Tutorial para uso de Julia

Vítor H. Nascimento, abr. 2022

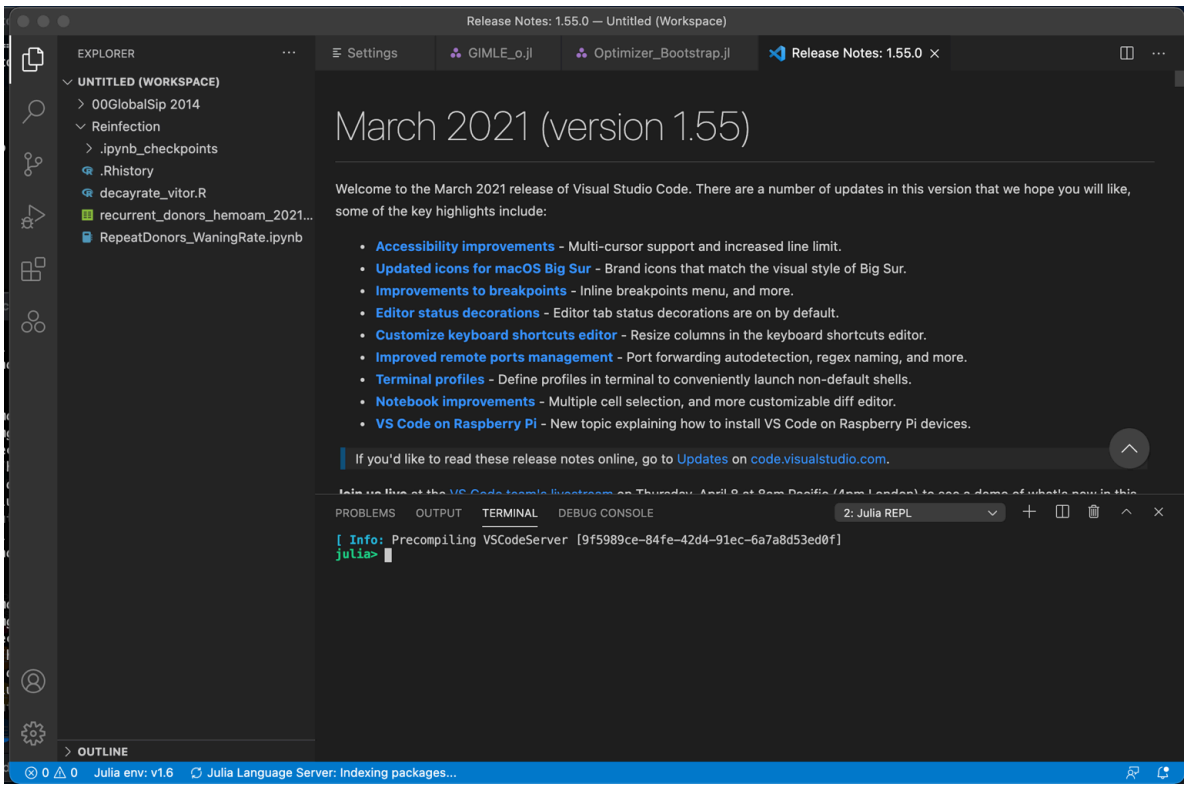
Julia é uma linguagem criada recentemente, com o objetivo de criar código que rode aproximadamente tão rápido quanto em C, mas com a possibilidade de uso interativo e a facilidade de tratamento de estruturas complexas do *Matlab* ou do *Python*. Por ser *software* aberto, é uma alternativa muito interessante ao *Matlab* (que é caro) e ao *Octave* (que tende a ser muito lento).

Assim como *Matlab* e *Python*, *Julia* permite que você use objetos mais abstratos como vetores e matrizes de maneira bem natural, facilitando bastante a programação. No entanto, ao contrário de *Matlab* e *Python*, *Julia* consegue uma velocidade de processamento próxima da velocidade de *C* (desde que alguns cuidados sejam tomados).

Instalação

Para instalar *Julia*, há diversas opções. você pode baixar o programa diretamente da Página principal da linguagem *Julia*, e usar um editor como o Code para criar um ambiente de desenvolvimento (IDE) no seu próprio computador.

Para instalar, é fácil: baixe o arquivo executável de acordo com o seu sistema operacional, e você já poderá programar. Com isso você já pode usar *Julia* através da linha de comando, e escrever programas usando um editor de texto qualquer. Vários editores de texto (como por exemplo *Emacs* e *Vim*) têm pacotes específicos para *Julia*. Uma boa opção é o Code, que tem uma extensão bem completa para *Julia*, incluindo recursos para *debugging*. Para instalar a extensão, basta você entrar na aba *Extensions* na barra do lado esquerdo da janela principal do *Code*, e buscar o pacote *julia* (depois da instalação, veja na barra de status na parte de baixo da tela do *Code* que a instalação de *Julia* vai ser indexada, o que vai demorar um pouco. Ao terminar, você poderá abrir um terminal para usar os comandos de *Julia*, usando `CTRL-SHIFT-P` (`CMD-SHIFT-P` no Mac) e escolhendo `Julia: Start REPL`.



Além do *Code*, você também pode usar *Julia* usando *notebooks* do *Jupyter* (para isso instale o pacote *IJulia*), ou *notebooks* do *Pluto*. Para usar o *Jupyter*, você pode tanto usar uma instalação prévia do *Python* com os pacotes de *Jupyter* já instalados, ou aproveitar a opção de instalar tudo diretamente do *IJulia*, usando os comandos (dentro de uma janela do *Julia* - na listagem abaixo foi usado o comando `]` para entrar no modo de instalação de pacotes)

```
julia> ]
pkg> add IJulia
julia> using IJulia
julia> notebook()
```

O programa vai então perguntar se você deseja instalar o *Jupyter*. Note que após a instalação, você não precisa mais rodar o comando `add IJulia`.

Para instalar o *Pluto*, use

```
julia> ]
pkg> add Pluto
julia> using Pluto
```

Ajuda

A documentação geral da linguagem está aqui. *Julia* também tem diversos pacotes de funções que podem ser instalados pelo usuário, veja abaixo na seção Gráficos. Uma lista de pacotes com a documentação correspondente você encontra no endereço.

Na linha de comando, você pode obter ajuda para qualquer comando digitando `?Nome_do_comando`

Se você digitar `?` no início da linha de comando, *Julia* entra no modo de help, e o cursor muda para

```
help?>
```

digite o nome de um comando e virá uma mensagem de ajuda.

Também é possível rodar um comando do `shell` (caso você use Linux ou Mac), digitando `;` no início da linha de comando. O cursor muda para

```
shell>
```

Atenção, no entanto, que caracteres-curinga como `*` não funcionam nesse shell simples.

Comandos básicos e entrada de dados

Tanto a linha de comando em *Julia* (o *REPL* (read-eval-print loop) quanto o *Jupyter* ou o *Pluto* permitem que você use a linguagem de maneira interativa. Este arquivo que você está lendo foi criado no *Jupyter*, assim como a maior parte dos exemplos das aulas. Você pode criar variáveis e usá-las diretamente, como nos comandos a seguir:

In [2]:
```
n=2
```

Out[2]:
```
2
```

In [3]:
```
x=2.5
```

Out[3]:
```
2.5
```

Se você terminar a linha com um ponto-e-vírgula, o resultado da operação não é mostrado. Isso é útil para definições, principalmente de vetores e matrizes:

In [4]:
```
y=3.5;
```

Uma característica muito interessante de *Julia* é a facilidade de trabalhar com variáveis de diferentes tipos: inteiros, racionais, reais (ponto flutuante), complexos, vetores, matrizes, etc. Em muitas situações você pode trabalhar com esses diferentes tipos de variáveis como faria com papel e lápis (o que também pode ser feito em *Matlab* e *Python*), e *Julia* se encarrega de transformar os formatos como necessário para você (na maior parte das vezes, veremos algumas exceções e o motivo para elas mais à frente).

