados desta forma são úteis para fazer gráficos com muitos pontos, como veremos a seguir.

5 Plotagem de Gráficos

MATLAB fornece uma gama enorme de métodos de plotagem de dados, neste documento será mostrado apenas o básico, porém é fortemente recomendado que procure na documentação do MATLAB os diferentes métodos de plotagem que podem ser úteis dependendo do seu propósito.

```
clc
clear all
```

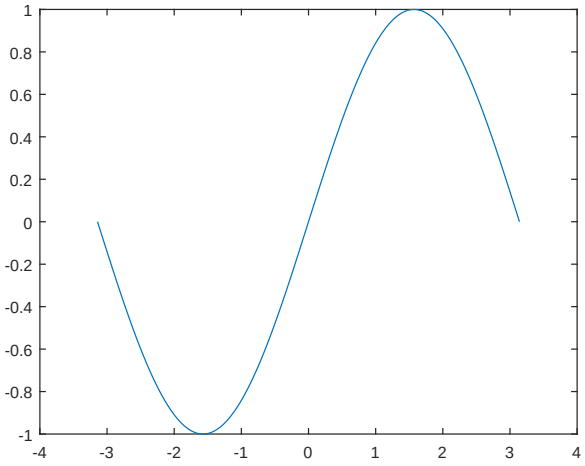
Primeiro criamos os dados:

```
x = -pi:pi/100:pi;
y = sin(x);
```

Nesta associação acima, y automaticamente vira um vetor, avaliando x elemento a elemento. Como x vai de $-\pi$ a $-\pi$, temos um período completo do seno armazenado em y, agora podemos plotar.

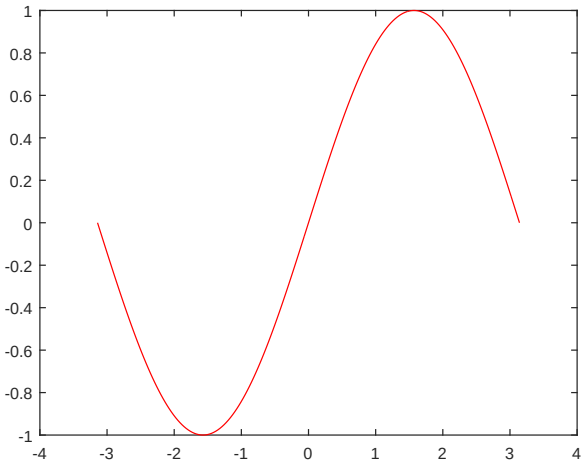
Sempre que vamos plotar um gráfico é necessário abrir uma nova janela com o comando `figure`, seguido de `plot(X,Y)`, onde X são os dados do eixo X e Y os do eixo Y. É importante que tenham as mesmas dimensões:

```
figure
plot(x,y);
```



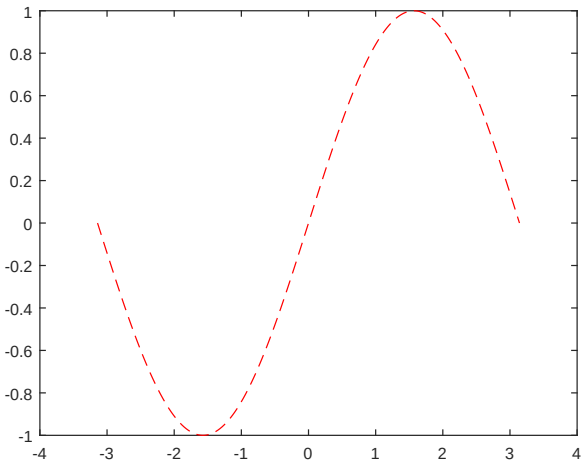
Pode-se mudar a cor conforme abaixo:

```
figure
plot(x,y,'color','red');
```



O estilo da linha também pode ser modificado, sempre logo após os dados:

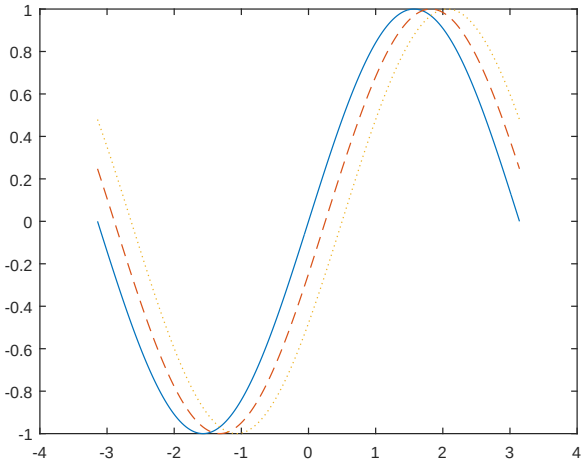
```
figure
plot(x,y,'--','color','red');
```



Plotar vários gráficos na mesma janela é simples caso tenham o mesmo tamanho, e podemos associar estilos diferentes a cada um simplesmente colocando os dados x e y ordenados corretamente:

```
x = -pi:pi/100:pi;
y1 = sin(x);
y2 = sin(x-0.25);
y3 = sin(x-0.5);
```

```
figure
plot(x,y1,x,y2,'--',x,y3,':')
```

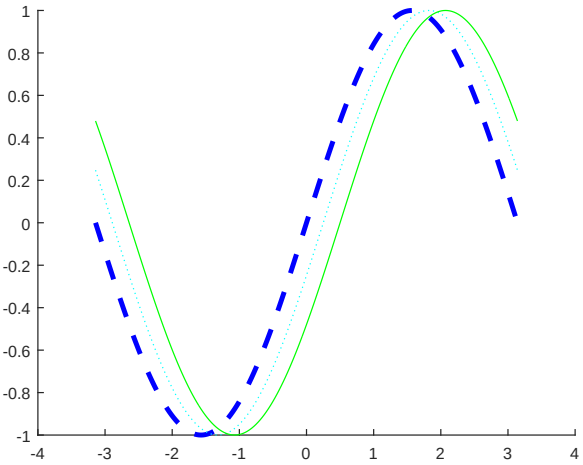


Para maior controle sobre os gráficos é comum utilizar o comando:

```
figure
hold on
.
.
.
hold off
```

Onde preenchemos os gráficos a serem plotados entre os holds, por exemplo:

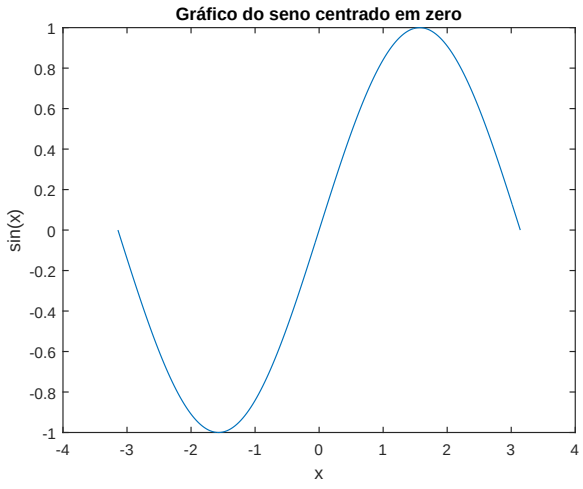
```
figure
hold on
plot(x,y1,'--','LineWidth',3,'color','blue');
plot(x,y2,':','color','cyan');
plot(x,y3,'color','green');
hold off
```



No exemplo acima a opção LineWidth seguida de um número altera a espessura da linha.

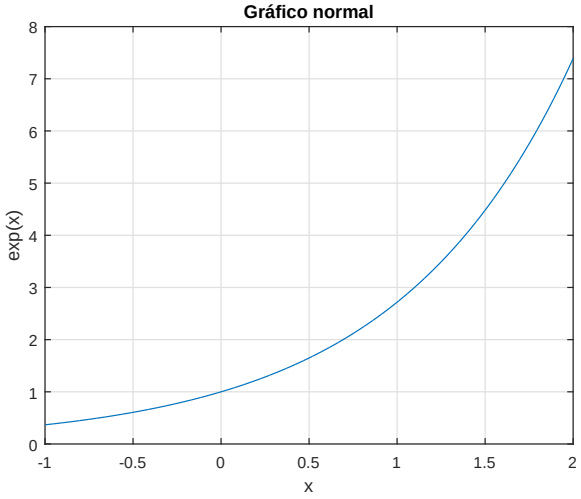
Adicionar título e legenda dos eixos é muito simples:

```
figure
plot(x,y)
title('Gráfico do seno centrado em zero')
xlabel('x')
ylabel('sin(x)')
```

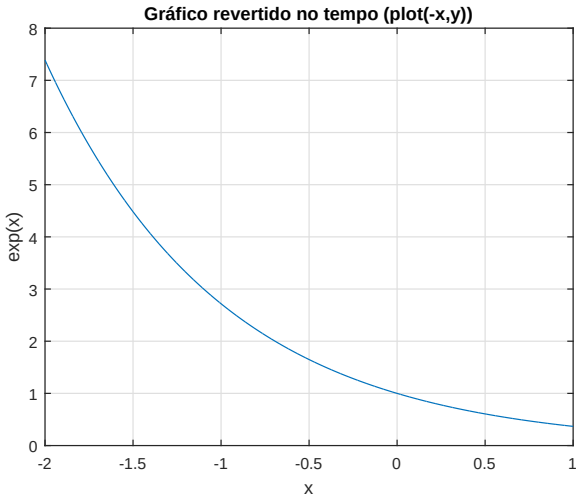


Fazer inversões no tempo, translações e outros é feito manipulando-se os vetores de dados diretamente:

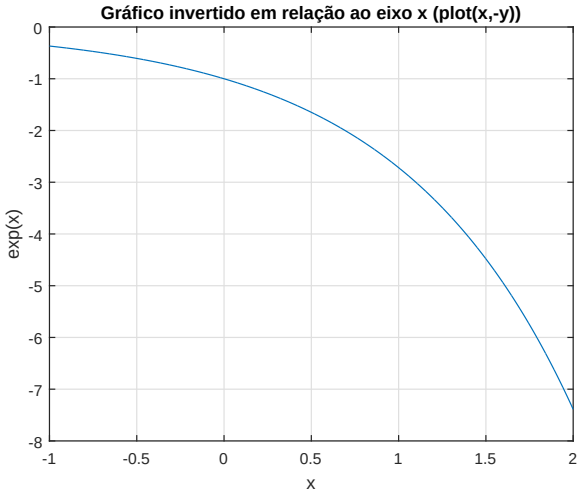
```
x = -1:0.001:2;
y = exp(x);

figure
plot(x,y)
grid on %este comando adiciona um quadriculado ao gráfico
title('Gráfico normal')
xlabel('x')
ylabel('exp(x)')
```




```
figure
plot(-x,y)
grid on
title('Gráfico revertido no tempo (plot(-x,y))')
xlabel('x')
ylabel('exp(x)')
```

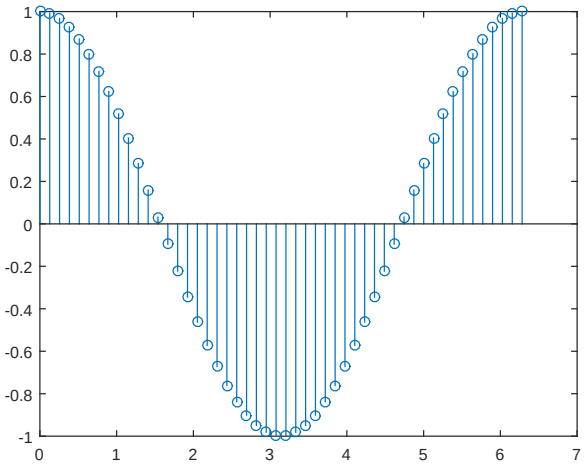



```
figure
plot(x,-y)
grid on
title('Gráfico invertido em relação ao eixo x (plot(x,-y))')
xlabel('x')
ylabel('exp(x)')
```



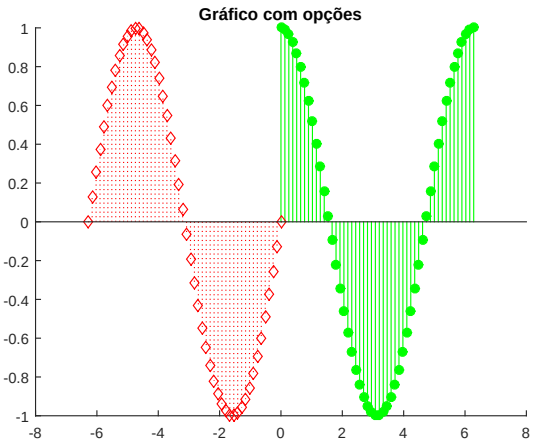
Todos os gráficos acima foram feitos para tempo contínuo, porém quando vamos plotar sinais discretos é interessante utilizar o comando `stem(X,Y)` em vez do plot. Na sequência abaixo também está sendo introduzido o comando linspace, que pode substituir o outro método de geração de vetores (observe a transposição):

```
figure
X = linspace(0,2*pi,50)';
Y = cos(X);
stem(X,Y)
```



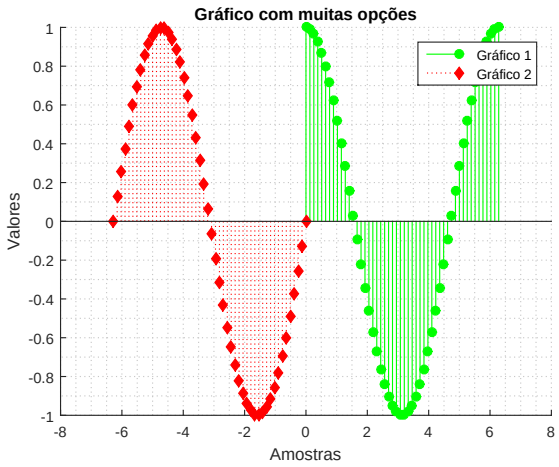
Gráficos discretos estão sujeitos a diversas opções também:

```
figure
X1 = linspace(0,2*pi,50)';
X2 = linspace(-2*pi,0,50)';
Y1 = cos(X1);
Y2 = sin(X2);
hold on
stem(X1,Y1,'filled','green')
stem(X2,Y2,':diamondr')
title('Gráfico com opções')
hold off
```



Para adicionar legendas dinâmicas podemos usar o seguinte:

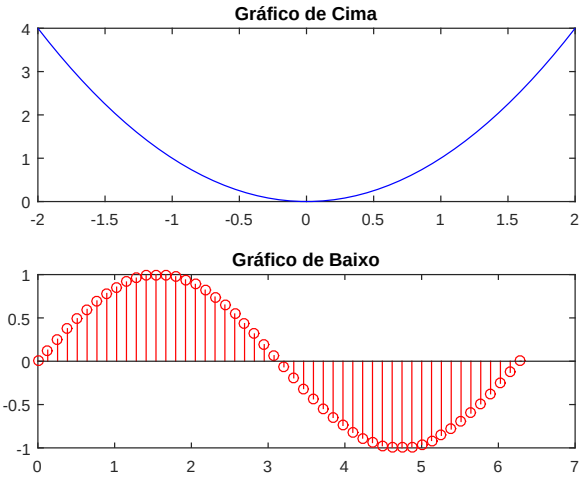
```
figure
hold on
grid minor
stem(X1,Y1,'filled','green','DisplayName','Gráfico 1')
stem(X2,Y2,'filled',':diamondr','DisplayName','Gráfico 2')
title('Gráfico com muitas opções')
xlabel('Amostras')
ylabel('Valores')
legend('-DynamicLegend')
hold off
```



Legendas dinâmicas carregam as informações do gráfico, facilitando muito a interpretação e dando maior poder de edição ao programador, já que estão associadas individualmente.

É útil colocar dois gráficos diferentes na mesma janela, por exemplo:

```
X = linspace(-2,2,50)';
Y = X.^2;

figure
subplot(2,1,1)
plot(X,Y,'b')
title('Gráfico de Cima')
subplot(2,1,2)
stem(X1,Y2,'r')
title('Gráfico de Baixo')
```



Esta ferramenta nos permite colocar gráficos completamente diferentes em todos os sentidos próximos para comparação. Para utilizar basta chamar o comando `subplot(a,b,c)` antes de cada comando de plotagem, seja plot ou stem, onde a e b geram um grid de a linhas e b colunas, e c diz em que posição deste grid o gráfico irá ficar.

O MATLAB também pode plotar funções em 3D e com animação, porém como fogem do escopo da matéria não será demonstrado aqui.

Parte II
Sinais e Sistemas

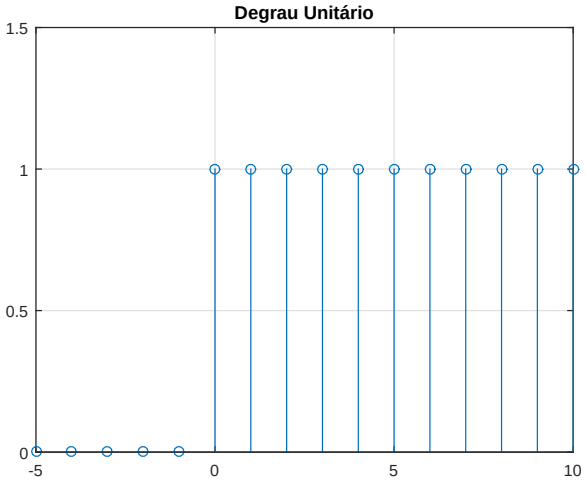
6 Geração de Sinais

Gerar sinais no MATLAB pode ser feito desde a forma mais simples até as mais automatizadas e complexas, vamos começar por algo simples: gerar um degrau unitário no tempo discreto. Conhecendo as definições das notas de aula é simples traduzir as condições num loop for/if, similar as linguagens em C:

```
tempoinicial = -5;
tempofinal = 10;
amostras = tempofinal - tempoinicial;
t = linspace(tempoinicial,tempofinal,amostras+1); %geramos nosso vetor tempo
for i = 1:length(t)
    if(t(i)>=0)
        u(i) = 1;
    else
        u(i) = 0;
    end
end
```

Podemos plotar o resultado:

```
figure
stem(t,u)
grid on
title('Degrau Unitário'),axis([tempoinicial tempofinal 0 1.5])
```

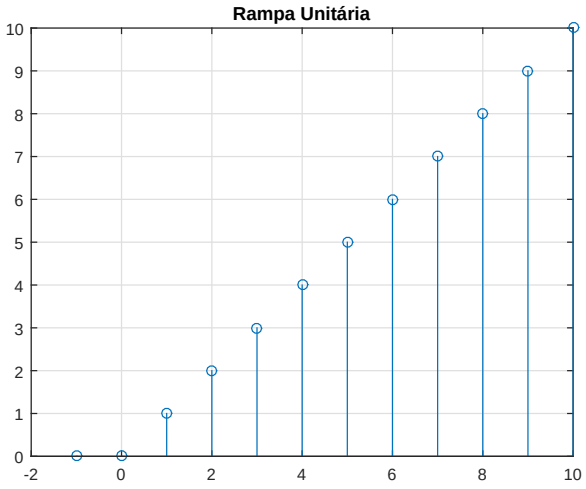


Modificar para gerar uma rampa unitária é simples da mesma forma:

```
clear all
tempoinicial = -1;
tempofinal = 10;
amostras = tempofinal - tempoinicial;
t = linspace(tempoinicial,tempofinal,amostras+1); %geramos nosso vetor tempo

alpha = 1; %inclinação da rampa

u = alpha*t;

for i = 1:length(u)
    if(u(i)<=0)
        u(i) = 0;
    end
end
```

Plotando:

```
figure
stem(t,u)
grid on
title('Rampa Unitária');
```



Para gerar a tempo continuo basta aumentar o numero de amostras para um número razoável e utilizar o plot:

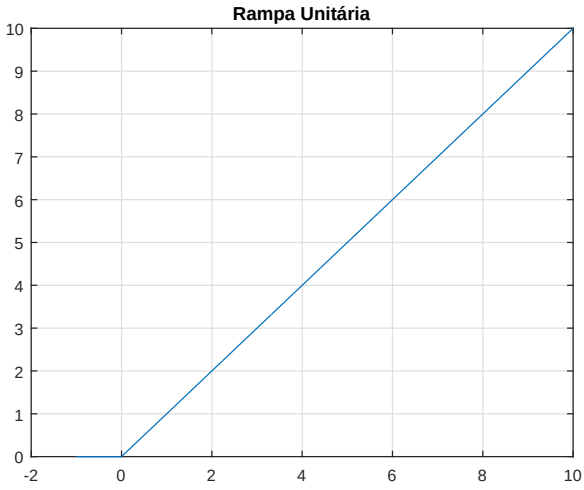
```
clear all
tempoinicial = -1;
tempofinal = 10;
amostras = tempofinal - tempoinicial;
t = linspace(tempoinicial,tempofinal,500); %geramos nosso vetor tempo

alpha = 1; %inclinação da rampa

u = alpha*t;
```

No loop abaixo percorremos o vetor da rampa procurando valores negativos (tempo negativo) e os retirando. Essa condição pode ser ajustada para o sinal que se quer gerar.

```
for i = 1:length(u)
    if(u(i)<=0)
        u(i) = 0;
    end
end
```

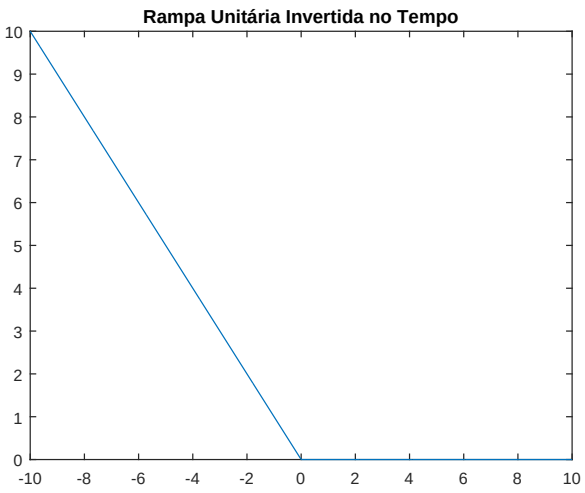
Plotando:

```
figure
plot(t,u)
grid on
title('Rampa Unitária');
```



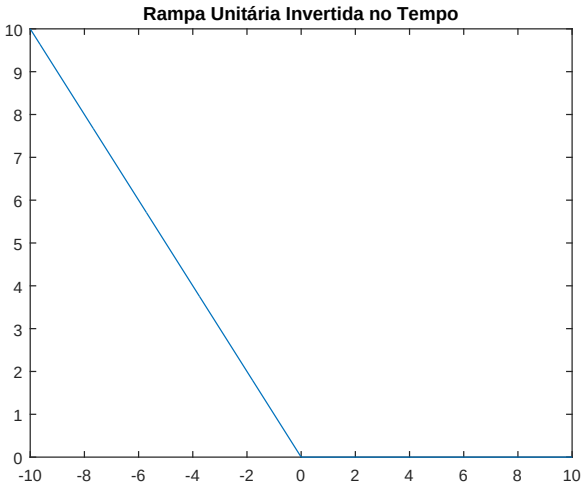
Inverter no tempo é feito da mesma forma, basta plotar com `-t` ou multiplicar `t` por `-1`:

```
clear all
tempoinicial = -10;
tempofinal = 10;
amostras = tempofinal - tempoinicial;
t = linspace(tempoinicial,tempofinal,500);
tinv = -t; %invertemos t

alpha = 1; %inclinação da rampa

u = alpha*t;

for i = 1:length(u)
    if(u(i)<=0)
        u(i) = 0;
    end
end

figure
plot(tinv,u)
title('Rampa Unitária Invertida no Tempo');
```