As quatro operações normais são obtidas com os operadores `+`, `-`, `*` e `/`. Exponenciação é obtida com o operador `^`.

In [5]:
```
z=n*x+n
```

Out[5]:
```
7.0
```

Em *Julia* podemos definir variáveis inteiras, escrevendo números sem ponto decimal, como

In [6]:
```
m=7
```

Out[6]:
```
7
```

Podemos definir variáveis reais (ponto flutuante), escrevendo o ponto decimal (seguido ou não de zero), como

In [7]:
```
z=7.0
```

Out[7]:
```
7.0
```

As quatro operações usam os comandos usuais, por exemplo

In [8]:
```
8*n+11
```

Out[8]:
```
27
```

Veja que, como todos os operandos são números inteiros, o resultado também é um número inteiro (repare que a saída `27` não tem ponto).

Em *Julia* não é necessário escrever explicitamente o `*` ao se multiplicar uma constante por uma variável como em `8n`.

In [9]:
```
8n
```

Out[9]:
```
16
```

In [10]:
```
2.5n
```

Out[10]:
```
5.0
```

Isso só vale no caso de se colocar um número constante na frente de uma variável, como `8n`, `2.5x`, etc. O símbolo `n8` representa uma variável diferente:

In [11]:
```
n8=11
```

Out[11]:
```
11
```

In [12]:
```
8n+n8
```

Out[12]:
```
27
```

Outra característica divertida de *Julia* é que podemos definir variáveis com letras gregas:

In [18]:
```
μ=0.5
```

Out[18]:
```
0.5
```

In [19]:
```
2μ
```

Out[19]:
```
1.0
```

As letras gregas podem ser obtidas digitando-se o nome da letra (em inglês), precedido por `\` (são os códigos usados em LaTeX) e seguido por `TAB`: `\alpha[TAB]`, `\beta[TAB]`, `\gamma[TAB]`. Letras maiúsculas gregas são obtidas como `\Gamma[TAB]`, por exemplo.

In [20]:
```
α=1.5
```

Out[20]:
```
1.5
```

In [21]:
```
β=0.7
```

Out[21]:
```
0.7
```

In [22]:
```
Γ=2/3
```

Out[22]:
```
0.6666666666666666
```

In [23]:
```
α+Γ*β
```

Out[23]:
```
1.9666666666666666
```

O símbolos `π` e `pi` são pré-definidos, e representam a mesma coisa:

In [24]:
```
π
```

Out[24]:
```
π = 3.1415926535897...
```

In [25]:
```
pi
```

```
Out[25]: π = 3.1415926535897...
```

```
In [26]: cos(π/3+2π)
```

```
Out[26]: 0.5000000000000006
```

O resultado da última operação é guardado na variável `ans`:

```
In [27]: 8n+11
```

```
Out[27]: 27
```

```
In [28]: ans/3
```

```
Out[28]: 9.0
```

Repare que o resultado de uma divisão usando `/` é um número em ponto flutuante, mesmo quando o dividendo e o divisor são inteiros e o resultado poderia ser representado exatamente como um inteiro. Para fazer divisão entre inteiros, você pode usar o operador `÷` (que você obtém escrevendo `\div` seguido por `TAB`).

```
In [29]: (8n+11)÷3
```

```
Out[29]: 9
```

Repare que, se o primeiro número não for divisível pelo segundo, `÷` fornece a parte inteira da divisão:

```
In [30]: 8÷3
```

```
Out[30]: 2
```

Por outro lado, `/` fornece o resultado em ponto flutuante:

```
In [31]: 8/3
```

```
Out[31]: 2.6666666666666665
```

O resto da divisão inteira é obtido com a função `mod` ou o operador `%`:

```
In [33]: 10 % 3
```

```
Out[33]: 1
```

```
In [34]: mod(10,3)
```

```
Out[34]: 1
```

Números racionais

Além de números inteiros e reais, podemos trabalhar também com frações (números racionais) diretamente em *Julia*, usando `//` em vez de `/`:

In [35]:
```
q=8//3
```

Out[35]:
```
8//3
```

Aqui não foi feita nenhuma conta - a variável q representa uma fração, e podemos fazer operações com frações normalmente. Por exemplo,

In [36]:
```
q+5//6
```

Out[36]:
```
7//2
```

Veja que o resultado já vem simplificado. Podemos usar o resultado como um número qualquer:

In [37]:
```
ans*n
```

Out[37]:
```
7//1
```

Repare que o resultado continua sendo uma fração. Na verdade, ao fazer uma conta com números de tipos diferentes, *Julia* "promove" os tipos mais restritivos para o tipo mais geral. Por exemplo, como os racionais contêm os inteiros, a soma de um racional com um inteiro resulta em um racional:

In [38]:
```
ans+1
```

Out[38]:
```
8//1
```

Já a soma de um racional (ou de um inteiro) com um número em ponto flutuante resulta um número em ponto flutuante:

In [39]:
```
ans+1.0
```

Out[39]:
```
9.0
```

Obviamente, não faz sentido escrever um número racional em que o numerador ou o denominador não seja inteiro (quer dizer, com numerador ou denominador em ponto flutuante) - se você fizer isso, você recebe uma mensagem de erro:

In [40]:
```
2.0//3
```

```
MethodError: no method matching //(::Float64, ::Int64)
Closest candidates are:
  //(::Integer, ::Integer) at ~/julia/usr/share/julia/base/rational.jl:62
  //(::Rational, ::Integer) at ~/julia/usr/share/julia/base/rational.jl:6
4
  //(::Complex, ::Real) at ~/julia/usr/share/julia/base/rational.jl:78
  ...

Stacktrace:
 [1] top-level scope
   @ In[40]:1
 [2] eval
   @ ./boot.jl:373 [inlined]
 [3] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::S
tring, filename::String)
   @ Base ./loading.jl:1196
```

Vamos explicar melhor como entender as mensagens de erro em outro capítulo mais à frente.

Números complexos

Também podemos definir números complexos usando o número pré-definido `im` (a unidade imaginária):

In [41]:
```
z=1+2im
```

Out[41]:
```
1 + 2im
```

In [42]:
```
z^2
```

Out[42]:
```
-3 + 4im
```

In [43]:
```
im^2
```

Out[43]:
```
-1 + 0im
```

Você pode fazer qualquer operação habitual com números complexos. Por exemplo, o módulo do número z é

In [44]:
```
abs(z)
```

Out[44]:
```
2.23606797749979
```

Lembrando que

$$e^{j\pi/3} = \cos(\pi/3) + j\operatorname{sen}(\pi/3)\,:$$

In [45]:
```
z1=exp((π/3)im)
```

Out[45]:
```
0.5000000000000001 + 0.8660254037844386im
```

Também podemos obter as partes real e imaginária, e o ângulo (em radianos) de um número complexo com os comandos

In [46]:
```
real(z1)
```

Out[46]:
```
0.5000000000000001
```

In [47]:
```
imag(z1)
```

Out[47]:
```
0.8660254037844386
```

In [48]:
```
angle(z1)
```

Out[48]:
```
1.0471975511965976
```

Podemos converter o ângulo para graus facilmente:

In [49]:
```
ans*180/π
```

Out[49]:
```
59.99999999999999
```

Vetores e matrizes

A grande inovação feita pelo *Matlab* foi a capacidade de trabalhar com vetores e matrizes de maneira natural, sem a necessidade de se escrever explicitamente laços e subrotinas para toda operação. Essa ideia foi usada também em *Python* na biblioteca *NumPy*, e em *Julia*. No caso de *Julia* podemos definir vetores e matrizes com uma sintaxe semelhante à do *Matlab*, usando `[` e `]` para marcar o início e o final do vetor ou matriz, e `;` para definir uma nova linha. Assim, por exemplo

In [50]:
```
a=[1 2 3;4 5 6]
```

Out[50]:
```
2×3 Matrix{Int64}:
 1  2  3
 4  5  6
```

resulta uma matriz de duas linhas e três colunas com elementos inteiros. Podemos acessar os elementos de `a` como a seguir:

In [51]:
```
a[1,2]
```

Out[51]:
```
2
```

In [52]:
```
a[2, 3]
```

Out[52]:
```
6
```

Preste atenção no fato que o primeiro elemento de cada linha ou coluna é indexado pelo número 1 (ao contrário de *C* e *Python*, em que o primeiro elemento é indexado por 0). Assim, o primeiro elemento de `a` é dado por `a[1,1]`:

In [53]:
```
a[1,1]
```

Out[53]:
```
1
```

Veja que *Matlab* usa parêntesis para acessar os elementos de um vetor ou matriz: o comando anterior em *Matlab* seria `a(1,1)`.

Outra inovação muito interessante e útil feita pelo *Matlab*, e que *Julia* também usa, é a notação ":" para escolher pedaços de matrizes e vetores. Para ver como essa ideia funciona, considere a matriz com números aleatórios

In [54]:
```
b=randn(3,5)
```

Out[54]:
```
3×5 Matrix{Float64}:
  2.60256    0.437981  -0.129639   1.88916    2.58672
  0.326565   0.152915   1.01963   -0.535376  -0.676528
 -0.530434  -1.60478   -0.152851   0.410814  -0.560158
```

Se quisermos tomar o bloco composto pelos elementos

$$\begin{bmatrix} b[1,2] & b[1,3] & b[1,4] \\ b[2,2] & b[2,3] & b[2,4] \end{bmatrix},$$

podemos usar o comando

In [55]:
```
b[1:2,2:4]
```

Out[55]:
```
2×3 Matrix{Float64}:
 0.437981  -0.129639   1.88916
 0.152915   1.01963   -0.535376
```

O operador ":" permite criar sequências de números. Assim, "2:4" é uma lista com os números 2, 3, 4

In [56]:
```
n=2:4
```

Out[56]:
```
2:4
```

In [57]:
```
n[1]
```

Out[57]:
```
2
```

In [58]:
```
n[2]
```

Out[58]:
```
3
```

In [59]:
```
n[3]
```

Out[59]:
```
4
```

Há uma outra forma do operador `:`, em que se acrescenta o passo da sequência - `1:2:5` é a lista com os números 1, 3, 5

In [60]:
```
n1=1:2:5
```

Out[60]:
```
1:2:5
```

In [61]:
```
n1[1]
```

Out[61]:
```
1
```

In [62]:
```
n1[2]
```

Out[62]:
```
3
```

In [63]:
```
n1[3]
```

Out[63]:
```
5
```

In [64]:
```
b[1:2:3,n1]
```

Out[64]:
```
2×3 Matrix{Float64}:
  2.60256   -0.129639   2.58672
 -0.530434  -0.152851  -0.560158
```

Para pegar uma linha ou uma coluna inteira de uma matriz, você pode usar `:` sozinho, como nos exemplos abaixo

In [58]:
```
b[:,1] # Primeira coluna de b
```

Out[58]:
```
3-element Vector{Float64}:
 -1.1012317707130723
  0.062111167498386334
  0.9152992030405941
```

In [65]:
```
b[2,:] # Segunda linha de b
```

Out[65]:
```
5-element Vector{Float64}:
  0.3265652058720791
  0.1529152350974509
  1.0196276091964787
 -0.5353759505328688
 -0.6765281756710261
```

Note que, como o formato-padrão para vetores em *Julia* é de vetores-coluna, `b[2,:]` resulta em um vetor-coluna. Se você precisa da segunda linha de b em formato de vetor-linha, use `b[[2],:]`.

In [66]:
```
b[[2],:]
```

Out[66]:
```
1×5 Matrix{Float64}:
 0.326565  0.152915  1.01963  -0.535376  -0.676528
```

O resultado é diferente porque `2` é um número inteiro, enquanto que `[2]` é um vetor com um único elemento.

O passo usado com o operador `:` pode ser um número qualquer, inclusive negativo. Por exemplo,

In [67]:
```
x=0:0.1:2
```

Out[67]:
```
0.0:0.1:2.0
```

In [68]:
```
x[1]
```

Out[68]:
```
0.0
```

In [69]:
```
x[2]
```

Out[69]:
```
0.1
```

In [70]:
```
x[3]
```

Out[70]:
```
0.2
```

In [71]:
```
ninv=5:-1:1
```

Out[71]:
```
5:-1:1
```

In [72]:
```
ninv[1]
```

Out[72]:
```
5
```

In [73]:
```
ninv[2]
```

Out[73]:
```
4
```

A variável `end` pode ser usada apenas ao se acessar os elementos de um vetor ou matriz, e representa o último elemento da dimensão correspondente (analogamente, `begin` representa o primeiro elemento):

In [74]:
```
b
```

Out[74]:
```
3×5 Matrix{Float64}:
  2.60256    0.437981  -0.129639   1.88916    2.58672
  0.326565   0.152915   1.01963   -0.535376  -0.676528
 -0.530434  -1.60478   -0.152851   0.410814  -0.560158
```

In [75]:
```
b[1,end]
```

Out[75]:
```
2.586715604329813
```

In [76]:
```
b[end,begin]
```

Out[76]:
```
-0.5304340748708503
```

In [77]:
```
b[end,end]
```

Out[77]:
```
-0.5601584109357035
```

Podemos inverter a ordem dos elementos da matriz usando `end:-1:1` (ou `end:-1:begin`):

In [78]:
```
b[1:2,end:-1:begin]
```

Out[78]:
```
2×5 Matrix{Float64}:
  2.58672    1.88916   -0.129639  0.437981  2.60256
 -0.676528  -0.535376   1.01963   0.152915  0.326565
```

In [79]:
```
b[end:-1:1,end:-1:1]
```

Out[79]:
```
3×5 Matrix{Float64}:
 -0.560158   0.410814  -0.152851  -1.60478   -0.530434
 -0.676528  -0.535376   1.01963    0.152915   0.326565
  2.58672    1.88916   -0.129639   0.437981   2.60256
```

Operações com matrizes podem ser feitas usando os operadores aritméticos normalmente:

In [80]:
```
a
```

Out[80]:
```
2×3 Matrix{Int64}:
 1  2  3
 4  5  6
```

In [81]:
```
b=[1 1;2 2]
```

Out[81]:
```
2×2 Matrix{Int64}:
 1  1
 2  2
```

In [82]:
```
b*a
```

Out[82]:
```
2×3 Matrix{Int64}:
  5   7   9
 10  14  18
```

Como acontece com vetores e matrizes, para o produto estar bem definido, o número de colunas da primeira parcela deve ser igual ao número de linhas da segunda parcela, senão o produto não é definido, e você recebe uma mensagem de erro:

In [83]:
```
a*b
```

```
DimensionMismatch("matrix A has dimensions (2,3), matrix B has dimensions
(2,2)")

Stacktrace:
 [1] _generic_matmatmul!(C::Matrix{Int64}, tA::Char, tB::Char, A::Matrix{
Int64}, B::Matrix{Int64}, _add::LinearAlgebra.MulAddMul{true, true, Bool,
Bool})
   @ LinearAlgebra ~/julia/usr/share/julia/stdlib/v1.7/LinearAlgebra/src/
matmul.jl:810
 [2] generic_matmatmul!(C::Matrix{Int64}, tA::Char, tB::Char, A::Matrix{I
nt64}, B::Matrix{Int64}, _add::LinearAlgebra.MulAddMul{true, true, Bool,
Bool})
   @ LinearAlgebra ~/julia/usr/share/julia/stdlib/v1.7/LinearAlgebra/src/
matmul.jl:798
 [3] mul!
   @ ~/julia/usr/share/julia/stdlib/v1.7/LinearAlgebra/src/matmul.jl:302
[inlined]
 [4] mul!
   @ ~/julia/usr/share/julia/stdlib/v1.7/LinearAlgebra/src/matmul.jl:275
[inlined]
 [5] *(A::Matrix{Int64}, B::Matrix{Int64})
   @ LinearAlgebra ~/julia/usr/share/julia/stdlib/v1.7/LinearAlgebra/src/
matmul.jl:153
 [6] top-level scope
   @ In[83]:1
 [7] eval
   @ ./boot.jl:373 [inlined]
 [8] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::S
tring, filename::String)
   @ Base ./loading.jl:1196
```