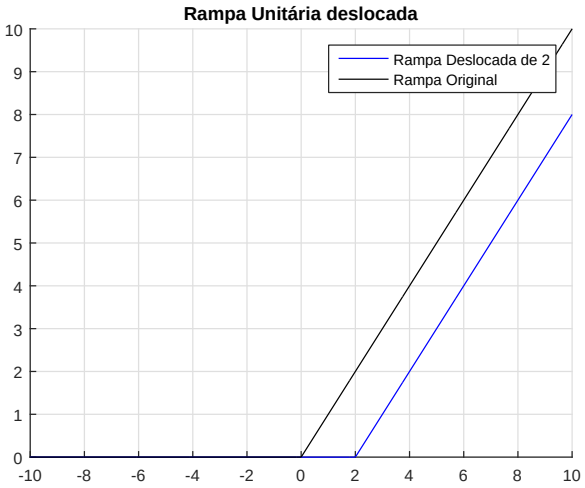
Análogamente (note o sinal de menos):

```
figure
plot(-t,u)
title('Rampa Unitária Invertida no Tempo');
```



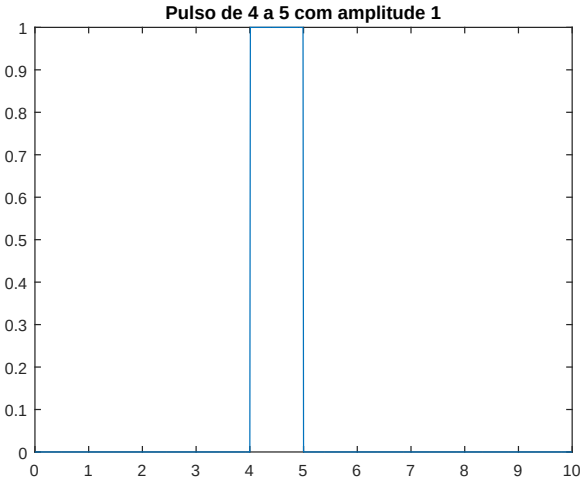
Para translações temporais basta manipular a função:

```
ud = alpha*(t-2);
u = alpha*t;

for i = 1:length(ud)
    if(ud(i)<=0)
        ud(i) = 0;
    end
    if (u(i)<=0)
        u(i) = 0;
    end
end

figure
hold on
grid on
plot(t,ud,'b','DisplayName','Rampa Deslocada de 2')
plot(t,u,'black','DisplayName','Rampa Original')
title('Rampa Unitária deslocada');
legend('-DynamicLegend');
hold off
```



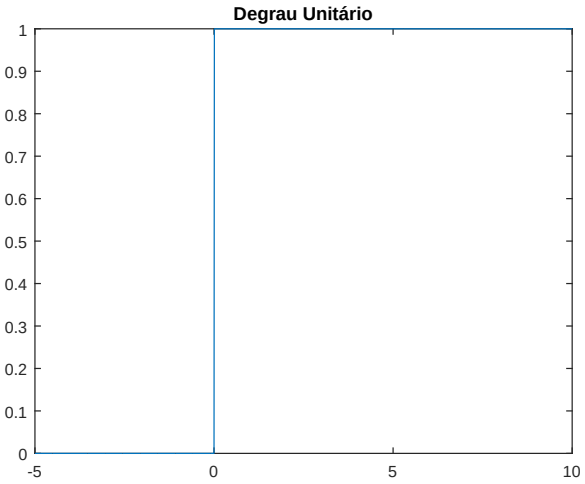
Gerar um pulso pode ser feito desta forma:

```
amp = 1;
t = 0:0.01:10;
u = zeros(1,length(t));
u(t>4 & t<5) = amp;
figure
plot(t,u)
title('Pulso de 4 a 5 com amplitude 1');
```



O trecho `u(t>4 & t<5) = 1;` simplesmente se traduz para se $4 > t > 5$, associar 1 (amplitude) ao sinal. A mesma estratégia pode ser utilizada para gerar um degrau de forma mais simples:

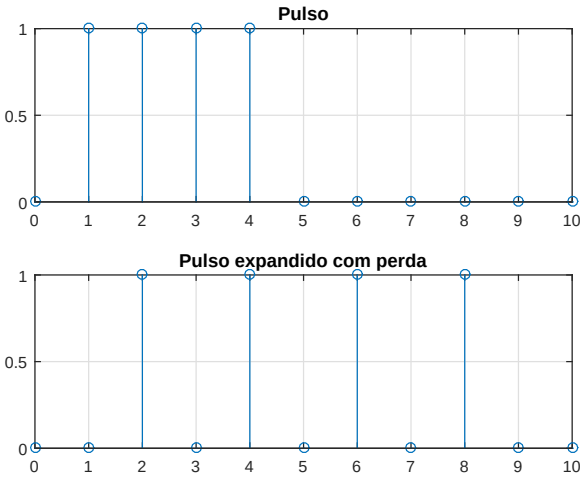
```
amp = 1;
t = -5:0.01:10;
u = zeros(1,length(t));
u(t>0 & t<=10) = amp;
figure
plot(t,u)
title('Degrau Unitário');
```



A função `zeros(...)` simplesmente cria um array de zeros. Leia a documentação para entende-la utilizando `help zeros` na Command Window

Expansão no tempo de sinais contínuos e discretos pode ser feita apenas multiplicando as variáveis por constantes, no entanto para gerar sinais expandidos com perda de informação podemos usar a função `upsample(x,n)` que expande as amostras por um fator inteiro n:

```
amp = 1;
ups = 2;
t = 0:1:10;
u = zeros(1,length(t));
u(t>=1 & t<=4) = amp;
us = upsample(u,ups);
ts = 0:(length(us)-1);
figure
subplot(2,1,1)
stem(t,u),grid on,axis([0 10 0 1])
title('Pulso');
subplot(2,1,2)
stem(ts,us),grid on,axis([0 10 0 1])
title('Pulso expandido com perda');
```



Manipulações com loops de for podem gerar qualquer sinal, basta criatividade e paciência. Felizmente o MATLAB fornece ferramentas prontas para geração dos sinais mais comuns que podem ser necessários.

A função `gensig` é chamada da seguinte forma : `[u,t] = gensig(type,tau,Tf,Ts)` onde type pode ser `'sin'` para seno, `'square'` para onda quadrada e `'pulse'` para pulsos periódicos, tau é o período do sinal, Tf é o tempo total do sinal e Ts o período de amostragem (tempo entre as amostras).

Exemplo: gerar uma senóide de 1 s, amostrada a 1 kHz com frequência de 10 Hz.

Começamos definindo os parâmetros:

```
close all
clear all
sigfreq = 10;
tau = 1/sigfreq;
sampfreq = 1000;
Ts = 1/sampfreq;
Tf = 1;
```

Chama-se a gensig e plotamos o gráfico:

```
[u,t] = gensig('sin',tau,Tf,Ts);
```

Em 1 segundo amostrando a 1 kHz esperamos 1000 amostras, contando com o zero espera-se um vetor de 1001 posições:

```
length(u)
```

```
ans =

        1001
```

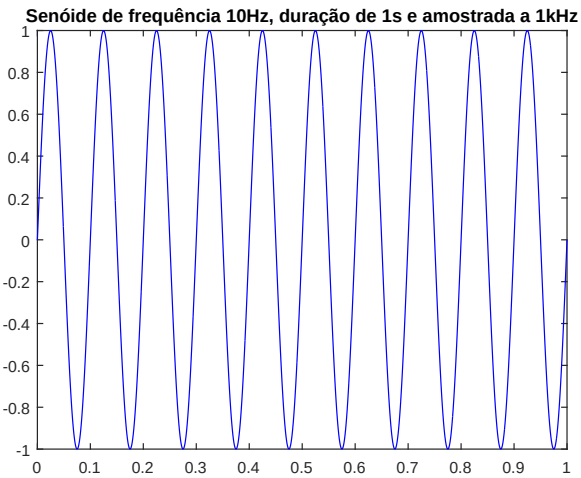
consequentemente o vetor t também tem a mesma dimensão:

```
length(t)
```

```
ans =

        1001
```

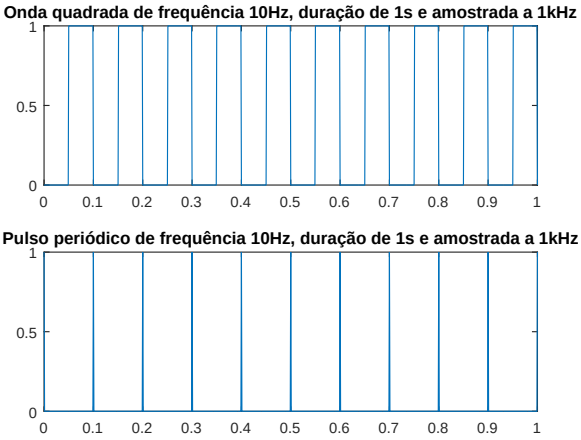
Esta ferramenta é útil pois não precisamos nos preocupar com a geração de vetores nas dimensões certas, ele faz tudo automaticamente. Tendo isto podemos plotar o resultado:

```
figure
plot(t,u,'b')
tt = sprintf(['Senóide de frequência %dHz, duração de %ds'...
    ' e amostrada a %dkHz'],sigfreq,Tf,sampfreq/1000);
title(tt)
```



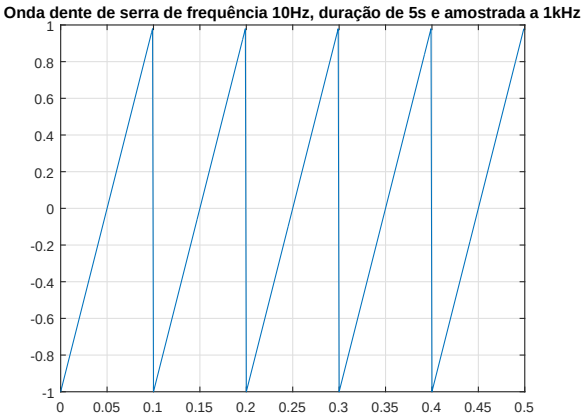
Onda quadrada e pulsos também estão disponíveis:

```
[u1,t] = gensig('square',tau,Tf,Ts);
[u2,t] = gensig('pulse',tau,Tf,Ts);

figure
subplot(2,1,1)
plot(t,u1)
tt = sprintf(['Onda quadrada de frequência %dHz, duração de %ds'...
    ' e amostrada a %dkHz'],sigfreq,Tf,sampfreq/1000);
title(tt)
subplot(2,1,2)
plot(t,u2)
tt = sprintf(['Pulso periódico de frequência %dHz, duração de %ds'...
    ' e amostrada a %dkHz'],sigfreq,Tf,sampfreq/1000);
title(tt)
```



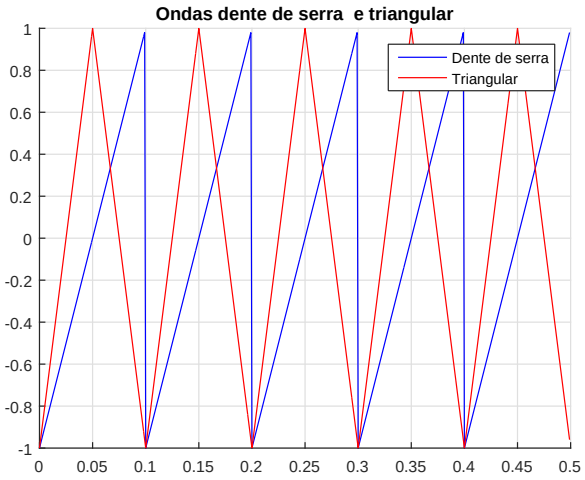
Onda dente de serra (10 Hz com 1 kHz de amostragem) pode ser feita com:

```
Tf = 5;
T = Tf*(1/sigfreq);
dt = 1/sampfreq;
t = 0:dt:T-dt;
x = sawtooth(2*pi*sigfreq*t);

figure
plot(t,x)
tt = sprintf(['Onda dente de serra de frequência %dHz, duração de %ds'...
    ' e amostrada a %dkHz'],sigfreq,Tf,sampfreq/1000);
title(tt)
grid on
```



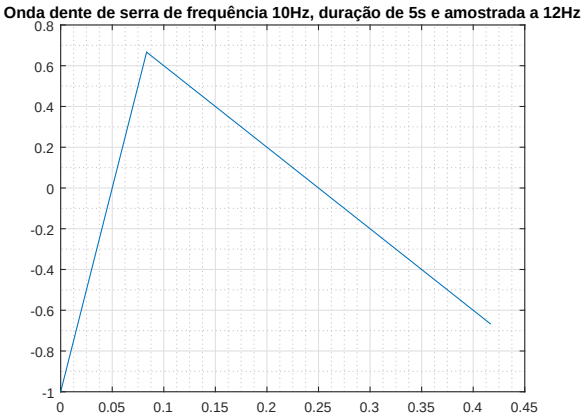
A função `sawtooth(t,w)` cria um sinal com período `2*pi`, logo manipulações algébricas precisam ser feitas no argumento para gerar um sinal apropriado. O parâmetro `w` é um escalar entre zero e um que diz o quão próximo do início do período o pico irá ficar, portanto para um sinal de onda triangular basta colocar em 0.5:

```
x2 = sawtooth(2*pi*sigfreq*t,0.5);

figure
hold on
plot(t,x,'color','blue','DisplayName','Dente de serra')
plot(t,x2,'color','red','DisplayName','Triangular')
tt = sprintf('Ondas dente de serra  e triangular');
title(tt)
legend('-DynamicLegend')
hold off
grid on
```



Você pode estar se perguntando a importância da taxa de amostragem em todos estes comandos. Como será visto adiante, na Transformada Z, a taxa de amostragem de um sinal muda completamente sua natureza, portanto se deseja utilizar estes sinais para qualquer tipo de conta ou simulação é de extrema importância que use uma taxa de amostragem correta, no exemplo abaixo vamos mudar a taxa de amostragem da onda dente de serra acima para apenas 12Hz:

```
sampfreq = 12;
Tf = 5;
T = Tf*(1/sigfreq);
dt = 1/sampfreq;
t = 0:dt:T-dt;
x = sawtooth(2*pi*sigfreq*t);

figure
plot(t,x)
tt = sprintf(['Onda dente de serra de frequência %dHz, duração de %ds'...
    ' e amostrada a %dHz'],sigfreq,Tf,sampfreq);
title(tt)
grid on
grid minor
```



Como podemos ver a função claramente não representa uma dente de serra, embora tenha sido amostrada do mesmo sinal.

7 Energia e potência de sinais

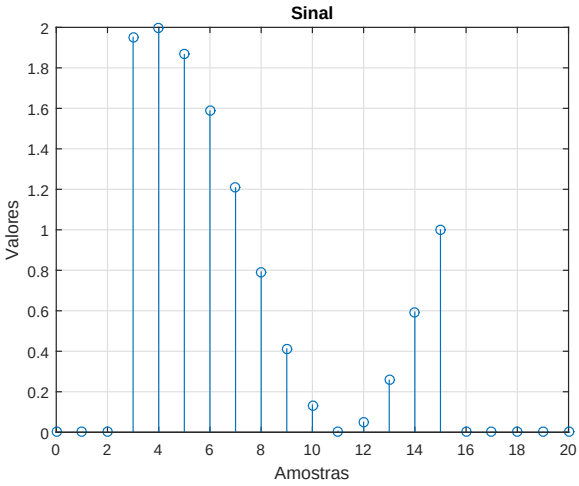
Calcular a energia e potência de um sinal é uma aplicação direta de fórmula, que são as abaixo para tempo discreto:

$$E_x = \sum_{n=-\infty}^{\infty} |x_n|^2 \qquad\qquad P_x = \lim_{N\to\infty} \frac{1}{2N+1} \sum_{n=-N}^{N} |x_n|^2$$

Fazer a energia pela definição é direto, vamos criar um sinal discreto e calcular:

```
t=0:1:20;
y = zeros(length(t),1)';y(4:16)=1;
x = y.*(sin(2*pi*t/15)+1);

figure
stem(t,x),grid on,title('Sinal')
xlabel('Amostras'),ylabel('Valores')

energia = sum(abs(x).^2)
```

```
energia =

   17.4825
```

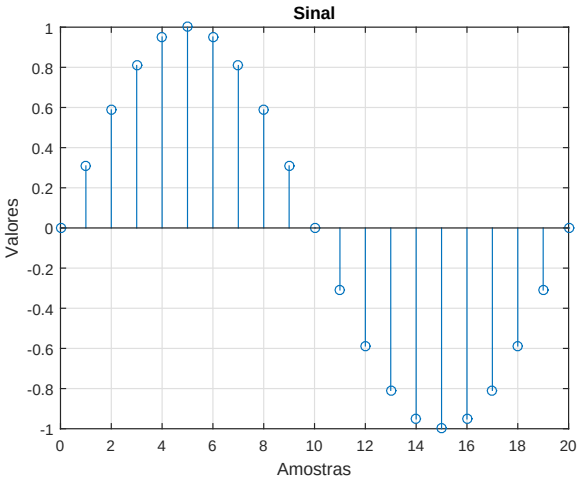


Neste caso a energia é finita, portanto a potência é zero. Podemos criar um sinal periódico para calcular a potência, uma vez que isso garante que P_x será finito. No caso de um sinal periódico basta considerar um período e fazer a soma neste, a fórmula também se simplifica para:

$$P_x = \frac{1}{N} \sum_{n=0}^{N-1} |x_n|^2$$

```
t = 0:20;
x = sin((pi*t)/10);
figure
stem(t,x),grid on,title('Sinal')
xlabel('Amostras'),ylabel('Valores')

potencia = sum(abs(x).^2)/length(x)
```

```
potencia =

    0.4762
```



O motivo pela qual o valor não é 0.5 exatamente é devido a pequenos erros numéricos. Se aumentarmos o número de amostras vemos que o erro diminui:

```
t = 0:0.001:20;
x = sin((pi*t)/10);
potencia = sum(abs(x).^2)/length(x)
```

```
potencia =

    0.5000
```

Para sinais contínuos fica ainda mais fácil, pois podemos aplicas as formulas diretamente:

$$E_x = \int_{-\infty}^{\infty} |x(t)|^2 dt \qquad\qquad P_x = \frac{1}{2T} \lim_{T\to\infty} \int_{-\infty}^{\infty} |x(t)|^2 dt$$

```
tic
syms t T

y = sin(t);

energia=int(y,t,-inf,inf)
potencia=limit((1/T)*int(y^2,-T/2,T/2),T,inf)
temposym = toc
```

```
energia =

NaN


potencia =

1/2


temposym =

    3.6608
```

Como agora estamos computando com variáveis simbólicas, o resultado é exato. O único jeito de verificar que a energia é infinita em sinais periódicos infinitos é por meio de cálculos simbólicos. Como este método é computacionalmente caro, as vezes é mais interessante utilizar uma pilha grande de pontos flutuantes fazendo um sinal continuo ser amostrado muito rápido e por muito tempo. A tendência dos resultados poderá dar conclusões se o usuário utilizar bom senso. Vamos resolver a função anterior deste outro método e comparar quanto tempo ganhamos em relação ao tempo decorrido durante a operação anterior.

```
tic
Tf = 10000;
t = 0:0.01:Tf;
y = sin(t);
energia = sum(abs(y).^2)
potencia = sum(abs(y).^2)/length(y)
tempofloat = toc
```

```
energia =

   4.9999e+05


potencia =

    0.5000


tempofloat =

    0.0292
```

Podemos ver que a energia claramente explode, pode verificar começando com um Tf pequeno e aumentar gradualmente. Percebemos também que o tempo computacional do segundo método foi muito menor.

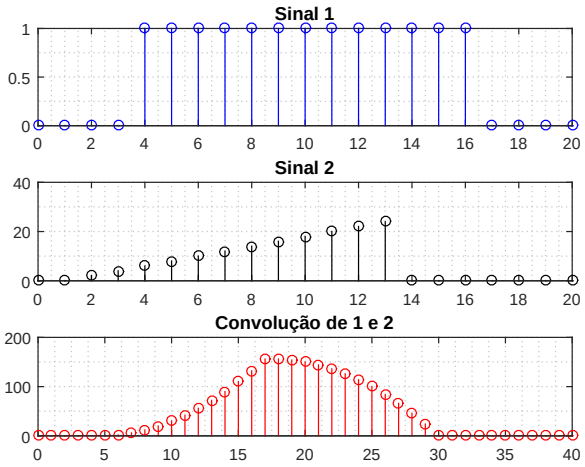
8 Convolução

A convolução, como já deve saber, é de extrema importância em processamento de sinais, por esse motivo o MATLAB já tem uma implementação da operação com o comando `conv(X,Y)`, onde X e Y são seus sinais de entrada, começamos pela geração dos sinais:

```
amp = 1;
t = 0:1:20;
u = zeros(1,length(t));
u(t>=4 & t<=16) = amp;
a = 2;

u2 = a*t - 2;

for i = 1:length(u2)
    if(u2(i)<=0)
        u2(i) = 0;
    end
    if(t(i)>=14) u2(i) = 0;
    end
end
```

Agora convoluímos e plotamos os resultado:

```
c = conv(u,u2);
tc = 0:(length(c)-1);
figure
subplot(3,1,1)
stem(t,u,'blue'),title('Sinal 1'),grid minor
subplot(3,1,2)
stem(t,u2,'black'),title('Sinal 2'),grid minor
subplot(3,1,3)
stem(tc,c,'red'),title('Convolução de 1 e 2'),grid minor
```

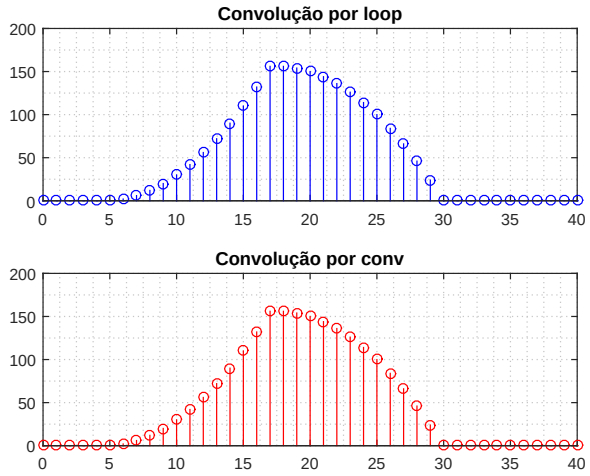


Também, caso seja necessário, podemos fazer a convolução por meio de um loop lógico, conforme abaixo:

```
L1 = length(u);
L2 = length(u2);
X=[u,zeros(1,L2)];
H=[u2,zeros(1,L1)];

for i=1:L2+L1-1
    C(i)=0;
    for j=1:L1
        if(i-j+1>0)
            C(i)=C(i)+X(j)*H(i-j+1);
        else
        end
    end

end
```

Plotando vemos que o resultado é idêntico:

```
tC = 0:(length(C)-1);

figure
subplot(2,1,1)
stem(tC,C,'b'),title('Convolução por loop'),grid minor
subplot(2,1,2)
stem(tc,c,'r'),title('Convolução por conv'),grid minor
```



9 Transformada Z

Transformada Z é muito fácil de ser calculada no MATLAB. A forma mais fácil envolve utilizar variáveis simbólicas, no caso se quisermos a Transformada Z de $a+\exp(n)$, fazemos a, n e z simbólicos e utilizamos o comando `zatrans(a,b,c)`, onde `a` é a expressão, `b` a variável no tempo e `c` a variável z.

```
clear all
close all

syms n z a
Y = ztrans(a+exp(n),n,z) %levar a+exp(n) de n  para z
```

```
Y =

z/(z - exp(1)) + (a*z)/(z - 1)
```

A transformada inversa pode ser calculada também por:

```
iztrans(Y,z,n) %levar Y de z para n
```

```
ans =

exp(1)*(exp(-1)*exp(n) - exp(-1)*kroneckerDelta(n, 0))
 + a*kroneckerDelta(n, 0) + kroneckerDelta(n, 0) -
a*(kroneckerDelta(n, 0) - 1)
```

A resposta nem sempre é a esperada, pois os métodos utilizados pelo MATLAB diferem muito. Usualmente não é recomendado confiar em transformadas inversas desta forma, e deve ser utilizado apenas para conferir suas respostas.

Normalmente é mais interessante utilizar o MATLAB para analisar comportamento de sistemas. Pode-se, por exemplo, criar uma função de transferência declarando o numerador e denominador com tempo de amostragem desejado:

```
num = [ 1 0 1 ];
den = [ 1  -1.85  0.9 ];
TS = 1;
H = tf(num,den,TS)

H =

       z^2 + 1
  ------------------
  z^2 - 1.85 z + 0.9

Sample time: 1 seconds
Discrete-time transfer function.
```

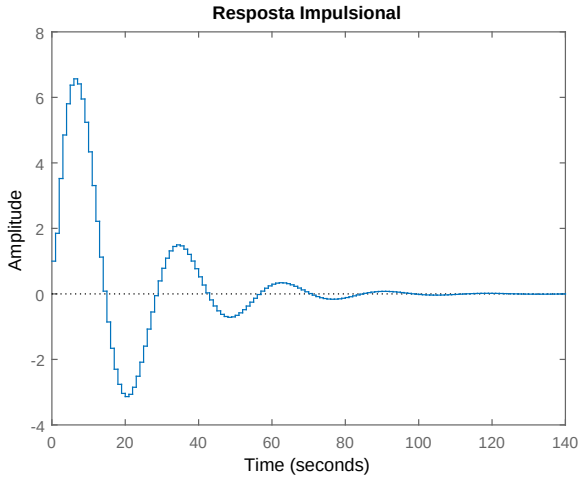
Ou alternadamente:

```
z = tf('z',1);
H = (z^2 + 1) / (z^2 - 1.85*z + 0.9)

H =

       z^2 + 1
  ------------------
  z^2 - 1.85 z + 0.9

Sample time: 1 seconds
Discrete-time transfer function.
```

num sempre armazena os coeficientes em ordem decrescente do numerador, sempre terminando em zero e contendo todos os coeficientes. No exemplo acima temos $1z^2 + 0z^1 + 1z^0$ no numerador. O mesmo vale para o denominador. TS é o tempo obrigatório de amostragem, normalmente empregado como 1 no curso.

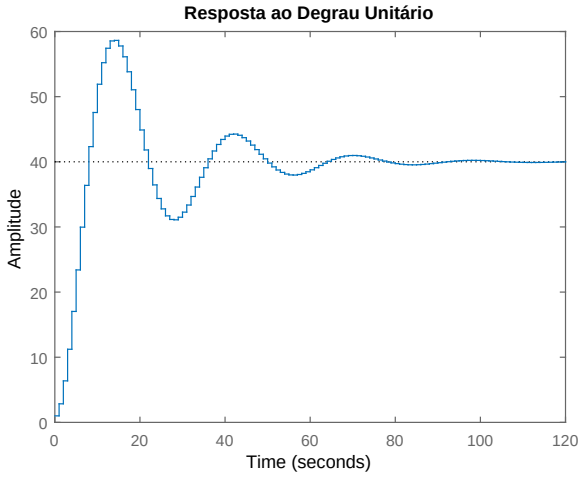
Podemos simular a resposta impulsional simplesmente com:

```
figure
impulse(H)
title('Resposta Impulsional')
```



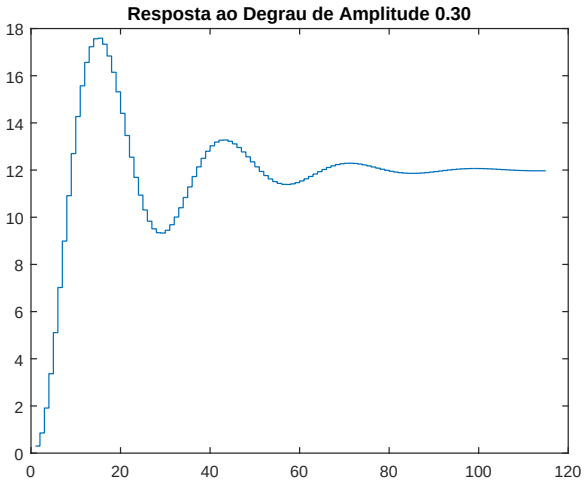
E resposta ao degrau com:

```
figure
step(H)
title('Resposta ao Degrau Unitário')
```



Como o sistema é linear, caso queiramos um degrau não unitário basta multiplicar a saída por uma constante, uma vez que step pode ser tratado como figura mas também retorna os valores da simulação:

```
amplitude = 0.3;

y = amplitude*step(H);

figure
stairs(y)
tt = sprintf('Resposta ao Degrau de Amplitude %0.2f',amplitude);
title(tt)
```



Como podemos ver em relação ao anterior, o valor final foi menor, como esperado.

Para fazer a resposta a rampa é necessário alguma manipulação, por exemplo, começamos criando um sistema:

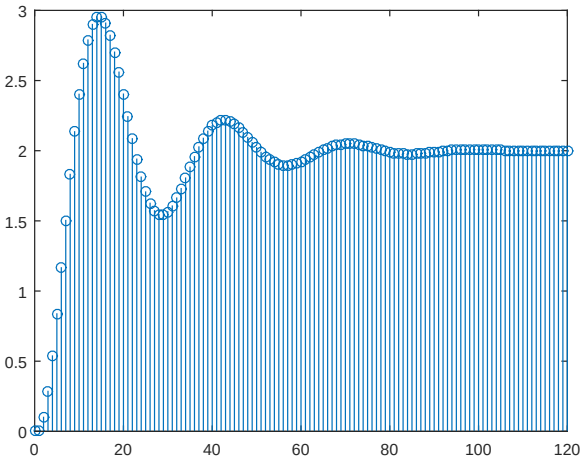
```
z = tf('z',1);
H = (0.05*(z - 1)) / (z^2 - 1.85*z + 0.9);
```

Em seguida declaramos o tempo de simulação

```
t=0:1:120; %120 amostras
alpha = 2; %inclinação da rampa
rampa = alpha*t;
```

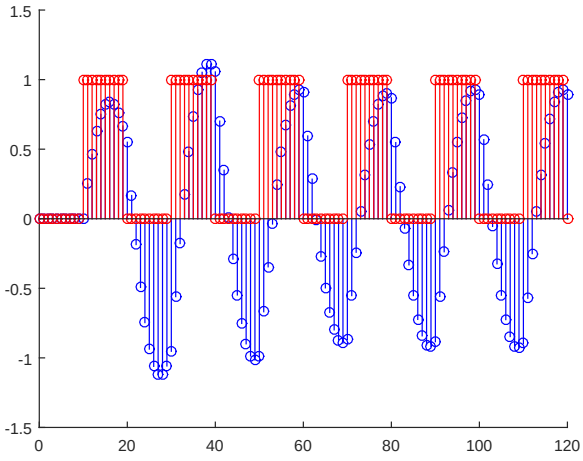
Agora utilizamos o comando `lsim(a,b,c)` onde `a` é o sistema, `b` a entrada e `c` o vetor tempo. `lsim` não plota nada, apenas gera dados, portanto plotamos os dados gerados em seguida:

```
[y,t]=lsim(H,rampa,t);
figure
stem(t,y)
```



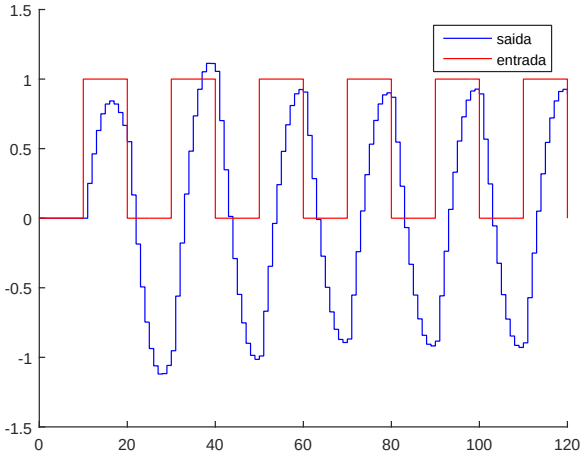
lsim pode simular qualquer tipo de entrada, por exemplo uma onda quadrada de período de 20 amostras, durando 100 amostras e de amplitude 5:

```
[u,t] = gensig('square',20,120,1);
[y,t] = lsim(H,5*u,t);
figure
hold on
stem(t,y,'b','DisplayName','saida')
stem(t,u,'r','DisplayName','entrada')
hold off
```



Como podemos ver, o gráfico stem dificulta muito a visualização, portanto é normal utilizar o gráfico stairs no lugar:

```
figure
hold on
stairs(t,y,'b','DisplayName','saida')
stairs(t,u,'r','DisplayName','entrada')
legend('-DynamicLegend')
hold off
```



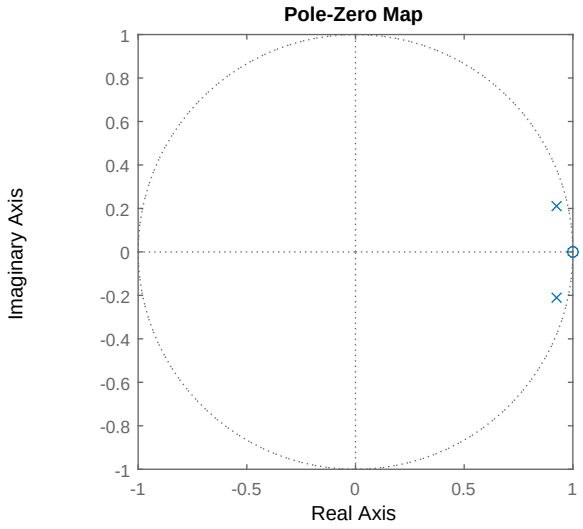
Podemos acessar os polos e zeros do sistema utilizando a função `pzmap(sys)`:

```
[p,z] = pzmap(H)
```

```
p =

   0.9250 + 0.2107i
   0.9250 - 0.2107i


z =

     1
```

Se não associarmos nada a `pzmap(sys)` um gráfico do plano de pólos e zeros é gerado:

```
figure
pzmap(H)
daspect([1 1 1]) %deixa os eixos iguais
```



Uma ferramenta adicional para analise de sistemas o `ltiview`, nesta janela existem diversas opções a serem exploradas de forma interativa, é fortemente recomendado que vá no help na janela para aprender mais.

```
ltiview('step',H,t);
```

10 Transformada de Laplace

A Transformada de Laplace em MATLAB é completamente análoga a Transformada Z, com a diferença que não precisamos passar a taxa de amostragem, já que estamos no tempo contínuo:

```
close all
clear all

syms a t s

F = laplace(sin(a*t),t,s)
```

```
F =

a/(a^2 + s^2)
```

O segundo argumento acima diz qual variável queremos usar na transformada, no caso é t. A inversa pode ser feita:

```
ilaplace(F,s,t) % levar F de s para t
```

```
ans =

sin(a*t)
```

Fazer Laplace inversa é mais seguro do que a Z inversa, porém ainda assim é boa prática utilizar apenas para conferir resultados.

Funções de Transferência são feitas da mesma forma:

```
num = [ 1/5 ];
den = [ 1  1/5  1/5 ];
H = tf(num,den)
```

```
H =

        0.2
  -----------------
  s^2 + 0.2 s + 0.2

Continuous-time transfer function.
```

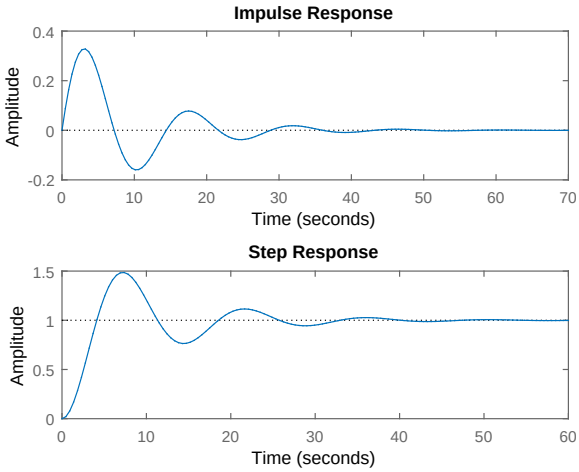
Ou alternadamente:

```
syms s
s = tf('s');
H = 1/5 / (s^2 + s/5 + 1/5)
```

```
H =

       1
  -------------
  5 s^2 + s + 1

Continuous-time transfer function.
```

Podemos simular utilizando todos os mesmos comandos anteriores para tempo discreto, o MATLAB se adapta:

```
figure
hold on
subplot(2,1,2)
step(H)
subplot(2,1,1)
impulse(H)
hold off
```



Polos e zeros também ficam disponíveis:

```
[p,z] = pzmap(H)
```

```
p =

  -0.1000 + 0.4359i
  -0.1000 - 0.4359i


z =

   Empty matrix: 0-by-1
```

No caso acima não temos zeros. Para passar de continuo para discreto e vice-versa pode-se usar o comando `c2d(TF,Ts,método)` para ir de continuo para discreto, onde TF é a função de transferência, Ts o período de amostragem e o método (normalmente utilizamos a *Zero-order hold* com a opção `'zoh'`) ou `d2c(TF,método)` para ir de discreto para contínuo.

```
Hd = c2d(H,1,'zoh')
Hc = d2c(Hd,'zoh')
```

```
Hd =

   0.09212 z + 0.08615
  ---------------------
  z^2 - 1.64 z + 0.8187

Sample time: 1 seconds
Discrete-time transfer function.


Hc =

         0.2
  -----------------
  s^2 + 0.2 s + 0.2

Continuous-time transfer function.
```

Estes resultados são confiáveis e podem ser usados com segurança. Recomenda-se pesquisar sobre transformações bilineares em MATLAB para melhor entendimento do processo.

11 Série de Fourier

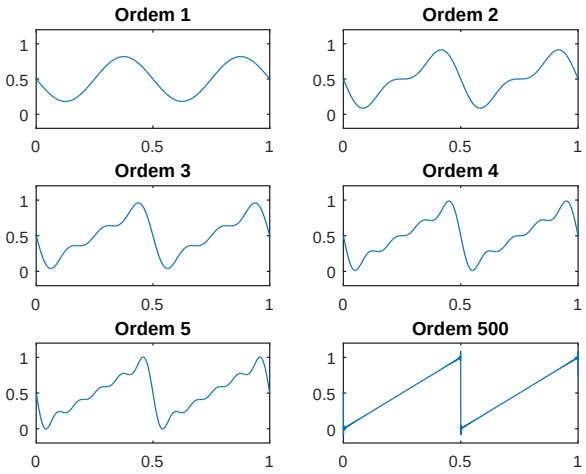
A série de fourier é uma aplicação direta de fórmulas e somatórios, serão apresentados dois loops exemplos sobre como fazer estas séries em código, porém em MATLAB é mais interessante ver as aplicações decorrentes do teorema em vez de puramente recriar sinais.

```
close all
```

Exemplo 1: onda dente de serra. Caso deseje rode este script para diferentes valores de N e descomente a linha `pause(0.1)`, verá uma animação da série se formando.

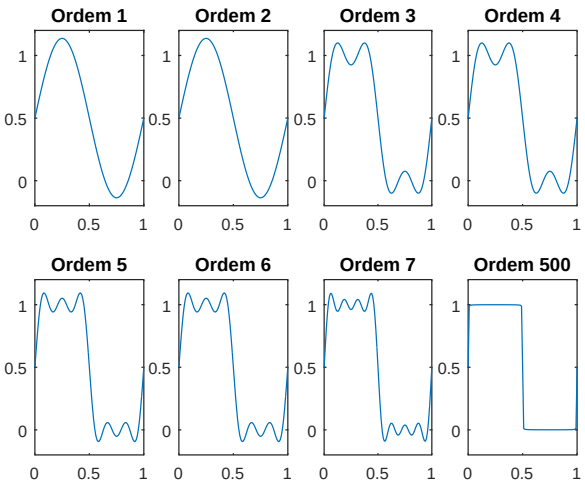
```
figure;
for i = 1:5
    N = i; %Ordem da série
    Fs = 8192; %Freq de amostragem
    t = linspace(0,1-1/8192,Fs);
```

```
    f = 2;
    amp = 1;
    subplot(3,2,N)
    h=plot(NaN,NaN);
    tt = sprintf('Ordem %d',N);
    title(tt),axis([0 1 -0.2 1.2])
    res = amp/2;
    for k=1:N
        res = res - (amp/(k*pi))*sin(2*pi*k*f*t);
        set(h,'XData',t,'YData',res);
        %pause(0.1)
    end
end

N = 500; %Ordem da série
Fs = 8192; %Freq de amostragem
t = linspace(0,1-1/8192,Fs);
f = 2;
amp = 1;
subplot(3,2,6)
h=plot(NaN,NaN);
tt = sprintf('Ordem %d',N);
title(tt),axis([0 1 -0.2 1.2])
res = amp/2;
for k=1:N
    res = res - (amp/(k*pi))*sin(2*pi*k*f*t);
    set(h,'XData',t,'YData',res);
    %pause(0.1)
end
```



Exemplo 2: Onda quadrada.

```
figure

for N = 1:7
    x = [0:100]/100;
    f = ones(1,101)*1/2;
    for i = 1:2:N
        a = 2/pi/i;
        f = f+ a*sin(2*pi*i*x);
    end
    subplot(2,4,N)
    plot(x,f)
    tt = sprintf('Ordem %d',N);
    title(tt),axis([0 1 -0.2 1.2])
end

N = 500;
x = [0:100]/100;
f = ones(1,101)*1/2;
for i = 1:2:N
    a = 2/pi/i;
    f = f+ a*sin(2*pi*i*x);
end
subplot(2,4,8)
plot(x,f)
tt = sprintf('Ordem %d',N);
title(tt),axis([0 1 -0.2 1.2])
```

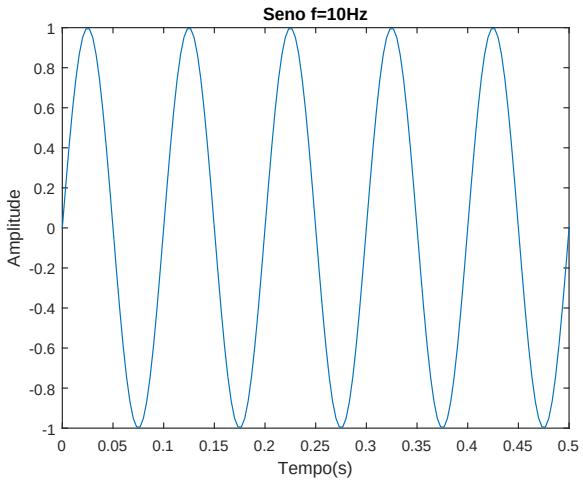


Neste exemplo vimos como apenas as harmônicas ímpares são coletadas. Nos exemplos de aplicação vamos ver como podemos usar a série de fourier para aproximar sinais e fazer análises.

12 Transformada de Fourier

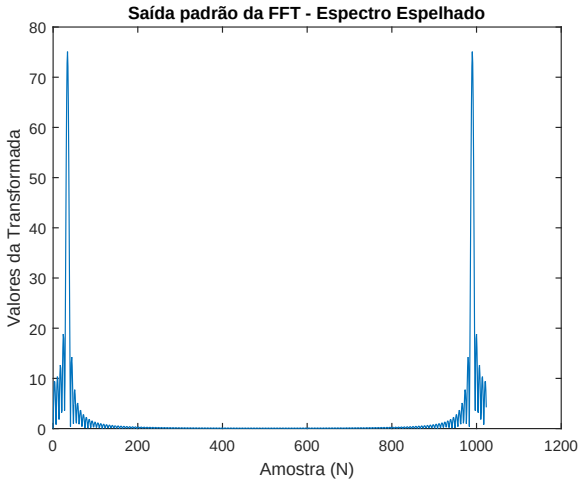
A transformada de Fourier serve para decompor o sinal nas frequências que o compõe, como isto é uma aplicação direta de fórmulas vamos ver com o MATLAB pode ser útil para análise de sinais com a TF. Primeiro geramos um sinal, no caso uma senóide com frequência de 60 Hz amostrada a 30 vezes a sua frequência e vamos utilizar 5 ciclos.

```
close all

f=10;
overSampRate=30;
fs=overSampRate*f;
nCyl = 5;
t=0:1/fs:nCyl*1/f;
x=sin(2*pi*f*t);

figure
plot(t,x);
title(['Seno f=', num2str(f), 'Hz']);
xlabel('Tempo(s)');
ylabel('Amplitude');
```



A função do MATLAB que faz a transformada de fourier é `fft(x,N)`, onde `x` é seu sinal e `N` a quantidade de amostras utilizadas na transformada. `N` deve ser pelo menos igual ao tamanho de `x`. Mais adiante vamos ver qual o papel de `N` na FFT (Fast Fourier Transform), por hora vamos gerar e analisar gráficos, começando com a saída padrão de uma FFT:

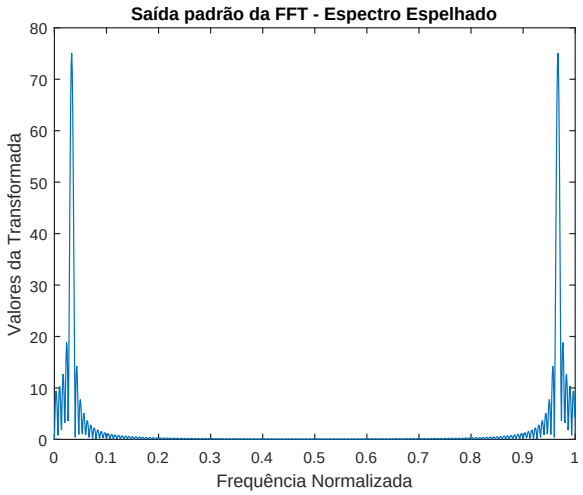
```
NFFT = 1024;
X = fft(x,NFFT);
nVals = 0:NFFT-1;
figure
plot(nVals,abs(X));
title('Saída padrão da FFT - Espectro Espelhado');
xlabel('Amostra (N)')
ylabel('Valores da Transformada');
```



Precisamos usar abs(x) para tomar a magnitude do sinal apenas, uma vez que temos parte real e imaginária. Você consegue entender o gráfico acima? Tudo que ele está nos dizendo são os valores da magnitude do sinal em cada amostra feita no espectro de frequência. É importante entender bem a parte teórica da matéria para utilizar a FFT corretamente.