É possível multiplicar escalares por matrizes e somar ou subtrair matrizes de mesma dimensão normalmente:

In [84]:
```
c=[1 2;3 4]
```

Out[84]:
```
2×2 Matrix{Int64}:
 1  2
 3  4
```

In [85]:
```
d=2b+c
```

Out[85]:
```
2×2 Matrix{Int64}:
 3  4
 7  8
```

Também pode-se multiplicar os elementos de duas matrizes elemento-a-elemento, usando `.*` em vez de `*`. Compare:

In [86]:
```
b*c
```

Out[86]:
```
2×2 Matrix{Int64}:
 4   6
 8  12
```

é um produto normal de matrizes. Por outro lado, o produto elemento-a-elemento resulta

In [87]:
```
b.*c
```

Out[87]:
```
2×2 Matrix{Int64}:
 1  2
 6  8
```

Também é possível usar o operador de exponenciação com matrizes quadradas:

In [88]:
```
b^2
```

Out[88]:
```
2×2 Matrix{Int64}:
 3  3
 6  6
```

Repare que o operador `.^` representa também a exponenciação termo-a-termo:

In [89]:
```
b.^2
```

Out[89]:
```
2×2 Matrix{Int64}:
 1  1
 4  4
```

Vetores são definidos de modo similar:

In [90]:
```
v=[1,2] # Note que você pode usar vírgula ou ponto-e-vírgula para definir
```

Out[90]:
```
2-element Vector{Int64}:
 1
 2
```

In [91]:
```
v1=[2;3]
```

Out[91]:
```
2-element Vector{Int64}:
 2
 3
```

O produto de uma matriz por um vetor é obtido como usualmente:

In [92]:
```
b*v
```

Out[92]:
```
2-element Vector{Int64}:
 3
 6
```

Da maneira como definido acima, `v` é um vetor-coluna. Para definir um vetor-linha, podemos usar espaços entre os elementos:

In [93]:
```
u=[1 2]
```

Out[93]:
```
1×2 Matrix{Int64}:
 1  2
```

Como `u` é um vetor linha, pode multiplicar `b` pela esquerda:

In [94]:
```
u*b
```

Out[94]:
```
1×2 Matrix{Int64}:
 5  5
```

Usado em matrizes ou vetores, o operador `'` realiza a operação de transposição:

In [95]:
```
b'
```

Out[95]:
```
2×2 adjoint(::Matrix{Int64}) with eltype Int64:
 1  2
 1  2
```

Veja que `'` também pode ser usado para criar um vetor linha:

In [96]:
```
v'
```

Out[96]:
```
1×2 adjoint(::Vector{Int64}) with eltype Int64:
 1  2
```

O tipo do resultado (`adjoint`) é diferente por detalhes de implementação que não são importantes neste momento.

O operador `:` gera uma variável do tipo `Range`

In [97]:
```
u=1:2
```

Out[97]:
```
1:2
```

In [98]:
```
typeof(u)
```

Out[98]:
```
UnitRange{Int64}
```

Essas variáveis quase sempre funcionam como se fossem um vetor-coluna. A grande diferença é que o vetor não é criado verdadeiramente, apenas fica indicado para ser calculado quando necessário (o que permite ganhos de economia de memória e eficiência). Para converter para um vetor comum, use

In [99]:
```
uvec = collect(u)
```

Out[99]:
```
2-element Vector{Int64}:
 1
 2
```

No entanto, em quase todas as situações os `Range` podem ser usados como se fossem um vetor comum:

In [100…
```
b*u
```

Out[100]:
```
2-element Vector{Int64}:
 3
 6
```

Multiplicar b pela esquerda agora gera um erro por incompatibilidade de dimensões:

In [101…
```
u*b
```

```
DimensionMismatch("matrix A has dimensions (2,1), matrix B has dimensions
(2,2)")

Stacktrace:
 [1] _generic_matmatmul!(C::Matrix{Int64}, tA::Char, tB::Char, A::Base.Re
shapedArray{Int64, 2, UnitRange{Int64}, Tuple{}}, B::Matrix{Int64}, _add:
:LinearAlgebra.MulAddMul{true, true, Bool, Bool})
   @ LinearAlgebra ~/julia/usr/share/julia/stdlib/v1.7/LinearAlgebra/src/
matmul.jl:810
 [2] generic_matmatmul!(C::Matrix{Int64}, tA::Char, tB::Char, A::Base.Res
hapedArray{Int64, 2, UnitRange{Int64}, Tuple{}}, B::Matrix{Int64}, _add::
LinearAlgebra.MulAddMul{true, true, Bool, Bool})
   @ LinearAlgebra ~/julia/usr/share/julia/stdlib/v1.7/LinearAlgebra/src/
matmul.jl:798
 [3] mul!
   @ ~/julia/usr/share/julia/stdlib/v1.7/LinearAlgebra/src/matmul.jl:302
[inlined]
 [4] mul!
   @ ~/julia/usr/share/julia/stdlib/v1.7/LinearAlgebra/src/matmul.jl:275
[inlined]
 [5] *
   @ ~/julia/usr/share/julia/stdlib/v1.7/LinearAlgebra/src/matmul.jl:153
[inlined]
 [6] *(a::UnitRange{Int64}, B::Matrix{Int64})
   @ LinearAlgebra ~/julia/usr/share/julia/stdlib/v1.7/LinearAlgebra/src/
matmul.jl:63
 [7] top-level scope
   @ In[101]:1
 [8] eval
   @ ./boot.jl:373 [inlined]
 [9] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::S
tring, filename::String)
   @ Base ./loading.jl:1196
```

O número de elementos em um vetor ou `Range` pode ser obtido com o comando `length`. As dimensões de uma matriz podem ser obtidas com o comando `size`:

In [102…
```
length(u)
```

Out[102]:
```
2
```

In [103…
```
size(a)
```

Out[103]:
```
(2, 3)
```

A inversa de uma matriz é obtida com a função *inv*

In [104…
```
inv(c)
```

Out[104]:
```
2×2 Matrix{Float64}:
 -2.0   1.0
  1.5  -0.5
```

In [105…
```
c*inv(c)
```

Out[105]:
```
2×2 Matrix{Float64}:
 1.0           0.0
 8.88178e-16   1.0
```

Para resolver o sistema de equações

$$cx = v$$

é possível usar

$$x = inv(c) * v,$$

mas isso é pouco eficiente (usa um número exageradamente grande de operações). É melhor usar o algoritmo de eliminação de Gauss com o operador `\` :

In [106…]
```
x=c\v
```

Out[106]:
```
2-element Vector{Float64}:
 0.0
 0.5
```

Exceto por erros numéricos, o resultado deve ser igual ao de

In [107…]
```
inv(c)*v
```

Out[107]:
```
2-element Vector{Float64}:
 0.0
 0.5
```

O operador `\` assume que `x` e `v` são vetores-coluna. Caso sejam vetores-linha, e a inversa deva ser multiplicada pela direita, você pode usar `/` :

In [108…]
```
v'/c
```

Out[108]:
```
1×2 adjoint(::Vector{Float64}) with eltype Float64:
 1.0  0.0
```

Isto é equivalente a resolver o sistema

$$c'y = v,$$

como podemos ver com o comando

In [109…]
```
y=c'\v
```

Out[109]:
```
2-element Vector{Float64}:
 1.0
 0.0
```

Para calcular o produto escalar entre dois vetores, você pode usar duas formas. Uma é simplesmente usar a transposta, e fazer

In [110…]
```
v'*v
```

Out[110]:
```
5
```

A outra maneira é usar o operador `\cdot` (obtido digitando-se `\cdot[TAB]`), disponível na biblioteca de Álgebra Linear.