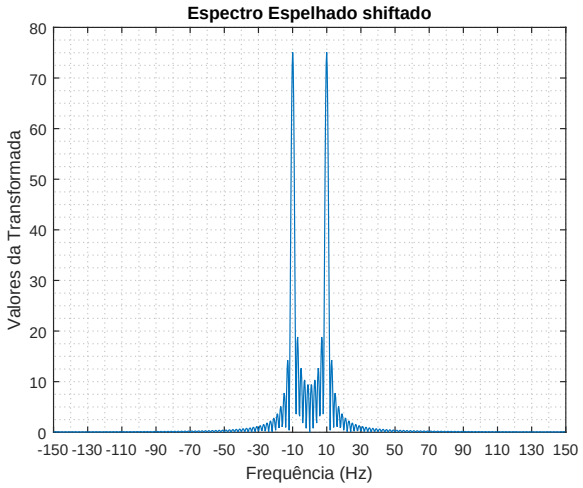
O gráfico anterior não nos serve de muita coisa para análise, pois o que nos interessa são as componentes espectrais do sinal, para isso precisamos "transformar"o eixo x em valores de frequência, podemos começar normalizando o vetor de amostras:

```
nVals = nVals/NFFT;
figure
plot(nVals,abs(X));
title('Saída padrão da FFT - Espectro Espelhado');
xlabel('Frequência Normalizada')
ylabel('Valores da Transformada');
```



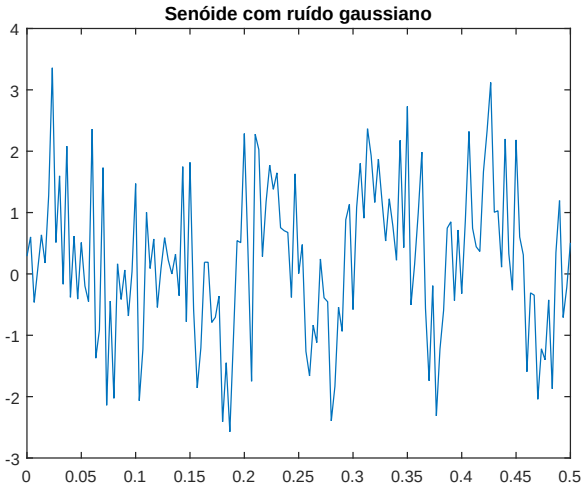
Como sabemos, no domínio da frequência, temos valores negativos e positivos, por isso pode estar estranho essa saída para alguns. Para resolver isso associamos valores negativos ao vetor de amostras e também utilizamos a função `fftshift(x)` para mover o zero em frequência para o centro do array:

```
fVals = fs*(-NFFT/2:NFFT/2-1)/NFFT;
X = fftshift(fft(x,NFFT));
figure
plot(fVals,abs(X));
set(gca,'xtick',[-150:20:150]) %melhora precisão de leitura
title('Espectro Espelhado shiftado');
xlabel('Frequência (Hz)')
ylabel('Valores da Transformada');
grid minor
```



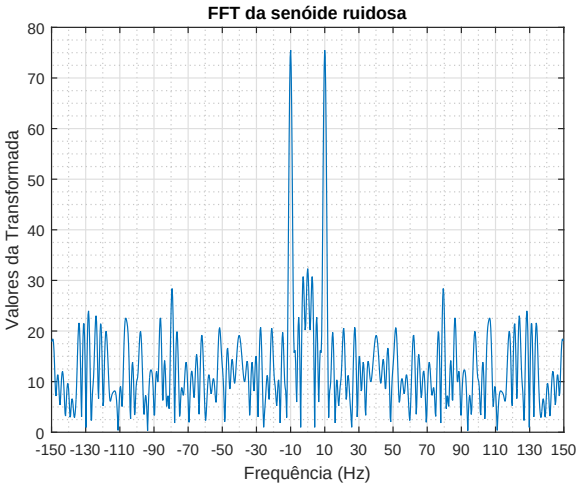
Como esperado as componentes espectrais se concentram em $\pm 10\,\text{Hz}$. Vamos adicionar ruído ao vetor agora, e ver o que acontece:

```
close all

y = x + randn(1,length(x)); %adiciona ruído

figure
plot(t,y)
title('Senóide com ruído gaussiano')
```



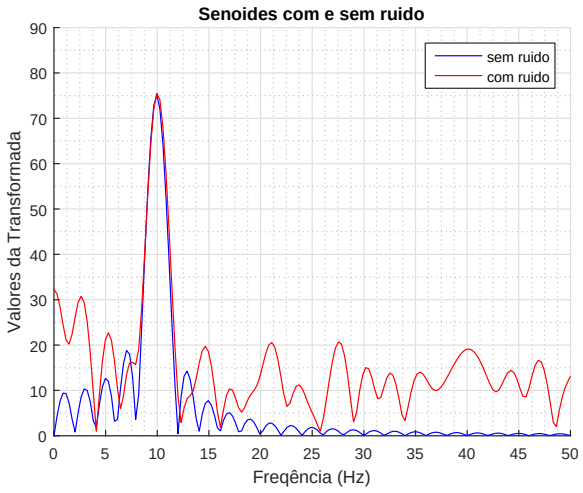
Nossa senóide agora está com muito ruído e praticamente irreconhecível, o que espera-se no espectro? Podemos refazer a FFT e verificar:

```
fVals = fs*(-NFFT/2:NFFT/2-1)/NFFT;
Y = fftshift(fft(y,NFFT));
figure
plot(fVals,abs(Y));
set(gca,'xtick',[-150:20:150])
title('FFT da senóide ruidosa');
xlabel('Frequência (Hz)')
ylabel('Valores da Transformada');
grid on
grid minor
```



Vemos que agora o resto do espectro ganhou magnitude, porém ainda assim a frequência dominante é a do sinal original, podemos conferir sobrepondo os gráficos e diminuindo a janela, como é usual não considerar frequências negativa esta parte é excluída do gráfico.

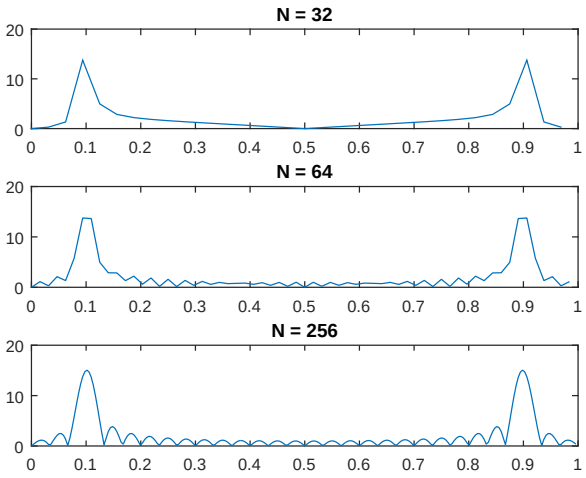
```
figure
hold on
plot(fVals,abs(X),'b','DisplayName','sem ruido'),axis([0 50 0 90])
plot(fVals,abs(Y),'r','DisplayName','com ruido'),axis([0 50 0 90])
title('Senoides com e sem ruido');
xlabel('Freqência (Hz)')
ylabel('Valores da Transformada');
legend('-DynamicLegend')
grid on
grid minor
hold off
```



Em sistemas complexos essa ferramenta é muito útil para encontrar fontes de ruídos e projetar filtros devidamente.

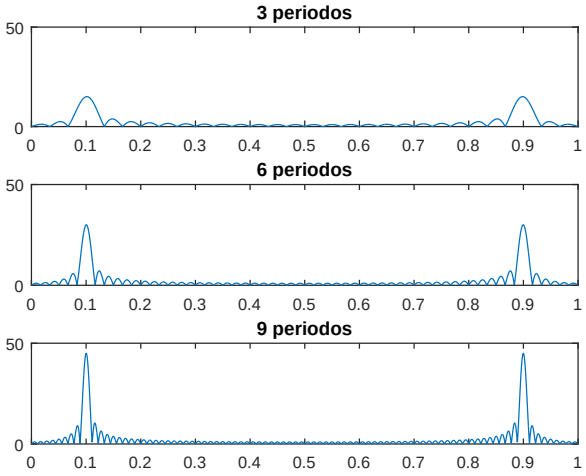
Vamos ver agora como mudar a quantidade de amostras utilizadas na FFT pode alterar o gráfico resultante.

```
n = [0:29];
x = cos(2*pi*n/10);

N1 = 32;
N2 = 64;
N3 = 256;

X1 = abs(fft(x,N1));
X2 = abs(fft(x,N2));
X3 = abs(fft(x,N3));

F1 = [0 : N1 - 1]/N1;
F2 = [0 : N2 - 1]/N2;
F3 = [0 : N3 - 1]/N3;

figure
subplot(3,1,1)
plot(F1,X1),title('N = 32'),axis([0 1 0 20])
subplot(3,1,2)
plot(F2,X2),title('N = 64'),axis([0 1 0 20])
subplot(3,1,3)
plot(F3,X3),title('N = 256'),axis([0 1 0 20])
```



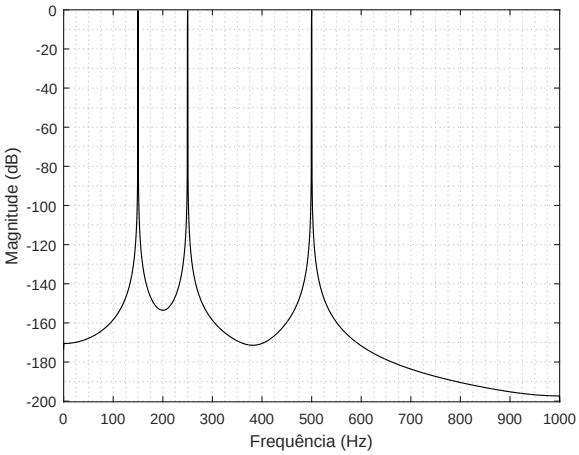
Como podemos ver claramente o aumento da quantidade amostras melhora a qualidade gráfico, é importante se restringir apenas para efeito de custo computacional, e sempre ter pelo menos a quantidade de amostras do próprio sinal. Normalmente valores na ordem de 2 a 5 vezes o tamanho do sinal são suficientes. Abaixo podemos ver o efeito da quantidade de períodos utilizados na FFT, quanto mais períodos adicionamos melhor fica a visualização do pico da frequência fundamental, porém vemos cada vez mais o efeito de vazamento de harmônicas no espectro:

```
n = [0:29];
x1 = cos(2*pi*n/10);
x2 = [x1 x1];
x3 = [x1 x1 x1];
N = 2048;
X1 = abs(fft(x1,N));
X2 = abs(fft(x2,N));
X3 = abs(fft(x3,N));
F = [0:N-1]/N;
figure
subplot(3,1,1)
plot(F,X1),title('3 periodos'),axis([0 1 0 50])
subplot(3,1,2)
plot(F,X2),title('6 periodos'),axis([0 1 0 50])
subplot(3,1,3)
plot(F,X3),title('9 periodos'),axis([0 1 0 50])
```



Pesquise por *Spectral Leakage* e leia sobre o teoria, pode ser muito útil aquisitando e tratando sinais discretos. Um método simples e rápido, porém com menos opções, é usar a função `pwelch(x)` para achar a magnitude doe sinais no espectro da frequência. Pesquise no manual do MATLAB como usar devidamente esta função, abaixo temos um exemplo simples:

```
T=10;
Ts=0.0005;
Fs=1/Ts;
t=[0:Ts:T];

x=cos(2*pi*150*t)+cos(2*pi*250*t)+sin(2*pi*500*t);

[pxx,f] = pwelch(x,5000,[],5000,Fs);
figure
plot(f,pow2db(pxx),'black'),grid minor
xlabel('Frequência (Hz)')
ylabel('Magnitude (dB)')
```



Este tipo de gráfico não pode ser considerado uma Transformada de Fourier diretamente, porém para identificar onde encontram-se as frequências fundamentais do sinal é útil.

Parte III
Assuntos Avançados

13 Diagrama de Bode

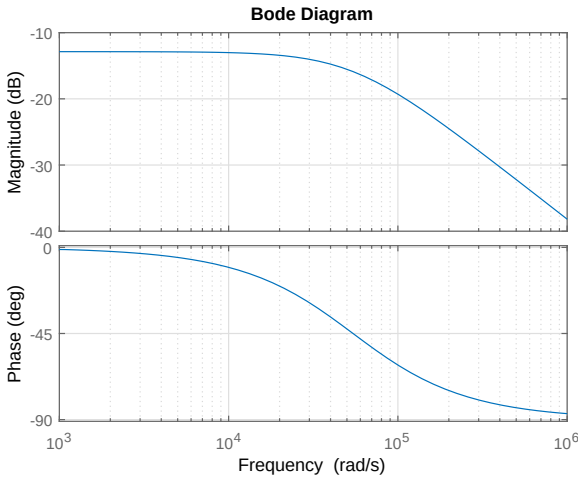
Diagrama de Bode é um tipo de gráfico muito utilizado e muito simples de se entender. Caso não saiba como interpretar um, leia sobre antes de tentar entender os códigos desta seção. Como diagramas de Bode sempre são de um sistema o primeiro passo é escrevê-lo por sua função de transferência:

```
s = tf('s');

H = (12345/(s+54321))
```

```
H =

    12345
  ---------
  s + 54321

Continuous-time transfer function.
```

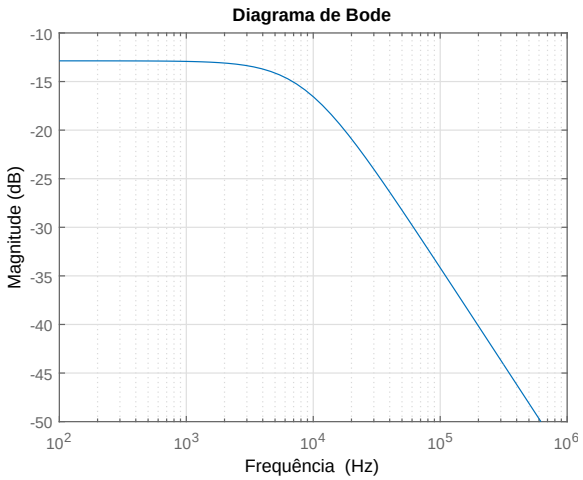
Em seguida chamamos a função:

```
figure
bodeplot(H)
grid on
```



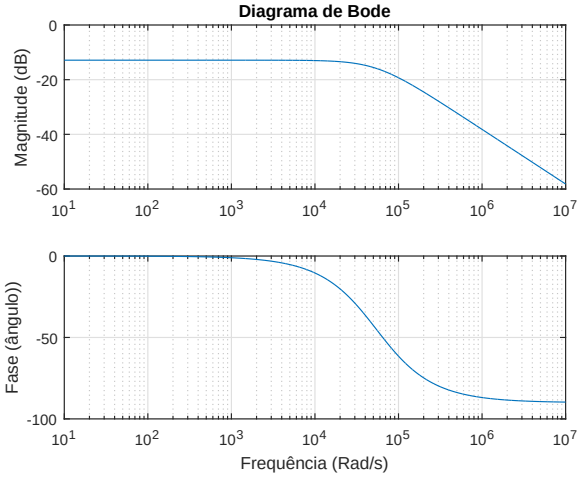
Assim como qualquer tipo de função gráfica, existem opções:

```
figure
h = bodeplot(H);
setoptions(h,'FreqUnits','Hz','PhaseVisible','off');
title('Diagrama de Bode')
ylabel('Magnitude')
xlabel('Frequência')
grid on
```



Um outro método não inclui nenhuma toolbox, mas é mais trabalhoso:

```
numTF=[0 12345];
denomTF=[1 54321];
w=0:10:10e6;

Y=freqs(numTF,denomTF,w); %freqs adquire dados em frequência
y1=abs(Y);
y2=angle(Y);

figure
subplot(2,1,1)
semilogx(w,20*log10(y1))
grid on
ylabel('Magnitude (dB)')
title('Diagrama de Bode')
subplot(2,1,2)
semilogx(w,y2*(180/pi))
grid on
ylabel('Fase (ângulo))')
xlabel('Frequência (Rad/s)')
```



Fazer manualmente é útil para melhor customizar as opções e escalas, mas deve ser feito com cuidado.

14 Filtros

Como será visto na matéria de Circuitos Elétricos e Eletrônicos, filtros são uma parte importantíssima do tratamento de sinais. Antes de começar a estudar os

códigos que virão a seguir, pesquise por filtros e aprenda os diferentes tipos. É muito provável que encontre (e entenda) os tipos mais básicos de filtros, sendo eles baseados em conceitos de circuitos elementares como o filtro RC, RL e RLC. Por mais que estes filtros sejam importantíssimos e formem a base para o entendimento de filtros em geral é importante ressaltar que os filtros utilizados em processamento de sinais são outros, que são idealizados puramente por uma função de transferência e tratados por software. Versões físicas destes filtros podem, também, serem idealizadas como circuitos mais complexos. São eles:

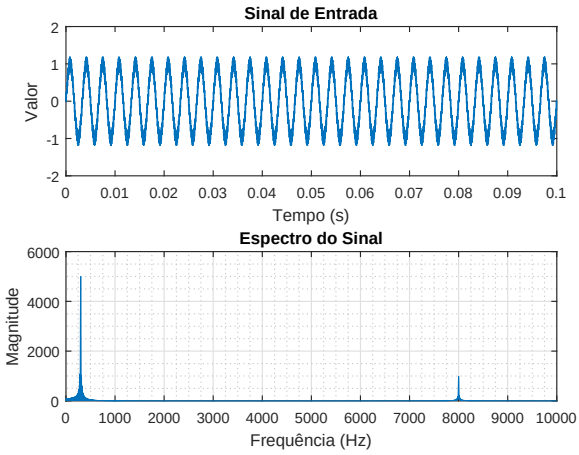
- Chebyshev 1 e 2: Melhor aproxima o filtro ideal quando especificada ordem e ripple permitido.
- Bessel: Mantém o diagrama de fase o mais reto possível, minimizando o delay.
- Elliptico: Possui a rampa mais inclinada de todos os tipos.
- Butterworth: Possui a resposta em frequência mais plana de todos os tipos, sendo o mais "comportado".

Qual filtro usar e como especifica-lo vem com experiência e bom senso. Assumindo que entenda o funcionamento de filtros em processamento de sinais, ou que ao menos aceite a existência deles, vamos ver como construir um filtro e filtrar um sinal no MATLAB.

No exemplo abaixo vamos construir um sinal composto de diversas frequências, fazer a FFT e ver quais são as componentes espectrais do sinal:

```
clear all
close all

ts = 0.00001; %periodo amostragem
fs  = 1/ts;
t = 0:ts:0.1;

x = sin(2*pi*300*t)+0.2*sin(2*pi*8000*t);
xlow = sin(2*pi*300*t);

figure
subplot(2,1,1)
plot(t,x)
title('Sinal de Entrada');
xlabel('Tempo (s)');
ylabel('Valor');

NFFT = length(t)*2;
fVals = fs*(-NFFT/2:NFFT/2-1)/NFFT;
X = fftshift(fft(x,NFFT));
subplot(2,1,2)
```

```
plot(fVals,abs(X));
grid on, grid minor, set(gca,'xtick',[0:1000:10000])
title('Espectro do Sinal'),axis([0 10000 0 6000]);
xlabel('Frequência (Hz)');
ylabel('Magnitude');
```



Como podemos ver acima, claramente as frequências do sinal encontram-se em 300 Hz e 8 kHz, conforme esperado já que criamos o sinal assim. Vamos agora criar um filtro que remova a componente de 8 kHz. Vamos usar um filtro do tipo Butterworth de ordem 5, e ver sua resposta em frequência.

Primeiro calculamos a frequência normalizada de 1 kHz, já que queremos um valor entre as duas componentes como frequência de corte:

```
nf = 2*pi*1000/fs
```

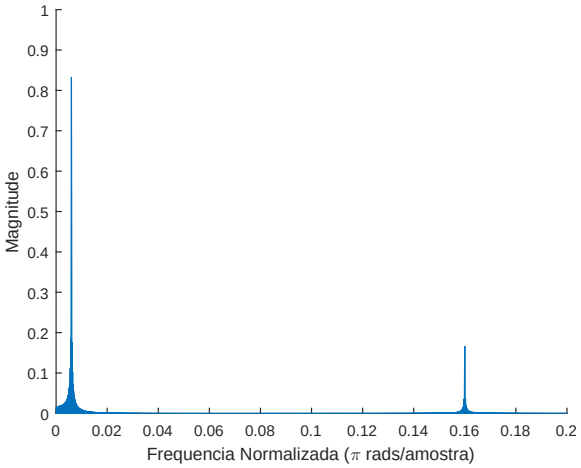
```
nf =

    0.0628
```

Alternadamente podemos plotar a FFT direto em frequência normalizada e achar a frequência normalizada por observação, embora isso necessite de algum trabalho de manipulação:

```
xm = abs(fft(x,fs*2));
tam = length(xm);
figure
hold on
plot(0:1/(tam/2 -1):1, xm(1:tam/2)/6000)
```

```
axis([0 0.2 0 1])
xlabel('Frequencia Normalizada (\pi rads/amostra)')
ylabel('Magnitude')
```

```
Warning: FFT length must be a nonnegative integer scalar.
Warning: Integer operands are required for colon operator when used as index
```

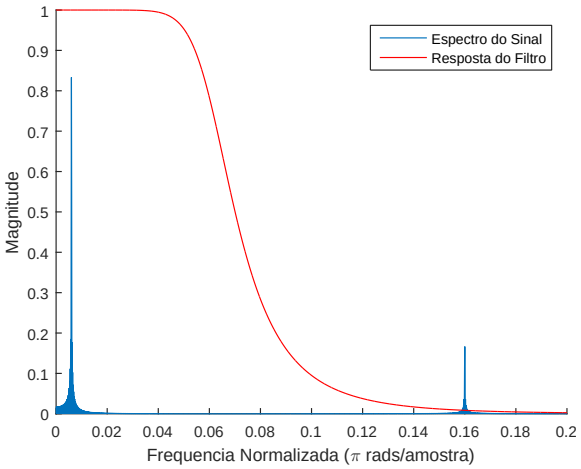


As funções de filtro (butter, cheby1, cheby2 e ellip) pegam como argumento a ordem do filtro, a(s) frequência(s) de corte e o tipo (passa-baixa, passa-alta, passa-banda, etc) e entrega os coeficientes de sua função de transferência. Olhe na documentação para aprender mais sobre seu uso. Criamos o filtro de acordo com as especificações:

```
ff = [0:1/(tam/2 -1):1];
[b,a] = butter(5,nf,'low');
```

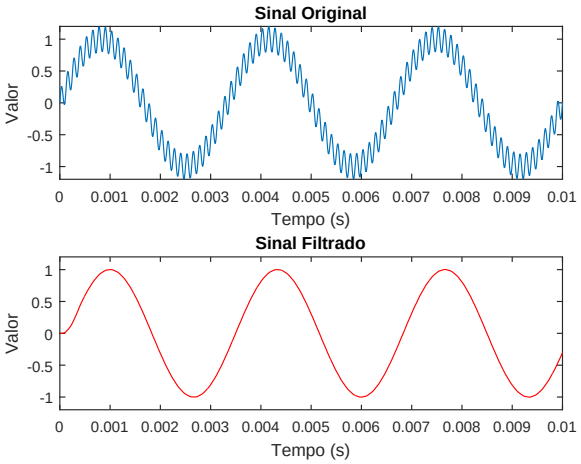
A função `freqz(b,a)` plota um diagrama de bode para análise de filtros discretos, pesquise pela `freqs(b,a)` para filtros contínuos.

```
h = freqz(b,a,floor(tam/2));
h = abs(h);
hold on
plot(ff,h,'r')
legend('Espectro do Sinal','Resposta do Filtro');
```



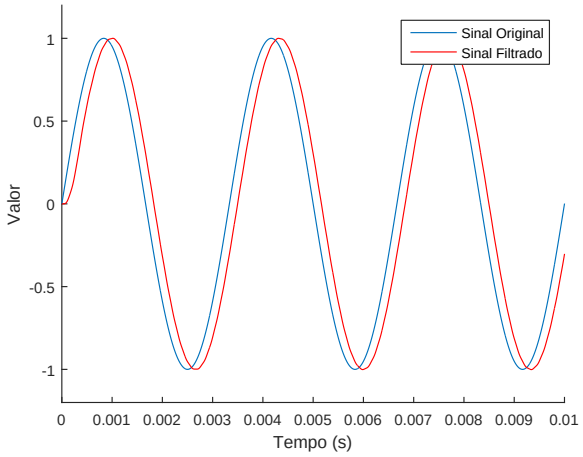
Como podemos ver o ganho do filtro passa a ser -3db na frequencia de corte especificada. Agora podemos filtrar:

```
xf = filter(b,a,x);

figure
subplot(2,1,1)
plot(t,x),title('Sinal Original'),axis([0 0.01 -1.2 1.2]);
xlabel('Tempo (s)');
ylabel('Valor');
subplot(2,1,2)
plot(t,xf,'r'),title('Sinal Filtrado'),axis([0 0.01 -1.2 1.2]);
xlabel('Tempo (s)');
ylabel('Valor');
```



É importante notar que o filtro apresenta atraso de fase, assim como qualquer sistema potencialmente pode. mais adiante vamos ver como a fase se comporta num determinado sistema a uma determinada frequência, mas por hora observe o atraso no gráfico comparativo abaixo, onde a curva do sinal original está sendo plotada apenas com a componente de baixa frequência (300 Hz).

```
figure
hold on
plot(t,xlow),axis([0 0.01 -1.2 1.2]);
xlabel('Tempo (s)');
ylabel('Valor');
plot(t,xf,'r'),axis([0 0.01 -1.2 1.2]);
legend('Sinal Original','Sinal Filtrado');
hold off
```



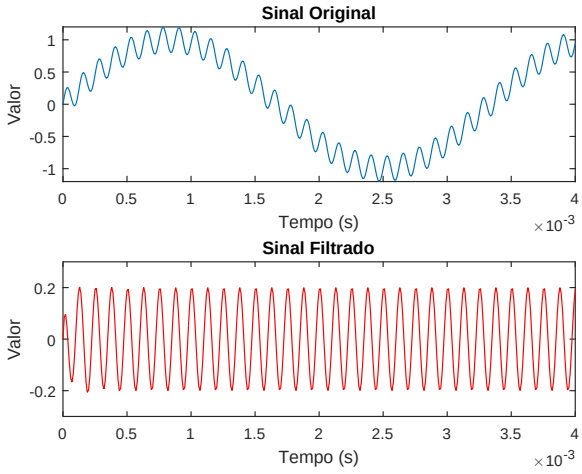
Observe que a maior parte das manipulações matemáticas foi para plotar gráficos didaticamente compreensíveis, se o objetivo é simplesmente filtrar, podemos usar menos comandos. No exemplo abaixo vamos trocar para um passa-altas, e ver a diferença:

```
[b,a] = butter(5,nf,'high');
xfh = filter(b,a,x);
```

Nos comandos acima o sinal já foi filtrado, plotar o resultado é simples, observe a escala dos eixos:

```
figure
subplot(2,1,1)
plot(t,x),title('Sinal Original'),axis([0 0.004 -1.2 1.2]);
xlabel('Tempo (s)');
```

```
ylabel('Valor');
subplot(2,1,2)
plot(t,xfh,'r'),title('Sinal Filtrado'),axis([0 0.004 -0.3 0.3]);
xlabel('Tempo (s)');
ylabel('Valor');
```



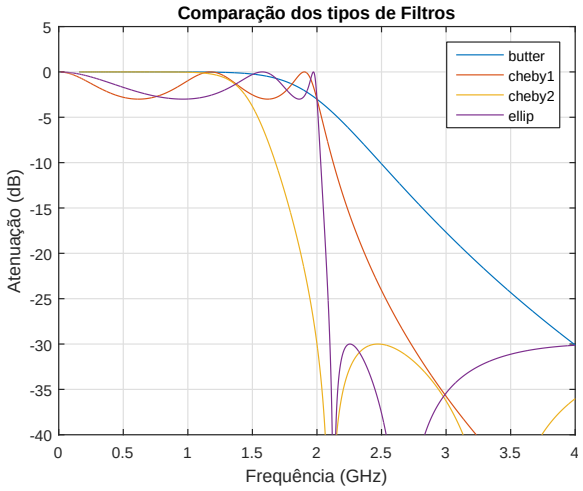
Filtros contínuos são feitos com a opção s, abaixo temos o exemplo do MATLAB, utilizando $F_c = 2\,\text{GHz}$ e ordem 5, todos passa-baixas. Na sequência de chamada `[zb,pb,kb] = butter(n,2*pi*f,'s');` , o filtro é gerado na forma de zeros/polos/ganhos, e como é contínuo passamos a freqência em radianos. A função `[bb,ab] = zp2tf(zb,pb,kb);` transforma os zeros/polos/ganhos em uma função de transferência, e `freqs(b,a,N);` gera a resposta em frequência para `b` e `a` com `N` amostras.

```
n = 5;
f = 2e9;

[zb,pb,kb] = butter(n,2*pi*f,'s');
[bb,ab] = zp2tf(zb,pb,kb);
[hb,wb] = freqs(bb,ab,4096);

[z1,p1,k1] = cheby1(n,3,2*pi*f,'s');
[b1,a1] = zp2tf(z1,p1,k1);
[h1,w1] = freqs(b1,a1,4096);

[z2,p2,k2] = cheby2(n,30,2*pi*f,'s');
[b2,a2] = zp2tf(z2,p2,k2);
[h2,w2] = freqs(b2,a2,4096);

[ze,pe,ke] = ellip(n,3,30,2*pi*f,'s');
```

```
[be,ae] = zp2tf(ze,pe,ke);
[he,we] = freqs(be,ae,4096);
```

Abaixo o eixo de frequência é normalizado para GHz e o eixo de magnitudes é transformado em dB, formato padrão.

```
figure
plot(wb/(2e9*pi),mag2db(abs(hb)))
hold on
plot(w1/(2e9*pi),mag2db(abs(h1)))
plot(w2/(2e9*pi),mag2db(abs(h2)))
plot(we/(2e9*pi),mag2db(abs(he)))
axis([0 4 -40 5])
grid
title('Comparação dos tipos de Filtros')
xlabel('Frequência (GHz)')
ylabel('Atenuação (dB)')
legend('butter','cheby1','cheby2','ellip')
```



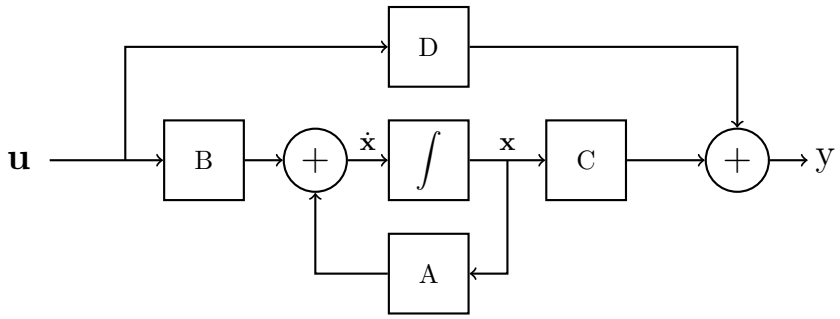
Na imagem acima fica clara a diferença fundamental entre os filtros.

15 Sistemas como Equações de Espaço-Estado

Funções de transferência também podem ser representadas por sistemas chamados de Equações de Espaço-Estado, que nada mais são do que representar as equações diferenciais que definem o sistema como um produto de matrizes. Equações diferenciais nesta forma assumem o formato abaixo:

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u}$$
$$\mathbf{y} = \mathbf{C}\mathbf{x} + \mathbf{D}\mathbf{u}$$

Onde **x** , **y** e **u** são os vetores de estado, saída e entrada, respectivamente, e **A** , **B** , **C** e **D** as matrizes que os relacionam. Pode-se traduzir esta relação para o diagrama de blocos abaixo, que torna mais fácil o entendimento:



Felizmente para nós o MATLAB contém diversas ferramentas para lidar com esse tipo de representação. A sequência de comandos abaixo cria um sistema neste formato:

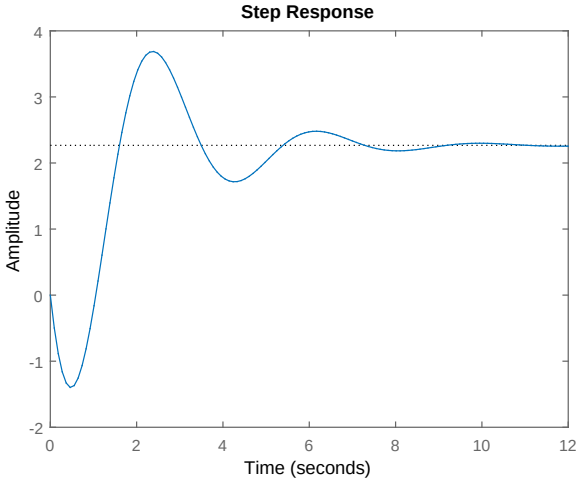
```
A = [1 1 ; -5 -2];
B = [1 ; 3];
C = [-1.2 -1.6];
D = 0;
sys = ss(A,B,C,D)
```

```
sys =

  a =
       x1  x2
   x1   1   1
   x2  -5  -2

  b =
       u1
   x1   1
   x2   3

  c =
```

```
       x1    x2
 y1  -1.2  -1.6

d =
     u1
 y1   0

Continuous-time state-space model.
```

É simulável da mesma forma que quando utilizando funções de transferência:

```
figure
step(sys)
```



também podemos achar a função de transferência e voltar facilmente:

```
[num, den] = ss2tf(A,B,C,D);
systf = tf(num,den)

[A,B,C,D] = tf2ss(num,den);
sys = ss(A,B,C,D)
```

```
systf =

  -6 s + 6.8
  -----------
  s^2 + s + 3
```

```
Continuous-time transfer function.


sys =

  a =
       x1  x2
   x1  -1  -3
   x2   1   0

  b =
       u1
   x1   1
   x2   0

  c =
        x1   x2
   y1   -6  6.8

  d =
       u1
   y1   0

Continuous-time state-space model.
```

Realizar sistemas no tempo discreto é feito adicionando-se um período de amostragem ao final do comando:

```
A = [0 1;-5 -2];
B = [0;3];
C = [0 1];
D = 0;

sys = ss(A,B,C,D,0.25)


sys =

  a =
       x1  x2
   x1   0   1
   x2  -5  -2

  b =
       u1
```

```
   x1   0
   x2   3

  c =
       x1  x2
   y1   0   1

  d =
       u1
   y1   0

Sample time: 0.25 seconds
Discrete-time state-space model.
```

Vamos simular um sistema Massa-Mola-Amortecedor unidimensional e ver como os parâmetros mudam a dinâmica do sistema. Começamos declarando os parâmetros base, onde `m` é a massa, `k` a constante elástica da mola e `b` a constante de atrito viscoso do do amortecedor.

```
clear all
close all

m = 1;
k = 1;
b = 0.2;
```

Montamos as matrizes de estado e em seguida o sistema:

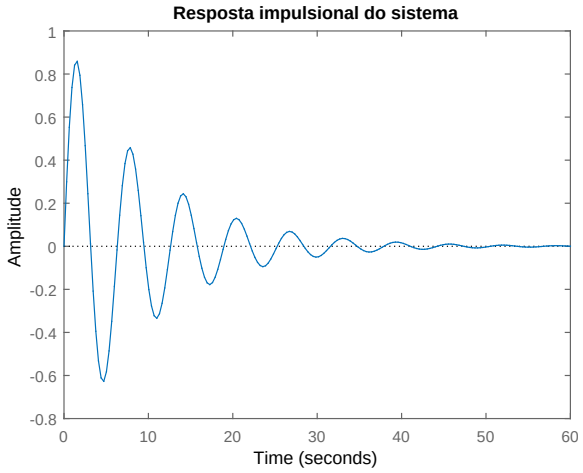
```
A = [0 1; -k/m -b/m];
B = [0 1/m]';
C = [1 0];
D = [0];
sys(1) = ss(A,B,C,D)
```

```
sys =

  a =
         x1    x2
   x1     0     1
   x2    -1  -0.2

  b =
       u1
   x1   0
```

```
   x2   1

c =
       x1  x2
   y1   1   0

d =
       u1
   y1   0

Continuous-time state-space model.
```

Utilizando um impulso no sistema vemos que o comportamento oscilatório é o esperado.

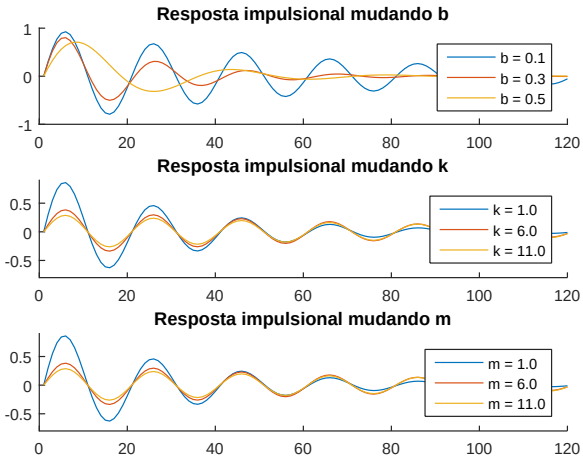
```
figure
impulse(sys(1))
title('Resposta impulsional do sistema')
```



Vamos agora mudar a constante de amortecimento, da mola e a massa, separadamente, e ver o que acontece com o sistema:

```
figure
subplot(3,1,1)
hold on
for b = 0.1:0.2:0.5
    A = [0 1; -k/m -b/m];
    B = [0 1/m]';
    sys(2) = ss(A,B,C,D);
```

```
    dn = (sprintf('b = %0.1f',b));
    plot(impulse(sys(2)),'DisplayName',dn),axis([0 120 -1 1])
    title('Resposta impulsional mudando b')
    legend('-DynamicLegend')
end
hold off
b = 0.2;
subplot(3,1,2)
hold on
for k = 1:5:11
    A = [0 1; -k/m -b/m];
    B = [0 1/m]';
    sys(2) = ss(A,B,C,D);
    dn = (sprintf('k = %0.1f',k));
    plot(impulse(sys(2)),'DisplayName',dn),axis([0 120 -0.8 0.9])
    title('Resposta impulsional mudando k')
    legend('-DynamicLegend')
end
hold off
k = 1;
subplot(3,1,3)
hold on
for m = 1:5:11
    A = [0 1; -k/m -b/m];
    B = [0 1/m]';
    sys(2) = ss(A,B,C,D);
    dn = (sprintf('m = %0.1f',m));
    plot(impulse(sys(2)),'DisplayName',dn),axis([0 120 -0.8 0.9])
    title('Resposta impulsional mudando m')
    legend('-DynamicLegend')
end
hold off
m = 1;
```



Todos os comportamentos comprovam o esperado.

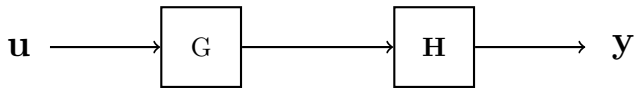
16 Associação de sistemas LTI

Associações de série, paralelo e realimentação são muito simples de se fazer em MATLAB, primeiro vamos criar algumas funções de transferência:

```
G = tf(2,[1 3 0])
H = zpk([],-5,5)
```

```
G =

     2
  ---------
  s^2 + 3 s

Continuous-time transfer function.


H =

    5
  -----
  (s+5)

Continuous-time zero/pole/gain model.
```

Conexões em série são da forma:

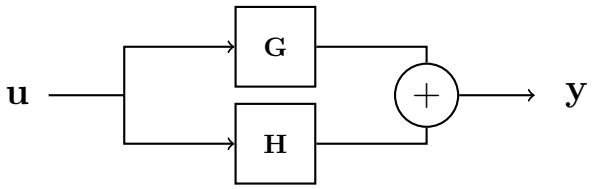


Estas conexões podem ser feitas multiplicando-se as funções de transferência ou utilizando o comando abaixo:

```
serie = series(G,H)
```

```
serie =

       10
  -------------
  s (s+5) (s+3)

Continuous-time zero/pole/gain model.
```

Conexões em paralelo são da forma:

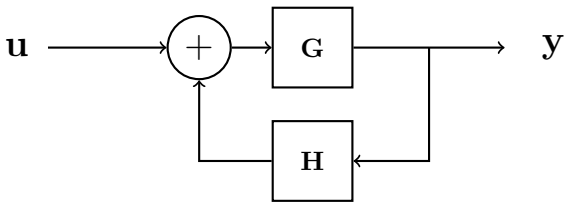


Estas conexões podem ser feitas utilizando o comando abaixo:

```
paralelo = parallel(G,H)
```

```
paralelo =

  5 (s+0.7566) (s+2.643)
  ----------------------
      s (s+3) (s+5)

Continuous-time zero/pole/gain model.
```

Conexões de realimentação são da forma:



Estas conexões podem ser feitas utilizando o comando abaixo:

```
realimentacao = feedback(G,H)
```

```
realimentacao =

             2 (s+5)
  --------------------------------
  (s+5.663) (s^2 + 2.337s + 1.766)

Continuous-time zero/pole/gain model.
```

A função de realimentação assume um ganho negativo unitário como padrão, caso queira mudar basta acrescentar o escalar ao final:

```
realimentacao2 = feedback(G,H,+1)
```

```
realimentacao2 =

              2 (s+5)
  --------------------------------
  (s-0.5157) (s^2 + 8.516s + 19.39)

Continuous-time zero/pole/gain model.
```

Todos estes comandos podem ser concatenados:

```
sistema = parallel(feedback(G,H),series(feedback(H,G),G))
```

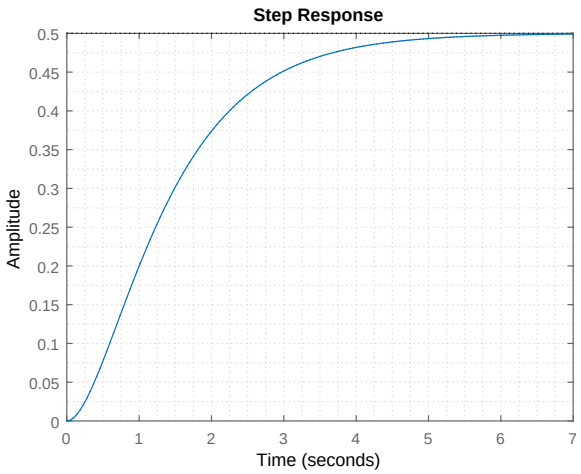
```
sistema =

  2 s (s+10) (s+5.663) (s+3) (s^2 + 2.337s + 1.766)
  ------------------------------------------------
    s (s+5.663)^2 (s+3) (s^2 + 2.337s + 1.766)^2

Continuous-time zero/pole/gain model.
```

17 Controle PID

Um dos tipos de controle mais famosos é o PID (*Proportional-Integral-Derivative*). Vamos ver como fazer um controlador PID em MATLAB e ver as vantagens que ele fornece. Começamos declarando a função de transferência da planta:

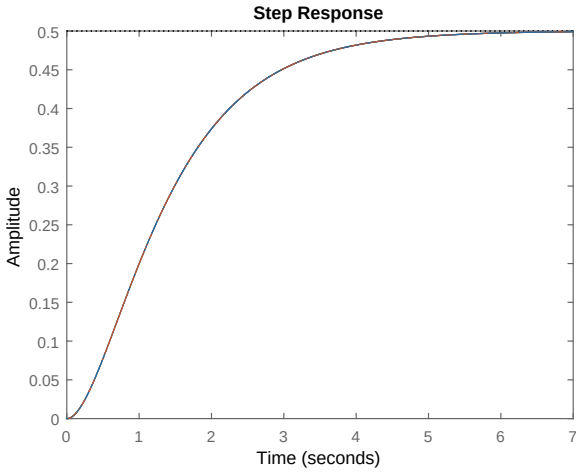
```
close all

num = [1];
den = [1 3 1];

Gp = tf(num,den)
Gu = feedback(Gp,1)

figure
grid minor
step(Gu)
```

```
Gp =

       1
  -------------
  s^2 + 3 s + 1

Continuous-time transfer function.


Gu =

       1
  -------------
  s^2 + 3 s + 2

Continuous-time transfer function.
```



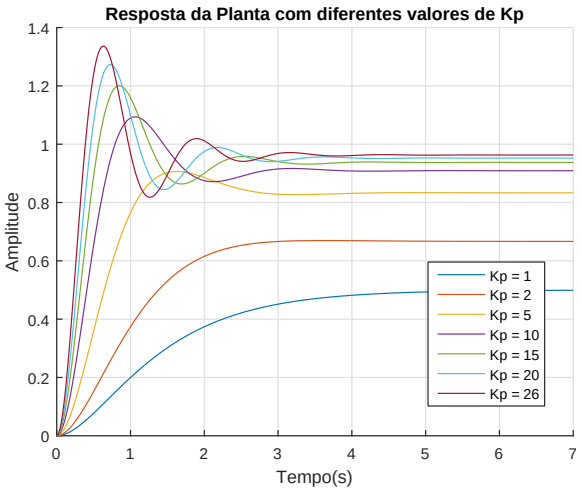
Como podemos ver o sistema demora bastante para responder e para em aproximadamente 0.5 em vez do nosso setpoint, que no caso é 1 já que o degrau de entrada é unitário. Vamos agora montar o controlador PID apenas com o ganho proporcional e ver o que acontece:

```
Kp = 1;
Kd = 0;
Ki = 0;
Gc = pid(Kp,Ki,Kd);
Gm = feedback(series(Gc,Gp),1);
t = (0:0.01:7)';

figure
grid on
step(Gu,Gm,t)
```



Com ganho unitário no controlador não vemos beneficio pois matematicamente a função só está sendo multiplicada por 1. Vamos plotar agora com alguns valores diferentes de Kp:

```
figure
hold on
grid on
for Kp = [1 2 5 10 15 20 26]
    Gc = pid(Kp,Ki,Kd);
    Gm = feedback(series(Gc,Gp),1);
    dn = sprintf('Kp = %d',Kp);
    data = step(Gm,t);
    plot(t,data,'DisplayName',dn)
    h = legend('-DynamicLegend');
    h.Location = 'best';
end
title('Resposta da Planta com diferentes valores de Kp')
ylabel('Amplitude')
xlabel('Tempo(s)')
```



Como podemos ver o sistema teve uma subida muito mais rápida, porém agora ganhamos um comportamento oscilatório. Podemos confirmar isto vendo os polos dos dois sitemas:

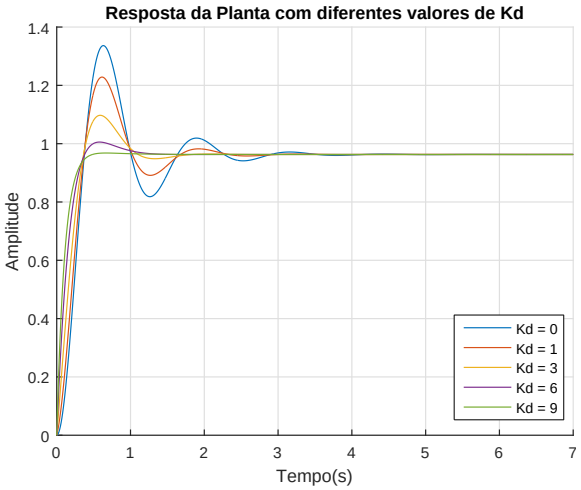
```
sem_controle = pole(Gu)
com_controle = pole(Gm)
```

```
sem_controle =

    -2
    -1


com_controle =

  -1.5000 + 4.9749i
  -1.5000 - 4.9749i
```

Claramente vemos que o sistema adquiriu polos imaginários, portanto oscilações. Vamos agora alterar o derivativo e ver o que acontece:

```
figure
hold on
grid on
for Kd = [0 1 3 6 9]
    Gc = pid(Kp,Ki,Kd);
    Gm = feedback(series(Gc,Gp),1);
```

```
    dn = sprintf('Kd = %d',Kd);
    data = step(Gm,t);
    plot(t,data,'DisplayName',dn)
    h = legend('-DynamicLegend');
    h.Location = 'best';
end
title('Resposta da Planta com diferentes valores de Kd')
ylabel('Amplitude')
xlabel('Tempo(s)')
```



Repare que os polos voltaram a ser somente reais, porém agora estão muito mais rápidos, ou seja, o sistema chega em regime permanente muito mais rápido e sem oscilações, podemos verificar vendo os polo novamente:

```
com_controle = pole(Gm)
```

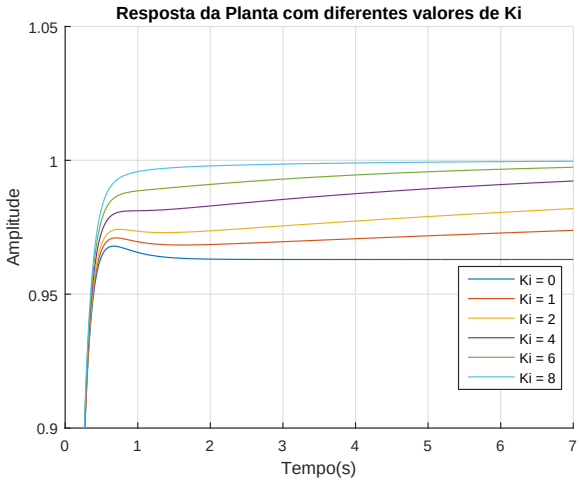
```
com_controle =

    -9
    -3
```

Vemos que o sistema agora está muito mais comportado, isto é devido ao fato de que o ganho derivativo tenta fazer com que a derivada do setpoint e da saída seja a mesma, portanto quanto maior mais comportado fica. Não podemos, no entanto, aumentar demais este parâmetro uma vez que ele "luta"contra o ganho proporcional e deixa o sistema mais lento. Olhando para este gráfico podemos achar que o sistema está ótimo já, mas repare que estamos fora do valor alvo em regime permanente que no nosso caso é 1. Isso é corrigido com ganho integral,

abaixo foi variado o valor de Ki e a janela da resposta diminuída para melhorar o entendimento:

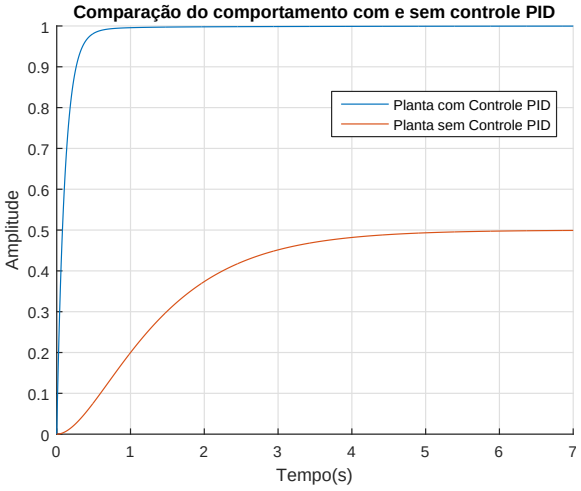
```
figure
hold on
grid on
for Ki = [0 1 2 4 6 8]
    Gc = pid(Kp,Ki,Kd);
    Gm = feedback(series(Gc,Gp),1);
    dn = sprintf('Ki = %d',Ki);
    data = step(Gm,t);
    plot(t,data,'DisplayName',dn),axis([0 7 0.9 1.05])
    h = legend('-DynamicLegend');
    h.Location = 'best';
end
title('Resposta da Planta com diferentes valores de Ki')
ylabel('Amplitude')
xlabel('Tempo(s)')
pole(Gm)
```

```
ans =

   -9.1425
   -2.5087
   -0.3488
```



A diferença entre a planta com controlador tunado e sem controlador é considerável:

```
figure
hold on
grid on
data = step(Gm,t);
plot(t,data,'DisplayName','Planta com Controle PID')
data = step(Gu,t);
plot(t,data,'DisplayName','Planta sem Controle PID')
h = legend('-DynamicLegend');
h.Location = 'best';
title('Comparação do comportamento com e sem controle PID')
ylabel('Amplitude')
xlabel('Tempo(s)')
```

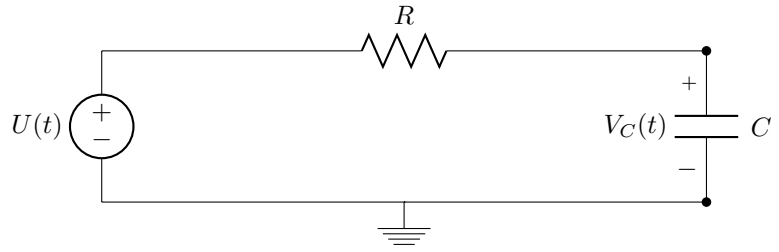


18 Circuitos RC, RL e RLC

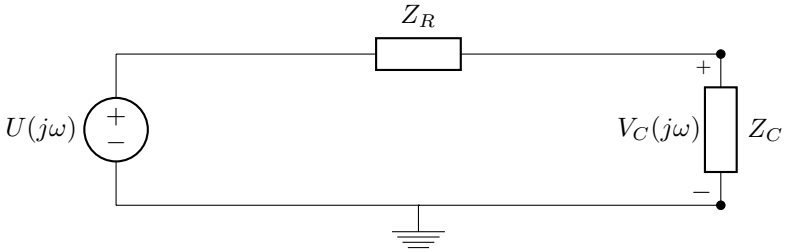
Vamos ver agora como simular circuitos básicos e fazer filtros com eles. Primeiro precisamos conhecer as impedâncias, são elas:

$$Z_R = R, \quad Z_C = \frac{1}{j\omega C} \quad e \quad Z_L = j\omega L$$

O que essas impedâncias nos dizem são que R não tem sua resistência aparente ao sistema alterada com a frequência, enquanto que a de L aumenta e a de C diminui com o aumento da frequência. Vamos modelar o circuito RC abaixo, tomando como saída a tensão $V_L(t)$ através do capacitor C :



Para fazer a análise no domínio da frequência basta trocar R e C por suas respectivas impedâncias e tomar como saída a tensão $V_c(j\omega)$ através do capacitor:



O problema agora passou a ser um simples divisor de tensão já que podemos tratar todos os elementos como resistências e a solução para a função de transferência $H(j\omega)$ vira:

$$H(j\omega) = \frac{V_C(j\omega)}{U(j\omega)} = \frac{\frac{1}{j\omega C}}{\frac{1}{j\omega C} + R} = \frac{1}{1 + j\omega RC}$$

Podemos agora aplicar o operador de Laplace e montar a função de transferência no MATLAB, utilizando valores para R e C:

$$H(s) = \frac{1}{1 + sRC} \qquad\qquad f_c = \frac{1}{2\pi RC}[\text{Hz}]$$

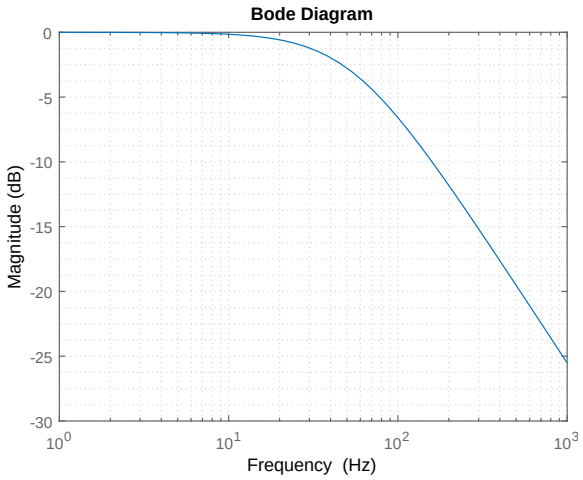
```
s = tf('s');
R = 1*10^3;
C = 3*10^-6;
```

```
H = 1/(1+s*R*C)
```

```
H =

      1
  -----------
  0.003 s + 1

Continuous-time transfer function.
```

O diagrama de Bode mostra como as magnitudes das baixas frequências ficam virtualmente inalteradas, enquanto que as mais altas sofrem atenuação.

```
figure
grid minor
h = bodeplot(H);
setoptions(h,'FreqUnits','Hz','PhaseVisible','off');
```



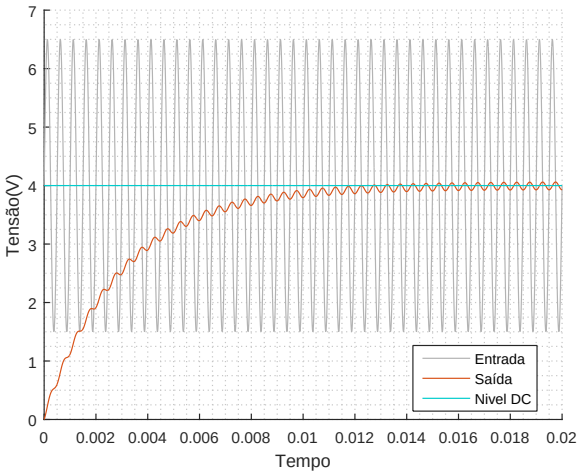
Podemos calcular a frequência de corte em Hz:

```
fc = 1/(2*pi*R*C)
```

```
fc =

   53.0516
```

Vamos passar uma senóide de alta frequência e ver se conseguimos extrair o nível DC:

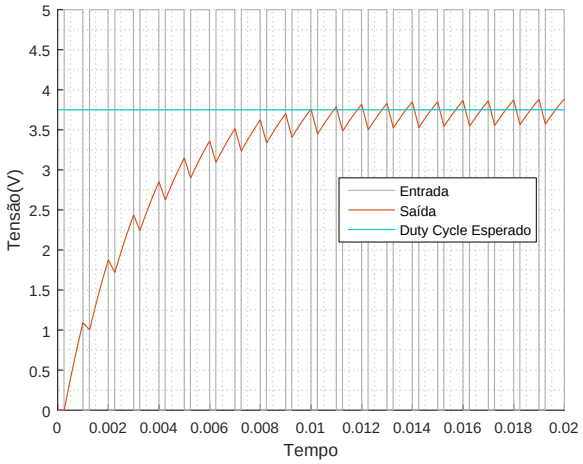
```
f = 2e3; AmpSin = 2.5;nDC = 4;
t0 = 0; Tf = 0.020; Ts = 10e-6;
t=t0:Ts:Tf;
fun=AmpSin*sin(2*pi*f*t)+nDC;
figure
hold on
grid minor
[y,t] = lsim(H,fun,t);
h = plot(t,fun,'DisplayName','Entrada');
set(h,'Color',[.7 .7 .7]);
plot(t,y,'DisplayName','Saída');
h = plot(t,nDC*ones(length(t),1),'DisplayName','Nivel DC');
set(h,'Color',[0 .8 0.8]);
xlabel('Tempo')
ylabel('Tensão(V)')
h = legend('-DynamicLegend');
h.Location = 'best';
hold off
```



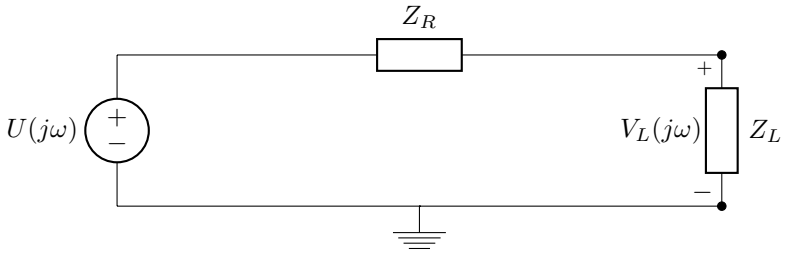
De fato vemos que o embora com algum ripple o filtro tirou grande parte das altas frequências. Podemos usar a mesma estratégia para extrair o *Duty-Cycle* de um sinal do tipo PWM (*Pulse Width Modulation*):

```
f = 1e3;AmpSaw = 2.5; duty = .75;
AmpPwm = 5; v = (1-duty)*AmpPwm;
fun=AmpSin*sawtooth(2*pi*f*t)+AmpSin;
pwm=AmpPwm*(fun>v);
```

```
figure
hold on
grid minor
[y,t] = lsim(H,pwm,t);
h = plot(t,pwm,'DisplayName','Entrada');
set(h,'Color',[.7 .7 .7]);
plot(t,y,'DisplayName','Saída');
h = plot(t,duty*AmpPwm*ones(length(t),1),'DisplayName',...
    'Duty Cycle Esperado');
set(h,'Color',[0 .8 0.8]);
xlabel('Tempo')
ylabel('Tensão(V)')
h = legend('-DynamicLegend');
h.Location = 'best';
hold off
```



Vemos novamente que dentro de um ripple aceitável o filtro se estabiliza na tensão efetiva do PWM. Simular um circuito RL pode ser feito da mesma forma, apenas trocando a impedância Z_C por Z_L e tomando $V_L(j\omega)$ como a saída. O circuito e a função de transferência resultante se torna:



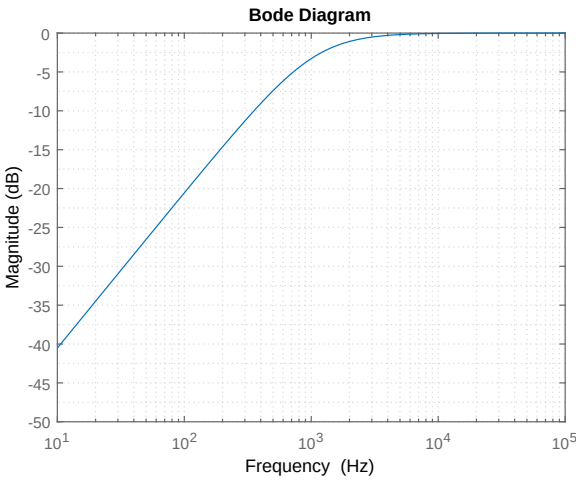
$$H(s) = \frac{Ls}{R + Ls} \qquad\qquad f_c = \frac{R}{2\pi L}[\text{Hz}]$$

```
s = tf('s');
R = 2*10^2;
L = 3*10^-2;
H = (L*s)/(R+L*s)

fc = R/(2*pi*L)
```

```
H =

    0.03 s
  ------------
  0.03 s + 200

Continuous-time transfer function.


fc =

   1.0610e+03
```

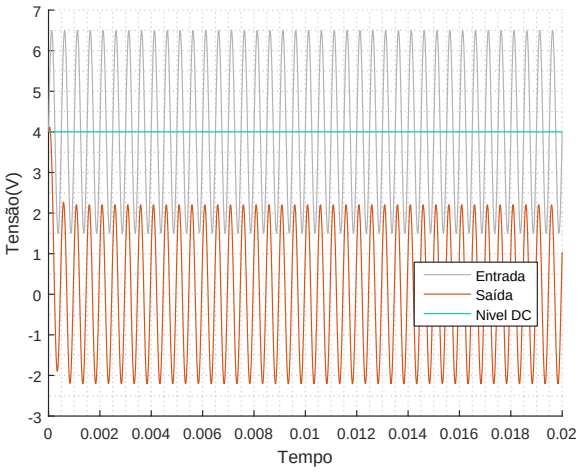
Se agora filtrarmos o mesmo sinal senóide anterior vamos manter as altas frequências, e eliminar o nível DC. Isto é intuitivo pois a impedância de um indutor aumenta com a frequência, tornando a resistência aparente maior para as altas frequências e estas portanto possuem uma presença maior na saída. Vamos ver como ficou o diagrama de Bode:

```
figure
grid minor
h = bodeplot(H);
setoptions(h,'FreqUnits','Hz','PhaseVisible','off');
```



Vemos que o comportamento agora é inverso, vamos simular e ver se eliminamos as baixas frequências:

```
f = 2e3; AmpSin = 2.5;nDC = 4;
t0 = 0; Tf = 0.020; Ts = 10e-6;
t=t0:Ts:Tf;
fun=AmpSin*sin(2*pi*f*t)+nDC;
figure
hold on
grid minor
[y,t] = lsim(H,fun,t);
h = plot(t,fun,'DisplayName','Entrada');
set(h,'Color',[.7 .7 .7]);
plot(t,y,'DisplayName','Saída');
h = plot(t,nDC*ones(length(t),1),'DisplayName','Nivel DC');
set(h,'Color',[0 .8 0.8]);
xlabel('Tempo')
ylabel('Tensão(V)')
h = legend('-DynamicLegend');
h.Location = 'best';
hold off
```



De fato podemos ver que a função perdeu o nível DC e encontra-se praticamente centrada em zero. Podemos verificar fazendo uma média parcial da senóide resultante:

```
mean(y(ceil(length(y)/2):length(y)))
```

```
ans =

    0.0010
```

Outra forma de inverter o tipo de filtro é trocar a ordem dos elementos. Para filtros de segunda ordem são necessários mais elementos armazenadores de energia, sejam estes mais dos mesmos ou um circuito RLC. A forma de achar a função de transferência e simular é sempre a mesma. Tente colocar um capacitor em paralelo com um indutor no lugar do único elemento armazenador de energia e verifique que agora o filtro é do tipo passa-banda. Experimente também chamar a interface interativa sobre circuitos RLC executando o comando `rlc_gui` na Command Window.

19 Filtragem por Convolução

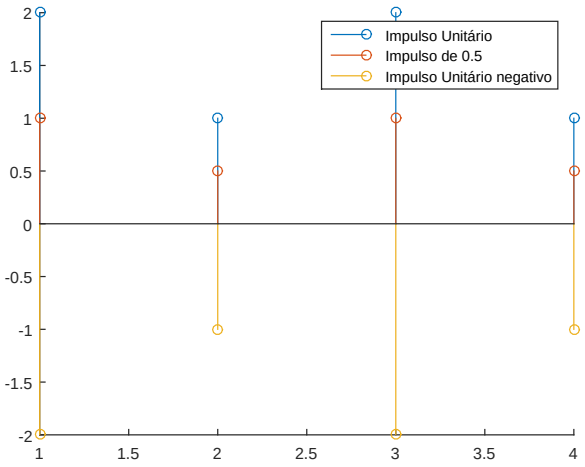
A convolução tem propriedades importantíssimas para o processamento de sinais. Vamos mostrar algumas utilizando MATLAB, é importante que possua um bom conhecimento sobre o processo de convolução para visualizar os procedimentos. Começamos criando um sinal simples:

```
v = [2 1 2 1];
```

Vamos tentar convoluir com alguns impulsos:

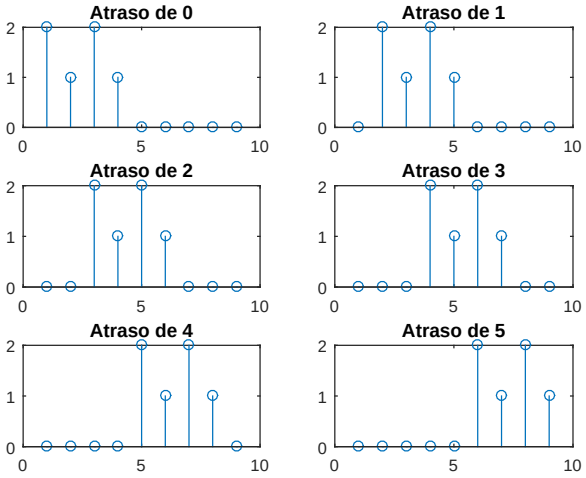
```
c1 = conv(1,v)
c2 = conv(0.5,v)
c3 = conv(-1,v)
figure
hold on
stem(c1);stem(c2);stem(c3);
h=legend('Impulso Unitário','Impulso de 0.5','Impulso Unitário negativo');
h.Location = 'best';
```

```
c1 =

     2     1     2     1


c2 =

    1.0000    0.5000    1.0000    0.5000


c3 =

    -2    -1    -2    -1
```



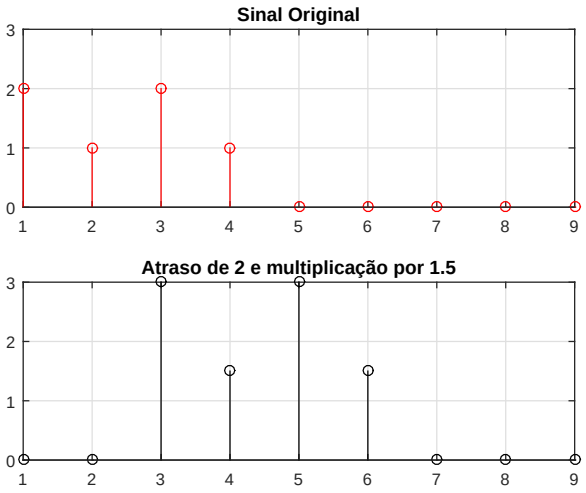
Convoluir um sinal por um escalar faz uma escala do vetor, daí o nome. Você pode conferir melhor checando os resultados numéricos em vez do plot. Esta propriedade é muito útil para operações com vetores. Vamos tentar algo mais complexo:

```
figure
for i=1:6
    subplot(3,2,i)
    d = zeros(1,6);
    d(i) = 1;
    stem(conv(v,d)),title(sprintf('Atraso de %d',i-1))
end
```



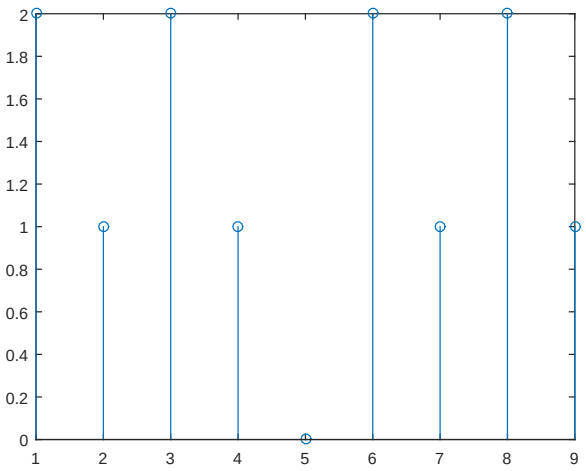
Conhecendo as propriedades da convolução podemos rapidamente compreender que ao convoluir com um impulso atrasado, atrasamos o sinal também. Juntando essas duas propriedades podemos multiplicar o sinal por constantes e fazer deslocamentos no tempo facilmente:

```
figure
hold on
subplot(2,1,1)
stem(conv(v,[1 0 0 0 0 0]),'r'),grid on,axis([1 9 0 3])
title('Sinal Original')
subplot(2,1,2)
stem(conv(v,[0 0 1.5 0 0 0]),'black')
title('Atraso de 2 e multiplicação por 1.5'), grid on
```