Para carregar uma biblioteca, use o comando `using` :

In [111…
```
using LinearAlgebra
```

In [112…
```
v⋅v
```

Out[112]:
```
5
```

Veja que agora não foi necessário transpor o vetor v.

Aplicação de funções a vetores ou matrizes

Uma outra ideia muito útil do *Matlab* que foi usada em *Python* e *Julia* é a possibilidade de aplicar funções quaisquer a todos os elementos de um vetor. Por exemplo, considere a função `cos()`:

In [113…
```
cos(π/3)
```

Out[113]:
```
0.5000000000000001
```

Vamos calcular a função $\cos(x)$ para os valores $x = 0, \pi/6, \pi/3, \pi/2, 2\pi/3, 5\pi/6, \pi$:

In [114…
```
x=0:π/6:π
```

Out[114]:
```
0.0:0.5235987755982988:3.141592653589793
```

In [115…
```
y=cos.(x)
```

Out[115]:
```
7-element Vector{Float64}:
  1.0
  0.8660254037844387
  0.5000000000000001
  6.123233995736766e-17
 -0.4999999999999998
 -0.8660254037844385
 -1.0
```

Repare que escrevemos `cos.(x)`, não `cos(x)`. Qualquer função que aceite um argumento escalar pode ser aplicada a todos os elementos de um vetor (ou matriz) desde que seja colocado um `.` antes do parêntesis:

In [116…
```
x=0:3
```

Out[116]:
```
0:3
```

In [117…
```
exp.(-x)
```

Out[117]:
```
4-element Vector{Float64}:
 1.0
 0.36787944117144233
 0.1353352832366127
 0.049787068367863944
```

In [118…
```
sqrt.(x)
```

Out[118]:
```
4-element Vector{Float64}:
 0.0
 1.0
 1.4142135623730951
 1.7320508075688772
```

Como além disso podemos usar o operador `.*` para multiplicar dois vetores elemento-a-elemento, podemos aplicar funções mais complicadas a vetores facilmente. O comando seguinte calcula $e^{-x}\sqrt{x}$ para todo elemento do vetor `x`:

In [119…
```
y=exp.(-x) .* sqrt.(x)
```

Out[119]:
```
4-element Vector{Float64}:
 0.0
 0.36787944117144233
 0.19139299302082188
 0.08623373197304565
```

Uma observação importante: se você quiser somar uma constante `cte` a todos os elementos de `y`, pode necessário usar o operador `.+`:

```
cte = 2
y .+ cte
```

é interpretado como somar cte a todos os elementos de `y`.

Por outro lado, `y + cte` resulta em erro, pois `y` é um vetor com 4 elementos, e `cte` é um escalar:

In [120…
```
cte = 2
y .+ cte
```

Out[120]:
```
4-element Vector{Float64}:
 2.0
 2.3678794411714423
 2.191392993020822
 2.0862337319730457
```

In [121…
```
y + cte
```

```
MethodError: no method matching +(::Vector{Float64}, ::Int64)
For element-wise addition, use broadcasting with dot syntax: array .+ sca
lar
Closest candidates are:
  +(::Any, ::Any, ::Any, ::Any...) at ~/julia/usr/share/julia/base/operat
ors.jl:655
  +(::T, ::T) where T<:Union{Int128, Int16, Int32, Int64, Int8, UInt128,
UInt16, UInt32, UInt64, UInt8} at ~/julia/usr/share/julia/base/int.jl:87
  +(::Rational, ::Integer) at ~/julia/usr/share/julia/base/rational.jl:31
1
  ...

Stacktrace:
 [1] top-level scope
   @ In[121]:1
 [2] eval
   @ ./boot.jl:373 [inlined]
 [3] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::S
tring, filename::String)
   @ Base ./loading.jl:1196
```

Gráficos

Há diversas bibliotecas para desenhar gráficos em *Julia*. Uma muito interessante de usar é a bibliteca *Plots*, que define uma interface comum que pode ser usada com diversos geradores de gráficos (*backends*). Algumas opções são *GR*, *PyPlot,* PlotlyJS*.

A biblioteca de funções para desenhar gráficos do *Matlab* é tão boa que foi basicamente clonada em *Python* (a biblioteca *Matplotlib*). E, como *Julia* pode facilmente usar código em *Python*, uma das bibliotecas para gráficos em *Julia* simplesmente usa o pacote do *Python*, através da biblioteca *PyPlot*.

Antes de usar diversos pacotes é necessário baixá-los (alguns, como *LinearAlgebra*, são pré-instalados). Para fazer isso, entre no modo de pacotes digitando `]`. O cursor vai mudar de

```
julia>
```

para

```
(@v1.7) pkg>
```

No modo de pacotes, dê o comando `add Nome_do_pacote`, como

```
pkg> add PyPlot
```

e o pacote desejado será baixado e instalado. Isso demora um pouco.

Também é bom atualizar os pacotes instalados de vez em quando, rodando (no modo de pacotes) o comando

```
pkg> update
```

Atenção: A instalação como descrito acima instala os pacotes do *Python* necessários. Se você já tiver o *Python* instalado no seu computador e quiser aproveitar, antes do comando `add PyPlot` (e antes de entrar no modo de pacotes), dê o comando `ENV["PYTHON"] = "comando completo para chamar o Python"`.

No restante deste documento vamos exemplificar os comandos gráficos usando *PyPlot*. Ao longo do curso apresentaremos diversos exemplos usando o pacote *Plots* com o backend *PlotlyJS*, que tem a vantagem de permitir gráficos interativos no *Jupyter*. O *PyPlot* só gera gráficos interativos diretamente no *REPL*.

In [122…

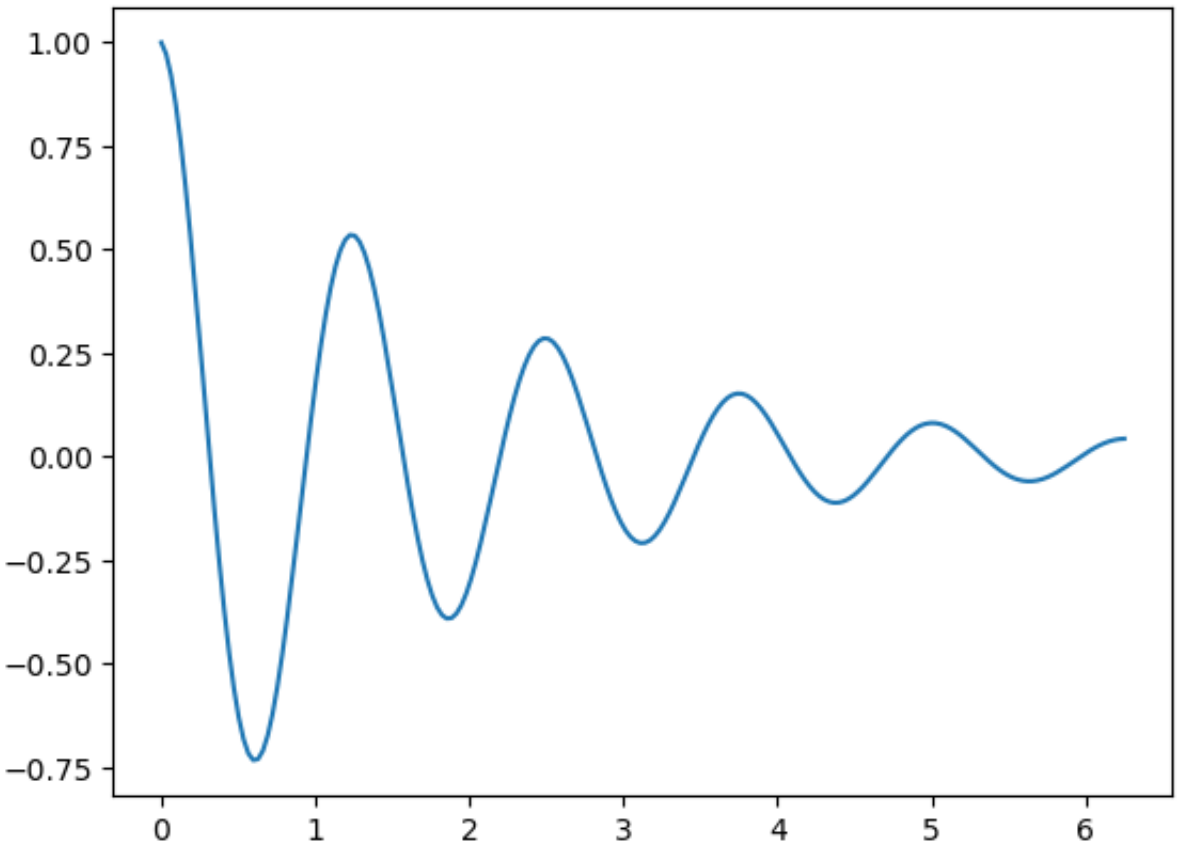
```
using PyPlot
```

Vamos terminar os comandos a seguir com ponto-e-vírgula para suprimir a impressão da saída.

In [123…
```
x=0:π/100:2π;
```

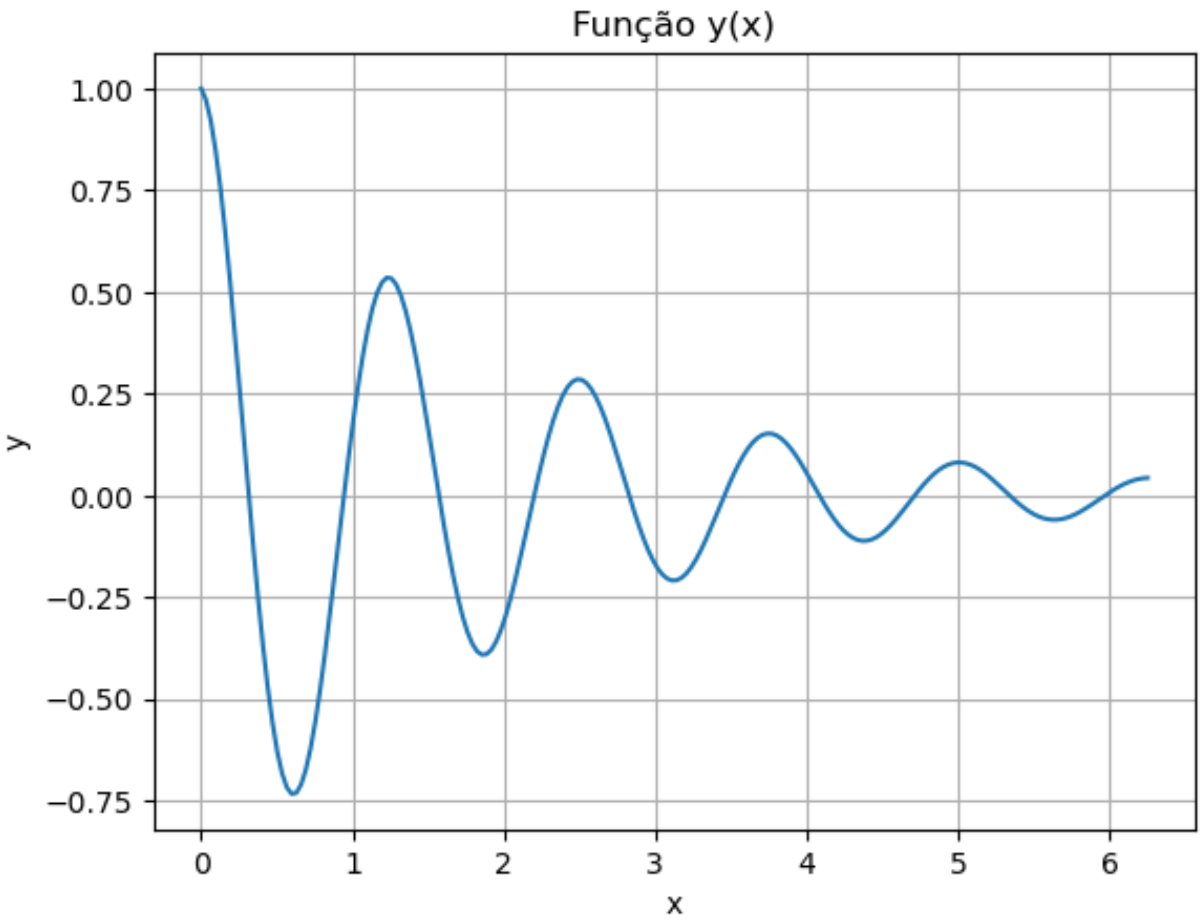
In [124…
```
y=cos.(5x) .* exp.(-0.5x);
```

In [125…
```
plot(x,y);
```



O gráfico pode ser melhorado com diversos comandos: `grid()` desenha uma grade, `xlabel()` acrescenta uma legenda no eixo x, `ylabel()` acrescenta uma legenda no eixo y, `title()` acrescenta um título:

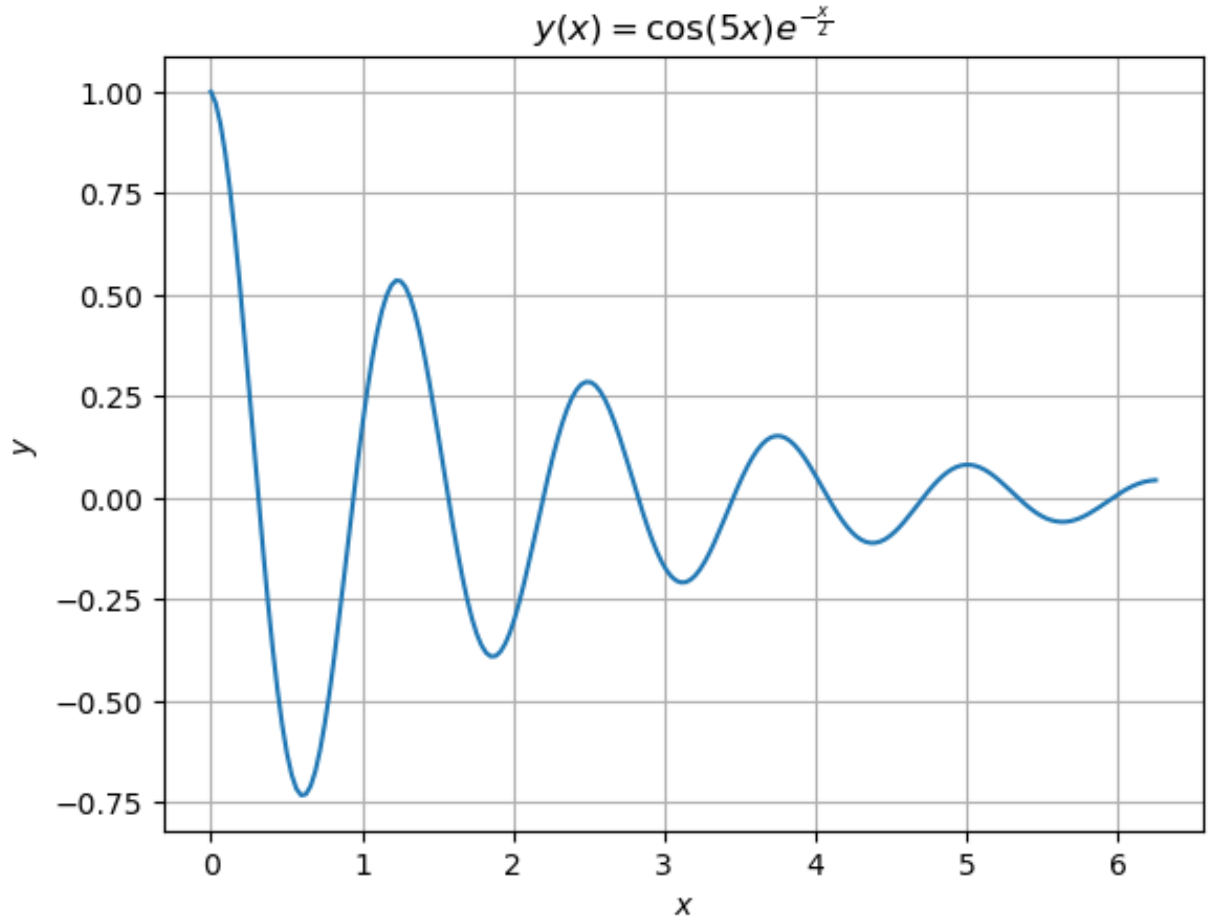
In [126…
```
plot(x,y)
grid()
xlabel("x")
ylabel("y")
title("Função y(x)");
```