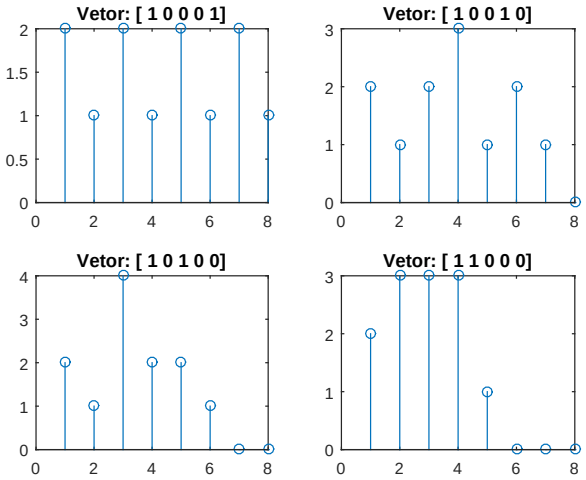
Isso tudo cai no fato de que podemos utilizar a convolução para filtrar sinais. Basta pensar que colocando o vetor certo para convoluir com o sinal, podemos alterar como quisermos o mesmo. Vamos fazer algumas operações para tentar entender como esse processo se realiza:

```
a=[1 0 0 0 0 1];

figure
stem(conv(v,a))
```



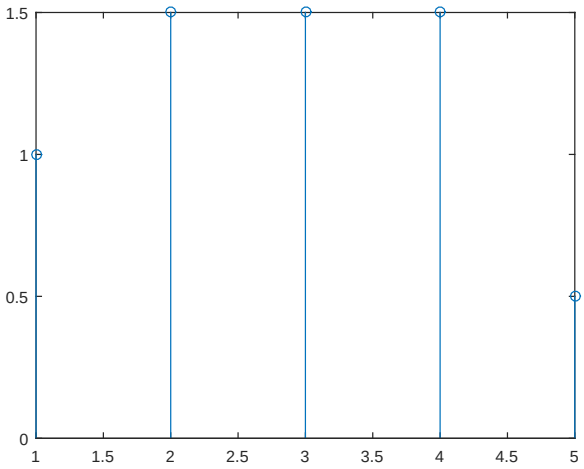
A convolução acima gerou duas cópias do sinal, se diminuirmos o atraso do segundo pulso vamos, eventualmente, fazer os sinais colidirem:

```
figure
for i=1:4
    subplot(2,2,i)
    d = zeros(1,5);d(1)=1;
    d(6-i) = 1;
    stem(conv(v,d)),title(strcat('Vetor: [',sprintf(' %d',d),']'))
end
```



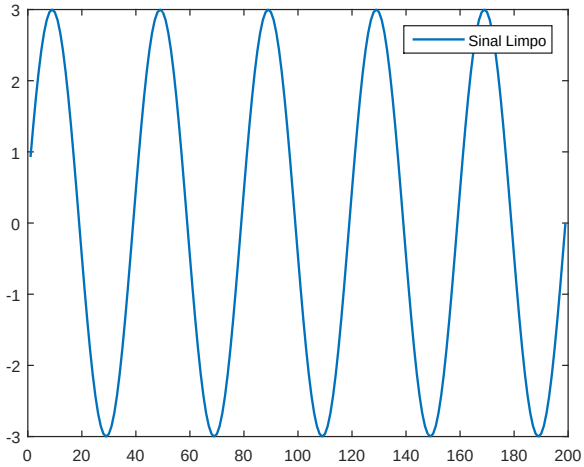
O que a operação acima fez foi mostrar que podemos também fazer produtos de sinais, e desta forma construir filtros. Vamos fazer um filtro de média móvel utilizando este conceito. Sabendo como a convolução faz o somatório podemos fazer um filtro de média móvel de duas amostras simplesmente com:

```
f=conv(v,[0.5 0.5]);
figure
stem(f)
```



A principio pode não parecer muita coisa, mas repare que o sinal ficou muito mais estável, perdeu oscilações. A visualização fica mais fácil com um sinal mais completo:

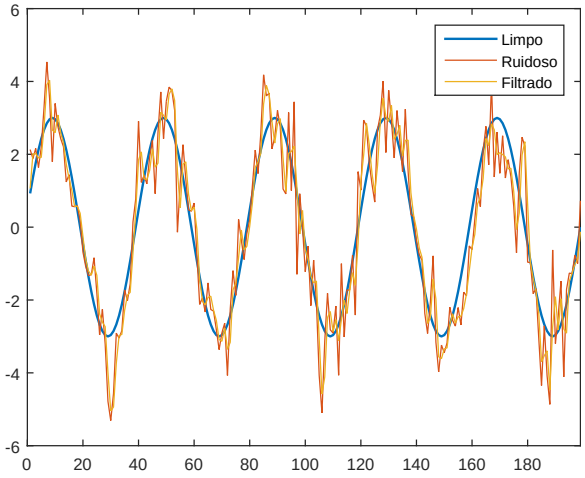
```
close all
s=sin(pi/10:pi/20:10*pi)*3;
figure
plot(s,'LineWidth',1.4)
legend('Sinal Limpo');
```



Vamos contaminar com ruído e ver se conseguimos filtrar com o mesmo filtro anterior:

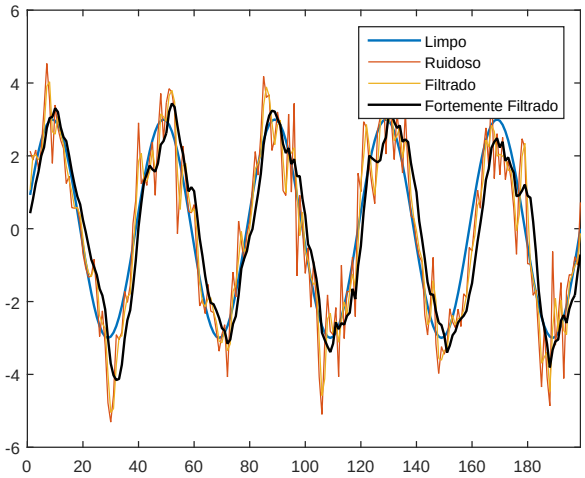
```
n=randn(size(s));
```

```
sn=s+n;
hold on;
plot(sn)
legend([{'Sinal Limpo'},{'Sinal Contaminado'}]);
plot(conv(sn,[0.5 0.5])),axis([0 length(sn) -6 6])
legend([{'Limpo'},{'Ruidoso'},{'Filtrado'}]);
```



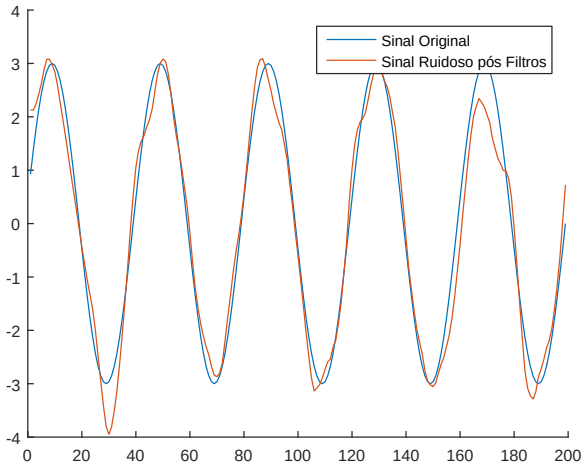
Vemos que o filtro de média móvel de duas amostras não se mostrou muito eficaz. Podemos aumentar para um de 5 fazendo:

```
N = 5;
d(1:N)=1/N;
plot(conv(sn,d),'black','LineWidth',1.4),axis([0 length(sn) -6 6])
legend([{'Limpo'},{'Ruidoso'},{'Filtrado'},{'Fortemente Filtrado'}]);
```



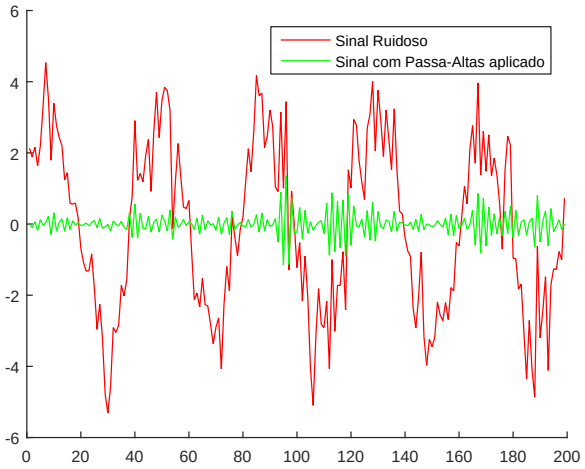
O filtro agora foi muito mais efetivo, porém causou um atraso de fase. O MATLAB possui uma função que contorna atrasos de fase, filtrando o sinal duas vezes (uma em cada sentido), isto também faz com que o filtro seja duas vezes mais potente:

```
figure
hold on
plot(s)
plot(filtfilt(d,1,sn))
legend({'Sinal Original','Sinal Ruidoso pós Filtros'})
```



O sinal filtrado agora não possui atraso, porém o filtro começou a diminuir a amplitude da senóide também. Estude filtros de média móvel para entender como funcionam e logo entenderá o motivo desta atenuação. Se inserir um filtro sem fortes flutuações faz o filtro impedir as fortes flutuações de passar, o que um filtro com fortes flutuações irá fazer? Vamos usar a mesma estratégia, porém fazendo o vetor do filtro flutuar:

```
a = [1 -1 1 -1]/4;

figure
hold on;
plot(sn,'r')
plot(filtfilt(a,1,sn), 'g')
legend([{'Sinal Ruidoso'},{'Sinal com Passa-Altas aplicado'}]);
hold off
```



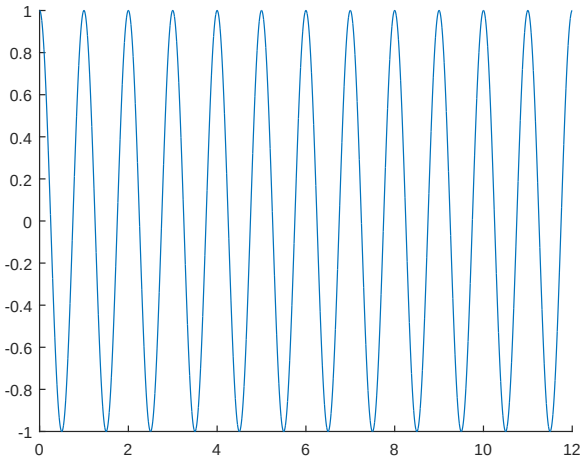
Um filtro de oscila o quão rápido a taxa de amostragem permite irá fazer com que todas as frequências altas passem. O que estamos fazendo aqui é criando uma resposta impulsional arbitrária e convoluindo-a com um sinal. A idéia é que podemos filtrar qualquer sinal com a convolução de um filtro cuja função de transferência gera uma resposta impulsional adequada, e isto é válido para qualquer filtro FIR (*Finite Impulse Response*).

20 Teorema de Amostragem de Nyquist-Shannon

O teorema de amostragem de Nyquist-Shannon diz que se amostrarmos um sinal ao dobro sua frequência, é possível reconstruir o sinal completamente. Vamos colocar em prática esta idéia e juntar com os conceitos de série de Fourier para verificar este mecanismo:

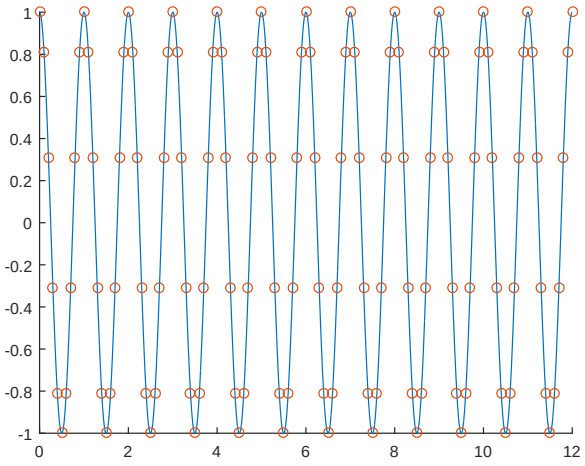
```
clear all
Tf = 12
Ts = 0.001;
Fs = 1/Ts;
t = 0:0.001:Tf;
y = sin(2*pi*t+pi/2);

figure
hold on
plot(t,y);


Tf =
```

12



Vamos começar amostrando a uma frequência maior que a frequência crítica de Nyquist (que neste caso é 2 Hz). Vamos começar a 10 Hz :

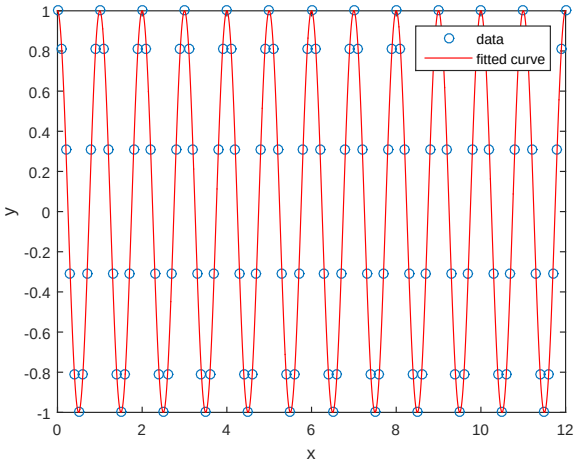
```
Fs = 10;

tsamp = 0:1/Fs:Tf;
ysamp = sin(2*pi*tsamp+pi/2);

plot(tsamp,ysamp,'o')
```



Vamos utilizar uma Série de Fourier nas amostras e ver a curva resultante:

```
figure
f = fit(tsamp',ysamp','fourier1')
plot(f,tsamp,ysamp,'o')
```

```
f =

     General model Fourier1:
     f(x) =  a0 + a1*cos(x*w) + b1*sin(x*w)
     Coefficients (with 95% confidence bounds):
       a0 =  -1.892e-17  (-3.503e-16, 3.124e-16)
       a1 =           1  (1, 1)
       b1 =   2.018e-16  (-7.393e-16, 1.143e-15)
       w =       6.283  (6.283, 6.283)
```

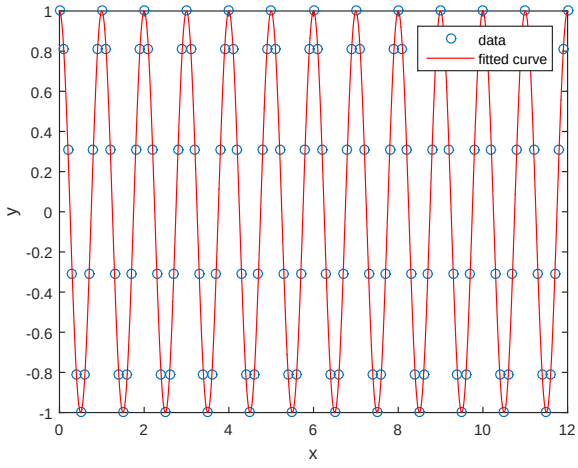


O MATLAB nos da os coeficientes da série como resposta, e plotando o resultado vemos que a função foi recriada perfeitamente. Podemos aumentar o grau mas não vamos ganhar nada com isso, repare nos valores dos coeficientes:

```
figure
f = fit(tsamp',ysamp','fourier4')
plot(f,tsamp,ysamp,'o'),axis([0 12 -1 1])
```

```
f =

     General model Fourier4:
     f(x) =  a0 + a1*cos(x*w) + b1*sin(x*w) +
```

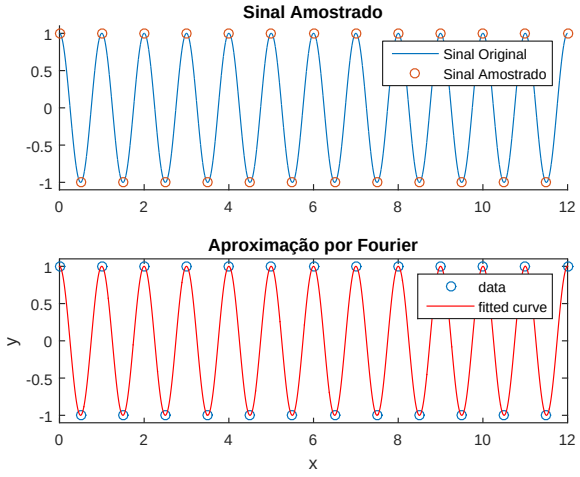
```
          a2*cos(2*x*w) + b2*sin(2*x*w) + a3*cos(3*x*w) + b3*sin(3*x*w) +
          a4*cos(4*x*w) + b4*sin(4*x*w)
Coefficients (with 95% confidence bounds):
  a0 =  -1.388e-17  (-3.4e-16, 3.122e-16)
  a1 =  -2.199e-16  (-6.803e-16, 2.406e-16)
  b1 =  -1.698e-16  (-6.319e-16, 2.923e-16)
  a2 =   1.098e-16  (-3.519e-16, 5.714e-16)
  b2 =   -6.63e-17  (-5.284e-16, 3.958e-16)
  a3 =  -6.362e-16  (-1.105e-15, -1.678e-16)
  b3 =   1.895e-16  (-2.727e-16, 6.516e-16)
  a4 =           1  (1, 1)
  b4 =   2.018e-16  (-7.473e-16, 1.151e-15)
  w =       1.571  (1.571, 1.571)
```



Vemos que novamente apenas um coeficiente prevaleceu, enquanto que o resto resultou em aproximadamente zero. Vamos tentar agora na frequência crítica de Nyquist.

```
figure
subplot(2,1,1)
hold on
plot(t,y),title('Sinal Amostrado'),axis([0 12 -1.1 1.1])
tsamp = 0:1/2:Tf;
ysamp = sin(2*pi*tsamp+pi/2);
plot(tsamp,ysamp,'o')
legend('Sinal Original','Sinal Amostrado')
f = fit(tsamp',ysamp','fourier4')
subplot(2,1,2)
plot(f,tsamp,ysamp,'o'),axis([0 12 -1.1 1.1])
title('Aproximação por Fourier')
```

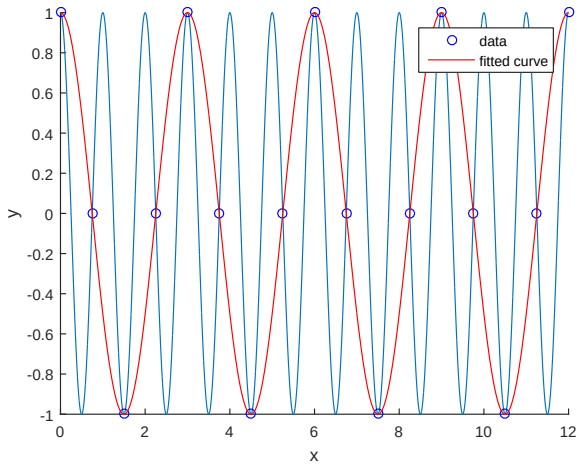
```
f =

     General model Fourier4:
     f(x) =  a0 + a1*cos(x*w) + b1*sin(x*w) +
               a2*cos(2*x*w) + b2*sin(2*x*w) + a3*cos(3*x*w) + b3*sin(3*x*w) +
               a4*cos(4*x*w) + b4*sin(4*x*w)
     Coefficients (with 95% confidence bounds):
       a0 =   2.073e-06  (-8.767e-06, 1.291e-05)
       a1 =   4.765e-06  (-1.055e-05, 2.008e-05)
       b1 =  -1.829e-08  (-1.56e-05, 1.556e-05)
       a2 =   0.0008358  (-0.0003875, 0.002059)
       b2 =  -6.417e-06  (-0.001067, 0.001054)
       a3 =           1  (0.9999, 1)
       b3 =    -0.01152  (-0.01507, -0.007965)
       a4 =  -0.0008316  (-0.002055, 0.0003919)
       b4 =   1.277e-05  (-0.001047, 0.001073)
       w =       2.094  (2.094, 2.094)
```



Por defeitos de aproximação, o MATLAB requer muitos pontos de amostragem para a função fit, em alguns casos isto é contornado aumentando o grau do polinômio, neste exercício grau 4 foi suficiente. Podemos verificar claramente que o sinal permanece intacto. Vamos diminuir mais um pouco a frequência e ver se ainda conseguimos reconstruir o sinal:

```
figure
hold on
plot(t,y)
tsamp = 0:0.75:Tf;
ysamp = sin(2*pi*tsamp+pi/2);
```

```
f = fit(tsamp',ysamp','fourier4')
plot(f,tsamp,ysamp,'bo')

f =

     General model Fourier4:
     f(x) =  a0 + a1*cos(x*w) + b1*sin(x*w) +
               a2*cos(2*x*w) + b2*sin(2*x*w) + a3*cos(3*x*w) + b3*sin(3*x*w) +
               a4*cos(4*x*w) + b4*sin(4*x*w)
     Coefficients (with 95% confidence bounds):
       a0 =  -2.366e-16  (-1.756e-15, 1.283e-15)
       a1 =   1.283e-15  (-8.429e-16, 3.408e-15)
       b1 =   2.655e-17  (-2.153e-15, 2.206e-15)
       a2 =    3.36e-16  (-1.835e-15, 2.507e-15)
       b2 =  -1.935e-16  (-2.373e-15, 1.986e-15)
       a3 =   6.724e-16  (-1.794e-15, 3.138e-15)
       b3 =  -9.616e-16  (-3.141e-15, 1.218e-15)
       a4 =           1  (1, 1)
       b4 =  -2.664e-15  (-7.967e-15, 2.638e-15)
       w =       0.5236  (0.5236, 0.5236)
```



Claramente o sinal não pode ser reconstruído, por isso a série de fourier retorna uma senóide de frequência diferente. Não importa o grau da aproximação esta situação vai se manter.

21 Estimativas de Comportamento

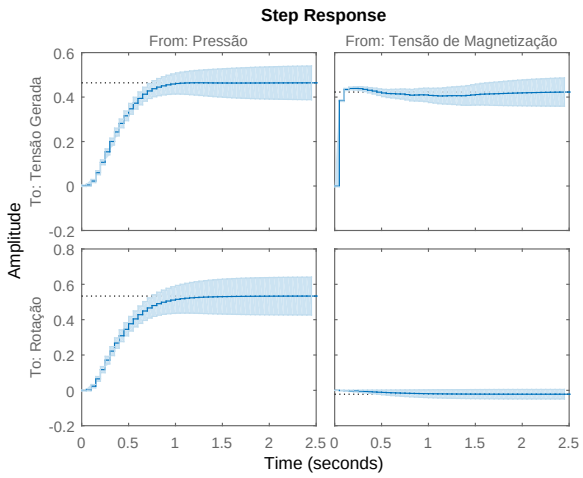
Vamos ver agora como lidar com sistemas MIMO e como estimar modelos com

dados quaisquer. No exemplo abaixo vamos carregar dados reais do MATLAB de um motor a vapor em escala. As entradas são a tensão de magnetização do gerador conectado ao eixo de saída do gerador e a pressão imposta no motor. As saídas são a tensão gerada no gerador e a velocidade de rotação do eixo. Os dados foram amostrados a 50ms. Utilizamos uma estimativa da resposta impulsional baseada em probabilidade e estatística:

```
load SteamEng
vapor = iddata([GenVolt,Speed],[Pressure,MagVolt],0.05);
vapor.InputName  = {'Pressão';'Tensão de Magnetização'};
vapor.OutputName = {'Tensão Gerada';'Rotação'};
respimp = impulseest(vapor,50);
```

Podemos plotar os resultados, vemos que o MATLAB automaticamente coloca janelas relacionando entradas e saídas. No exemplo abaixo foi pedido ao MATLAB para mostrar também as regiões de confiança com grau 3, já que é um modelo probabilístico:

```
figure
showConfidence(stepplot(respimp),2)
```



Da mesma forma como estimamos a função de transferência podemos estimar as matrizes de estado. No caso da função de transferência o resultado foi omitido por ser muito extenso, porém aqui vamos exibir as matrizes para melhor entender o sistema:

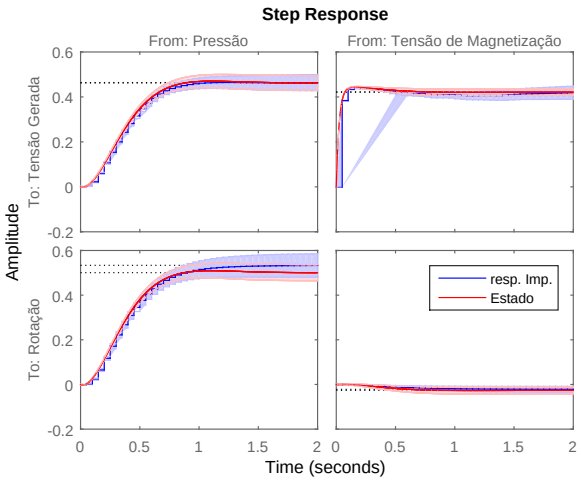
```
espest = ssest(vapor)
```

```
espest =
  Continuous-time identified state-space model:
      dx/dt = A x(t) + B u(t) + K e(t)
       y(t) = C x(t) + D u(t) + e(t)

  A =
            x1       x2       x3
   x1   -39.81    3.896  -0.4863
   x2     1.66   0.3261    6.661
   x3   -3.357   -3.675   -8.094

  B =
             Pressão  Tensão de Ma
   x1     -0.009856         1.116
   x2      -0.07379      -0.04638
   x3        0.5102       0.06856

  C =
                     x1      x2      x3
   Tensão Gerad   16.01   2.205  0.4865
   Rotação        0.129   4.042  0.1007

  D =
                      Pressão  Tensão de Ma
   Tensão Gerad             0             0
   Rotação                  0             0

  K =
       Tensão Gerad       Rotação
   x1        0.1744        0.1011
   x2         1.581         3.112
   x3        -1.235       -0.6663

Parameterization:
   FREE form (all coefficients in A, B, C free).
   Feedthrough: none
   Disturbance component: estimate
   Number of free coefficients: 27
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using SSEST on time domain data "vapor".
Fit to estimation data: [93.92;78.93]% (prediction focus)
FPE: 3.795e-06, MSE: 0.004155
```

Observe o final do resultado, o MATLAB informa o quão bem conseguiu apro-

ximar os dados, no caso obtivemos `[93.92;78.93]%` sendo cada grau de certeza para cada saída.

```
figure
showConfidence(stepplot(respimp,'b',espest,'r',2))
legend('resp. Imp.','Estado')
```



Vemos que as duas aproximações são praticamente equivalentes. Como estamos trabalhando com incertezas se pegarmos menos amostras vamos obter mais erros. Vamos agora considerar apenas as primeiras 250 amostras e ver como a aproximação piora:

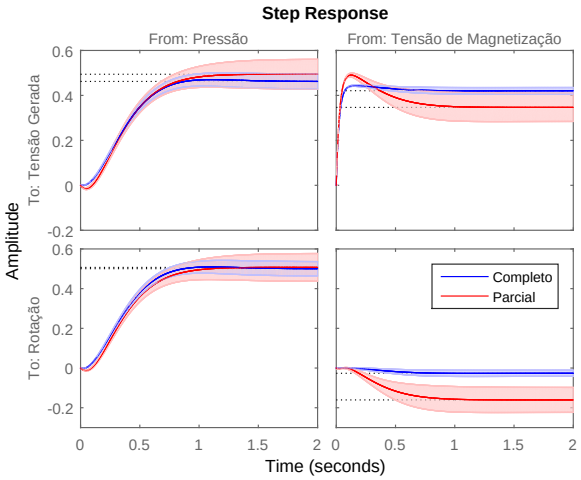
```
espest2 = ssest(vapor(1:250))
```

```
espest2 =
  Continuous-time identified state-space model:
      dx/dt = A x(t) + B u(t) + K e(t)
       y(t) = C x(t) + D u(t) + e(t)

  A =
           x1       x2       x3       x4
   x1   -29.43   -4.561   0.5994   -5.199
   x2   0.4848  -0.8662   -4.101   -2.336
   x3    2.839    5.084   -8.566   -3.855
   x4   -12.13   0.9224    1.818   -34.29

  B =
           Pressão  Tensão de Ma
   x1       0.1033        -1.617
```

```
   x2       -0.3028      -0.09415
   x3        -1.566        0.2953
   x4      -0.04476        -2.681

  C =
                       x1       x2       x3       x4
   Tensão Gerad   -16.39   0.3767  -0.7566    2.808
   Rotação        -5.623    2.246  -0.5356    3.423

  D =
                     Pressão  Tensão de Ma
   Tensão Gerad            0             0
   Rotação                 0             0

  K =
        Tensão Gerad       Rotação
   x1       -0.3555       0.08531
   x2      -0.02308         5.195
   x3         1.526         2.132
   x4         1.787       0.03216

Parameterization:
   FREE form (all coefficients in A, B, C free).
   Feedthrough: none
   Disturbance component: estimate
   Number of free coefficients: 40
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using SSEST on time domain data.
Fit to estimation data: [86.9;74.84]% (prediction focus)
FPE: 4.242e-05, MSE: 0.01414
```

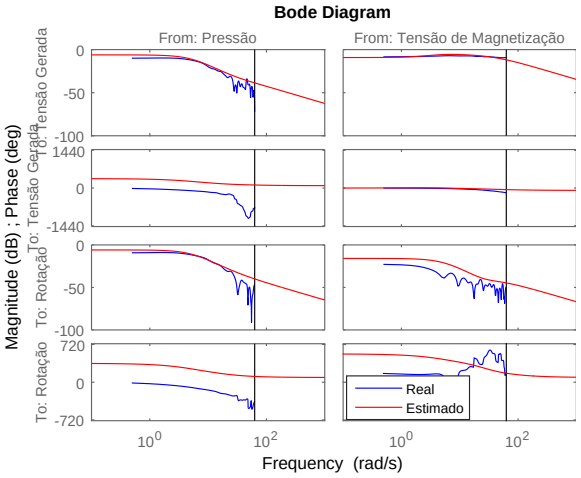
Imediatamente vemos que a precisão piorou, os graus de certeza agora caíram para `[86.9;74.84]%` . Podemos verificar plotando os dois modelos:

```
figure
showConfidence(stepplot(espest,'b',espest2,'r',2))
legend('Completo','Parcial')
```



Analises espectrais também funcionam em sistemas MIMO. Abaixo vamos não só plotar os diagramas de Bode para os dados reais como também para o estimado com dados parciais:

```
espectro = spa(vapor);
figure
bode(espectro,'b',espest2,'r')
h = legend('Real','Estimado');
h.Location = 'best';
```



22 Simulação de um Motor DC

Vamos analisar rapidamente o comportamento de um motor DC. Não vamos entrar na parte de modelagem, existem inúmeras fontes explicando o processo.

Os dados para o exemplo abaixo foram extraídos de um documento aberto pela TU Delft. Este último exemplo irá servir apenas para apresentar diversos comandos, uma vez que fazem parte de matérias avançadas. Caso queira pesquise a raiz deles para aprender a teoria, é muito recomendado e será muito útil para seu conhecimento de controle.

```
s = tf('s');
```

Inserimos os dados do motor e criamos as funções de transferência para velocidade e posição (integrando a primeira).

```
J=0.01;
b=0.1;
K=0.01;
R=1;
L=0.5;

Gv = K/((L*s + R)*(J*s + b) + K^2)
Ga = Gv/s
```

```
Gv =

             0.01
  --------------------------
  0.005 s^2 + 0.06 s + 0.1001

Continuous-time transfer function.


Ga =

               0.01
  -------------------------------
  0.005 s^3 + 0.06 s^2 + 0.1001 s

Continuous-time transfer function.
```

É conveniente dar nomes aos seus I/O's:

```
Gv.InputName = 'Tensão';
Gv.OutputName = 'Velocidade';
Ga.InputName = 'Tensão';
Ga.OutputName = 'Ângulo';
```

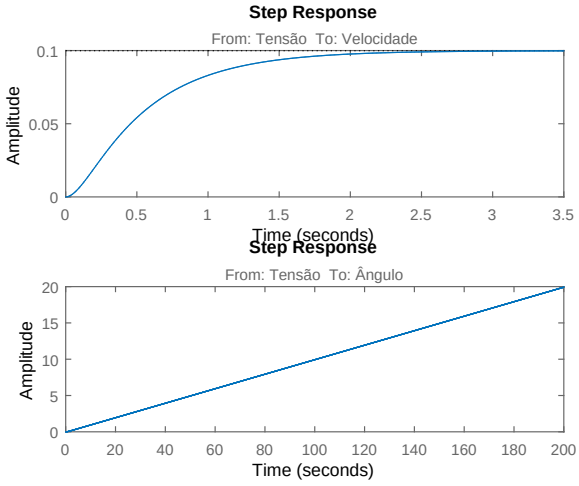
Facilmente podemos ver se os sistemas são estáveis:

```
polos_velocidade = pole(Gv)
polos_angulo = pole(Ga)
```

```
polos_velocidade =

   -9.9975
   -2.0025


polos_angulo =

         0
   -9.9975
   -2.0025
```

Claramente o polo em zero pode introduzir instabilidade, plotamos para checar:

```
figure
subplot(2,1,1)
step(Gv)
subplot(2,1,2)
step(Ga)
```

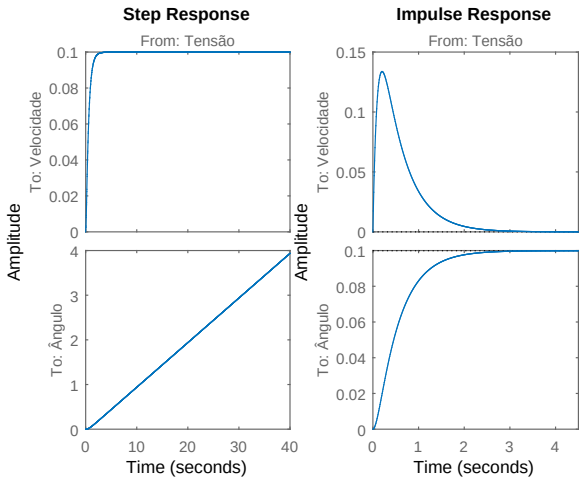


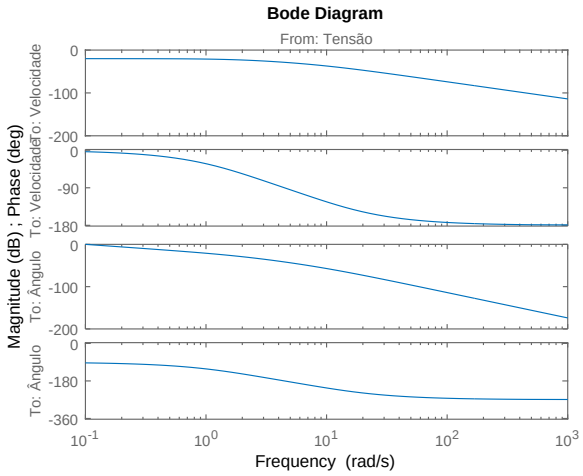
Vemos que quando o tempo vai a infinito, o output também vai, logo o sistema não é BIBO-estável. Para facilitar vamos tornar dois sistemas SISO em um SIMO (*Single Input Multiple Outputs*):

```
G = [Gv; Ga];
```

Agora fica mais fácil analisar os comportamentos:

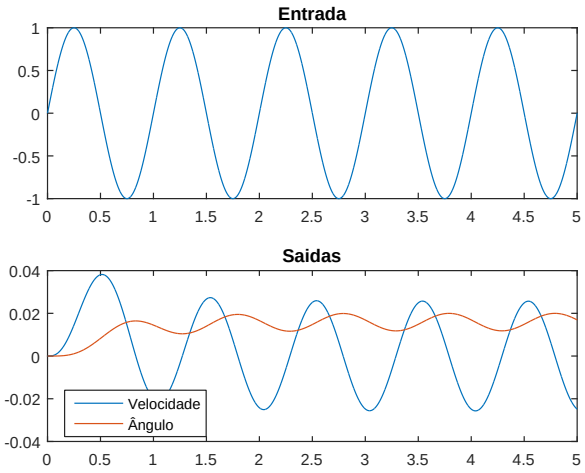
```
figure
subplot(1,2,1)
step(G);
subplot(1,2,2)
impulse(G);
figure
bode(G);
```





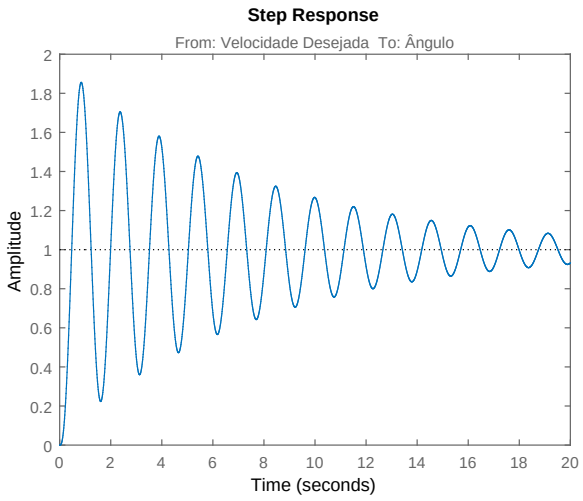
Vamos simular com um sinal arbitrário:

```
t = 0:0.001:5;
u = sin(2*pi*t);
figure
[y,t] = lsim(G,u,t);
subplot(2,1,1)
plot(t,u),title('Entrada');
subplot(2,1,2)
plot(t,y),title('Saidas'),h=legend('Velocidade','Ângulo');
h.Location  = 'best';
```



Como podemos ver nem sempre o comportamento é o mais intuitivo, principalmente em sistemas com grau maior que 2. Vamos implementar um controle proporcional e tentar estabilizar a saída angular:

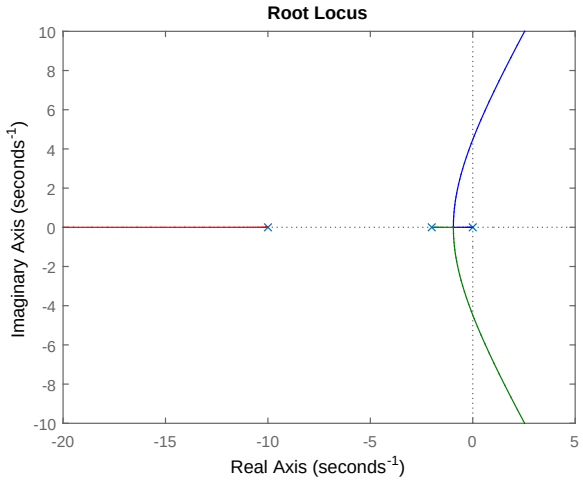
```
Kp = 100;
Gc = feedback(Ga*Kp,1);
Gc.InputName = 'Velocidade Desejada';

figure
step(Gc,0:0.01:20);
```



Claramente o sistema irá se estabilizar pelo único motivo que o motor possui atritos internos modelados pela constante b.

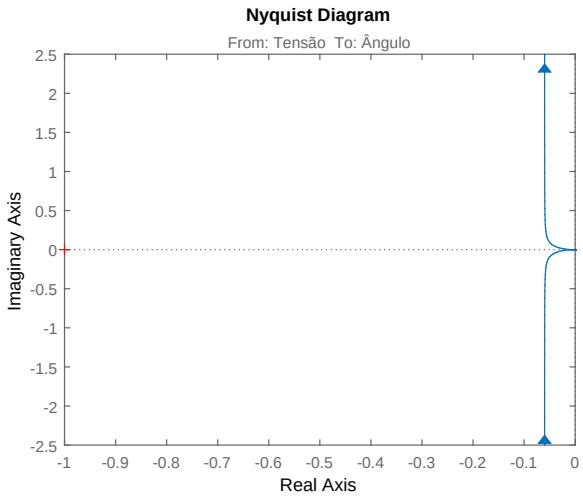
Vamos utilizar alguns métodos de critério de estabilidade para ver como o sistema se comporta para o controle de posição:

```
figure
rlocus(Ga),axis([-20 5 -10 10]);
```



Podemos ver que mesmo com amortecimento, um ganho muito alto irá desestabilizar o sistema. Execute o comando acima e use o cursor para descobrir este ganho, substitua no controle proporcional e veja que de fato desestabiliza o sistema. O diagrama de Nyquist é facilmente plotável para ajudar nestas conclusões:

```
figure
nyquist(Ga)
```



Podemos ainda extrair a forma canônica de Jordan, matriz de observabilidade e controlabilidade:

```
[a,b,c,d] = tf2ss(0.01,[0.005 0.06 0.1001 0]);
[V, Jo] = jordan(a)
Ob = obsv(a,c)
Co = ctrb(a,b)
```

```
V =

         0   99.9500    4.0100
         0   -9.9975   -2.0025
    1.0000    1.0000    1.0000


Jo =

         0         0         0
         0   -9.9975         0
         0         0   -2.0025


Ob =

     0     0     2
     0     2     0
```

```
     2     0     0


Co =

    1.0000  -12.0000  123.9800
         0    1.0000  -12.0000
         0         0    1.0000
```

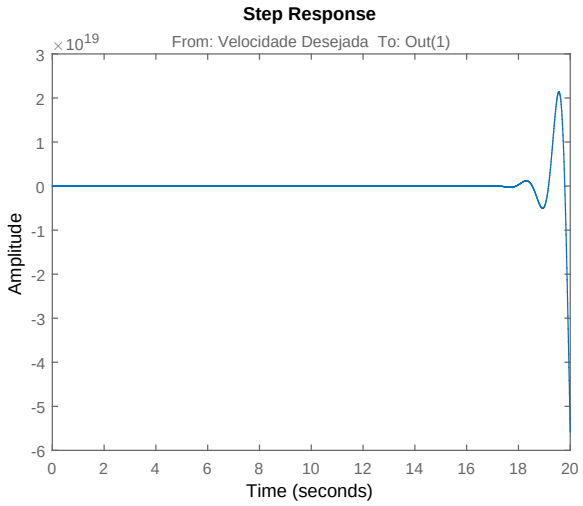
E assim achar o número de estados não observáveis e não controláveis:

```
inob = length(a)-rank(Ob)
inco = length(a)-rank(Co)


inob =

     0


inco =

     0
```

A nossa conclusão é que nosso sistema é completamente controlável e observável para o controle do ângulo. Vamos zerar a constante de amortecimento e ver o que ocorre na mesma janela de tempo de simulação:

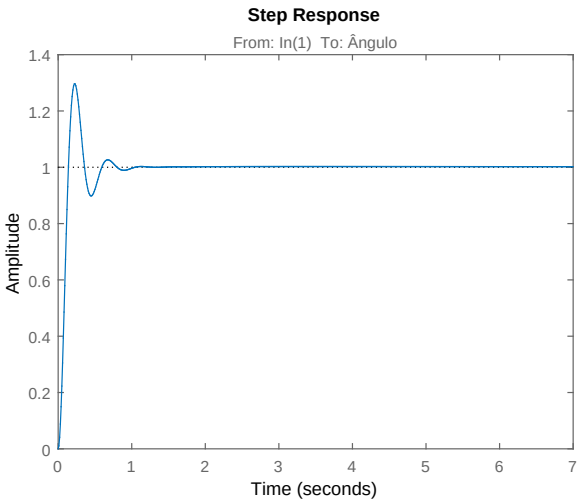
```
b = 0;

Gv2 = K/((L*s + R)*(J*s + b) + K^2);
Ga2 = Gv2/s
Kp = 100;
Gc2 = feedback(Ga2*Kp,1);
Gc2.InputName = 'Velocidade Desejada';

figure
step(Gc2,0:0.01:20);


Ga2 =

               0.01
  -------------------------------
```

```
  0.005 s^3 + 0.01 s^2 + 0.0001 s

Continuous-time transfer function.
```



Vemos que o sistema, de fato, se torna instável. Mesmo que seja estável com atrito o controle proporcional não é viável devido ao longo tempo de assentamento, alto overshoot, muitas oscilações e eventual instabilidade. Vamos implementar um controle PID para atenuar estes efeitos:

```
Kp = 180;
Kd = 110;
Ki = 10;
Gc = pid(Kp,Ki,Kd);
Gm = feedback(series(Gc,Ga),1);
t = (0:0.01:7)';

figure
step(Gm,t)
```



Novamente o controle PID aumentou em muito a qualidade da planta, porém foi necessário analisar a natureza do sistema para determinar parâmetros seguros. Controladores podem ser o quão complexos quisermos, o controlador abaixo foi projetado com a ferramenta `sisotool` do MATLAB. É importante lembrar que embora a resposta desse controlador pareça perfeita, o atuador está sendo muito requisitado e é improvável que esse controle funcione na vida real, ele está aqui apenas para ilustrar a complexidade possível de se atingir com o MATLAB. Foi Feito também um controlador do tipo LQG, este é mais complexo que o PID porém realizável:

```
KL = 3629365359589.16;
ZL = [-4099.1739050971;-9.99749803456527;-2.00250672035499;...
    -0.000239385686210815;-0.000244138720373711;-4094.41313804988+...
    2.75004872395928i;-4094.41313804988-2.75004872395928i];
PL = [-4099.17391633088;0;-5462.9811678135;-0.000244139915105965;...
    -4094.41313243444+2.75005916623674i;-4094.41313243444-...
    2.75005916623674i;-3510.25431148115+1435.21774068424i;...
    -3510.25431148115-1435.21774068424i];

KG = 14339905.5815134;
ZG = [-0.988430249319106;-8.4895499071315+3.020614333216951;...
    -8.4895499071315-3.020614333216951];
PG = [-138.516201360636;0;-63.3666467558354+116.433332601832i;...
    -63.3666467558354-116.433332601832i];

CLS = zpk(ZL,PL,KL);
CLQG = zpk(ZG,PG,KG);
```

```
Gi = feedback(series(CLS,Ga),1);
Glqg = feedback(series(CLQG,Ga),1);
figure
step(Gm,Gi,Glqg,1.4)
legend('Controle PID','Controle por Loop Shaping de ordem 8'...
    ,'Controle LQG')
```