Você também pode usar comandos do LaTeX para incrementar as legendas, colocando um `L` antes da *string*:

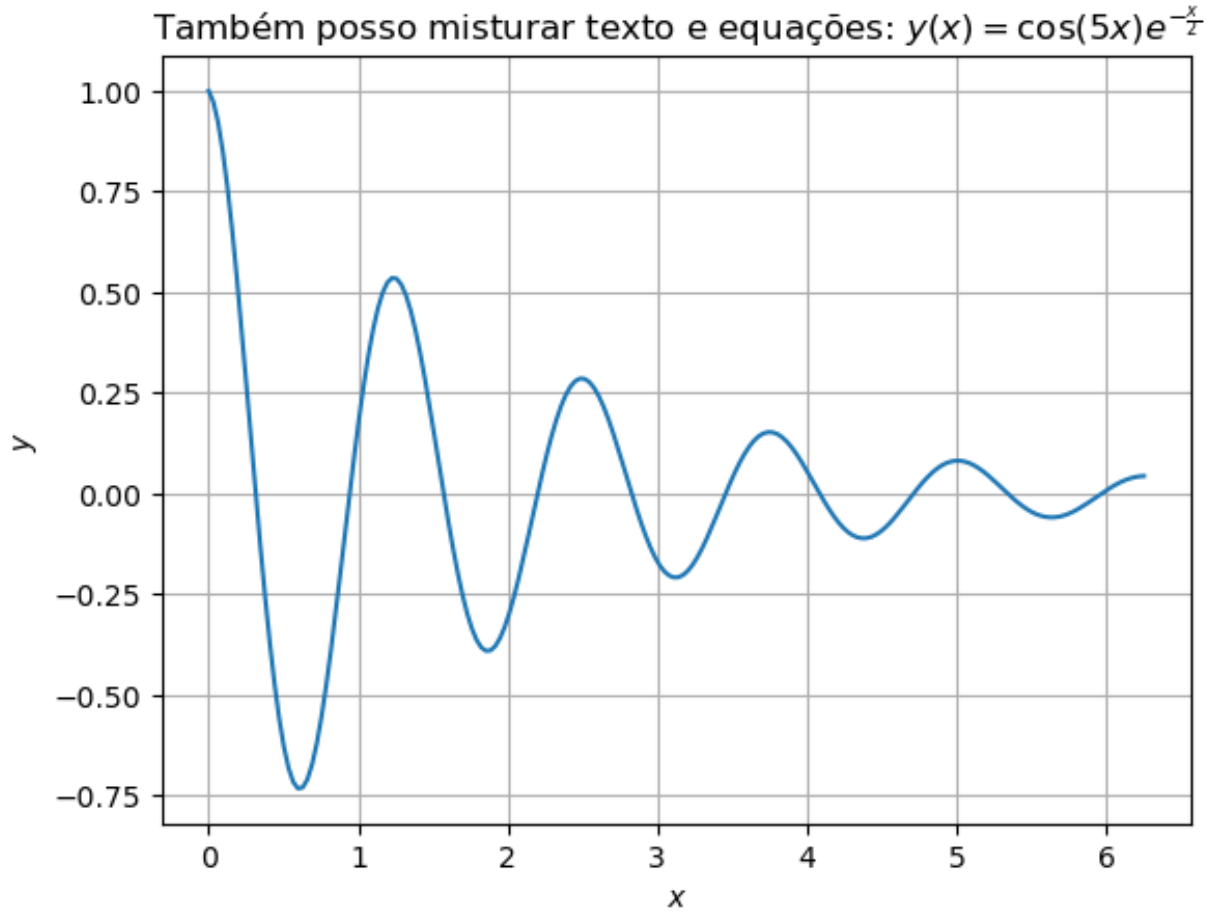
In [127…

```
plot(x,y)
grid()
xlabel(L"x")
ylabel(L"y")
title(L"y(x)=\cos(5x)e^{-\frac{x}{2}}");
```



In [133…

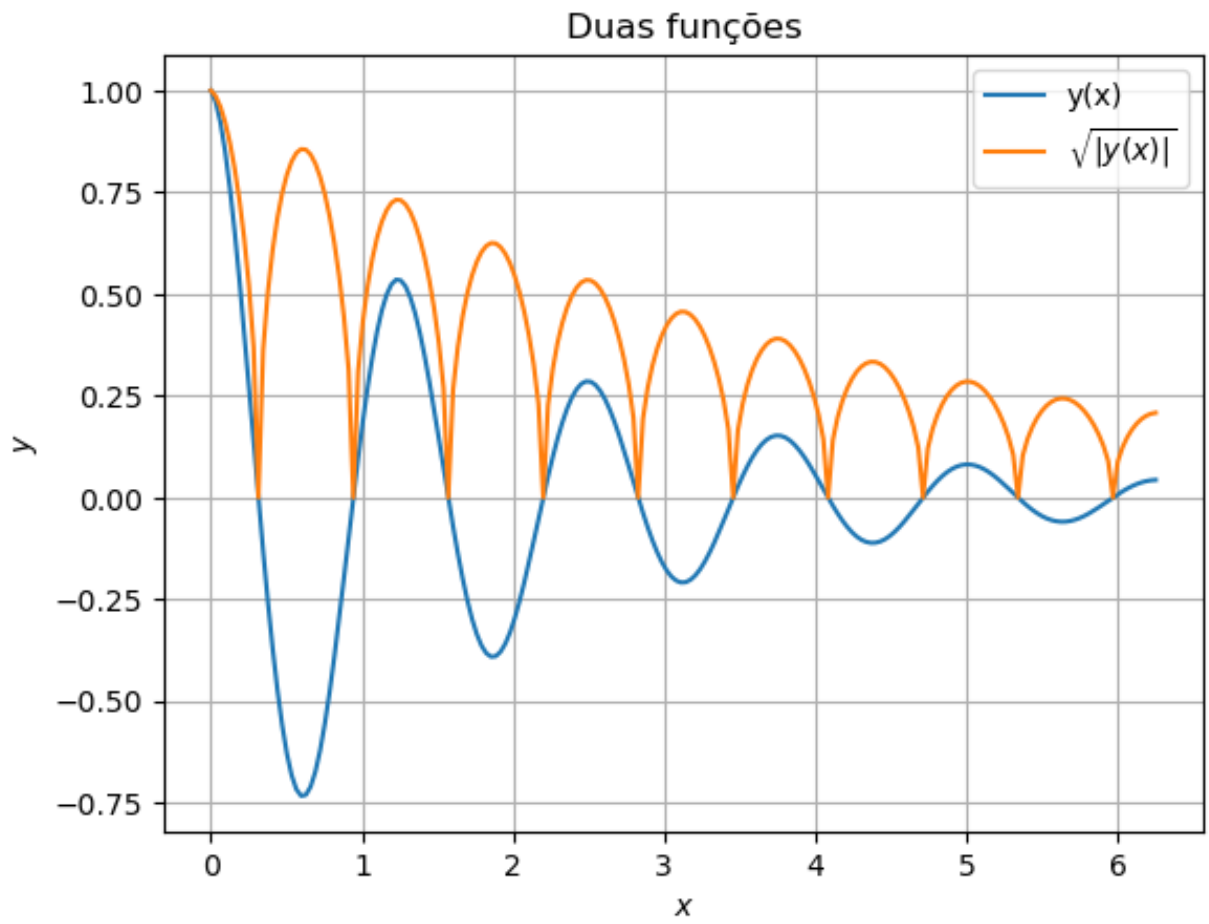
```
plot(x,y)
grid()
xlabel(L"x")
ylabel(L"y")
title(L"Também posso misturar texto e equações: $y(x)=\cos(5x)e^{-\frac{x
```



Finalmente, você pode acrescentar legendas aos gráficos usando a função `legend()`, desde que você coloque uma etiqueta para cada curva, como no exemplo abaixo.

In [134…

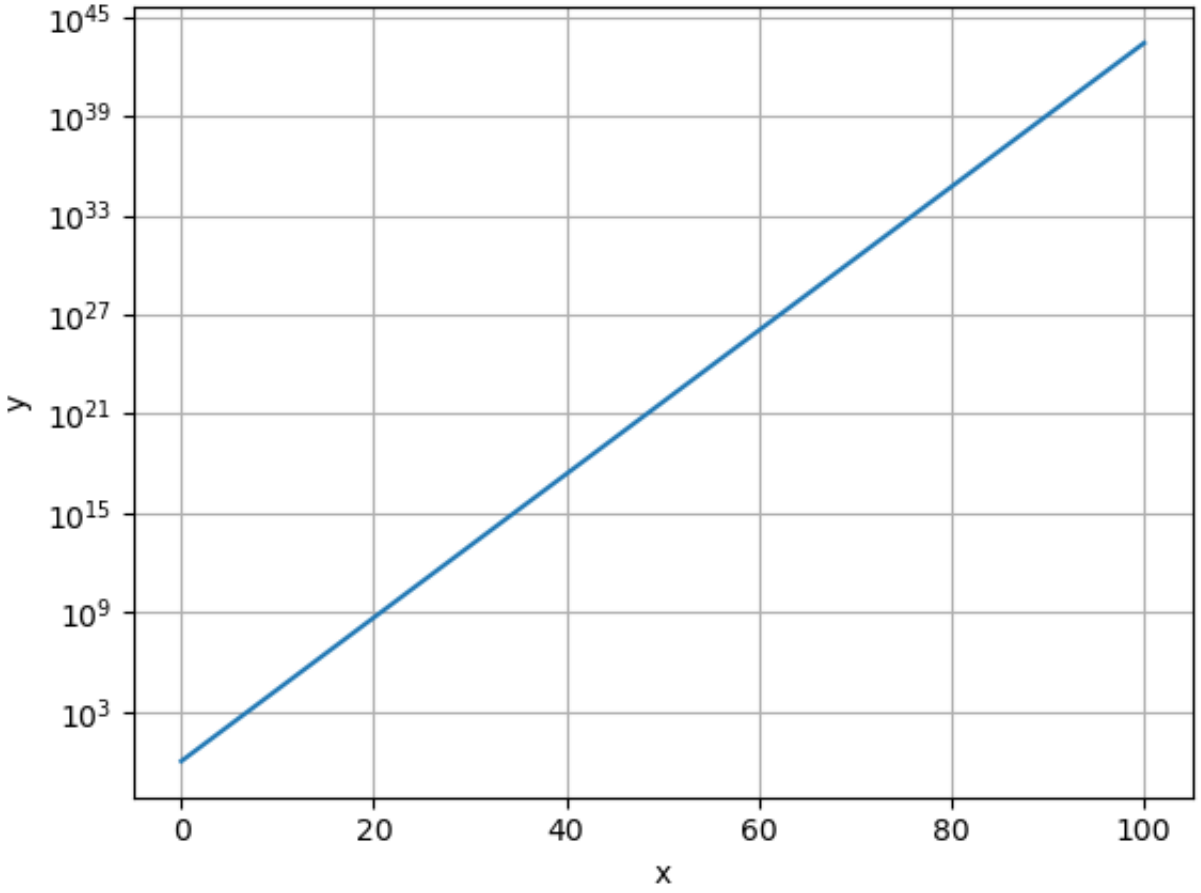
```
z=sqrt.(abs.(y))
plot(x,y, label="y(x)")
plot(x,z, label=L"\sqrt{|y(x)|}")
grid()
xlabel(L"x")
ylabel(L"y")
title("Duas funções")
legend();
```



Como no *Matlab*, há funções para gerar gráficos em escala logarítmica:

In [135…

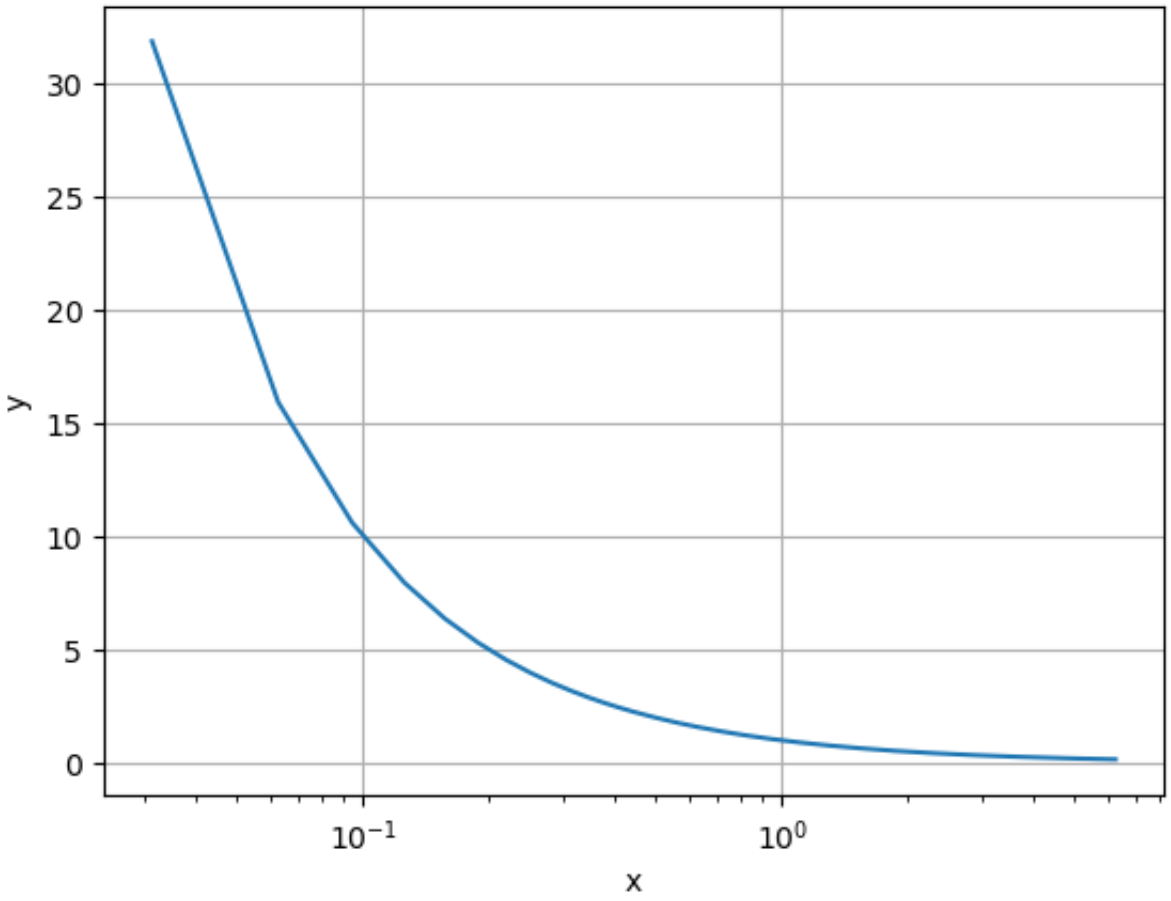
```
x1=10 .^(-2:0.1:2)
y1=exp.(x1)
semilogy(x1,y1);grid()
xlabel("x");ylabel("y");
```



Também é possível gerar gráficos com o eixo x em escala logarítmica usando o comando `semilogx(x,y)`, e com as duas escalas logarítmicas, usando o comando `loglog(x,y)`.

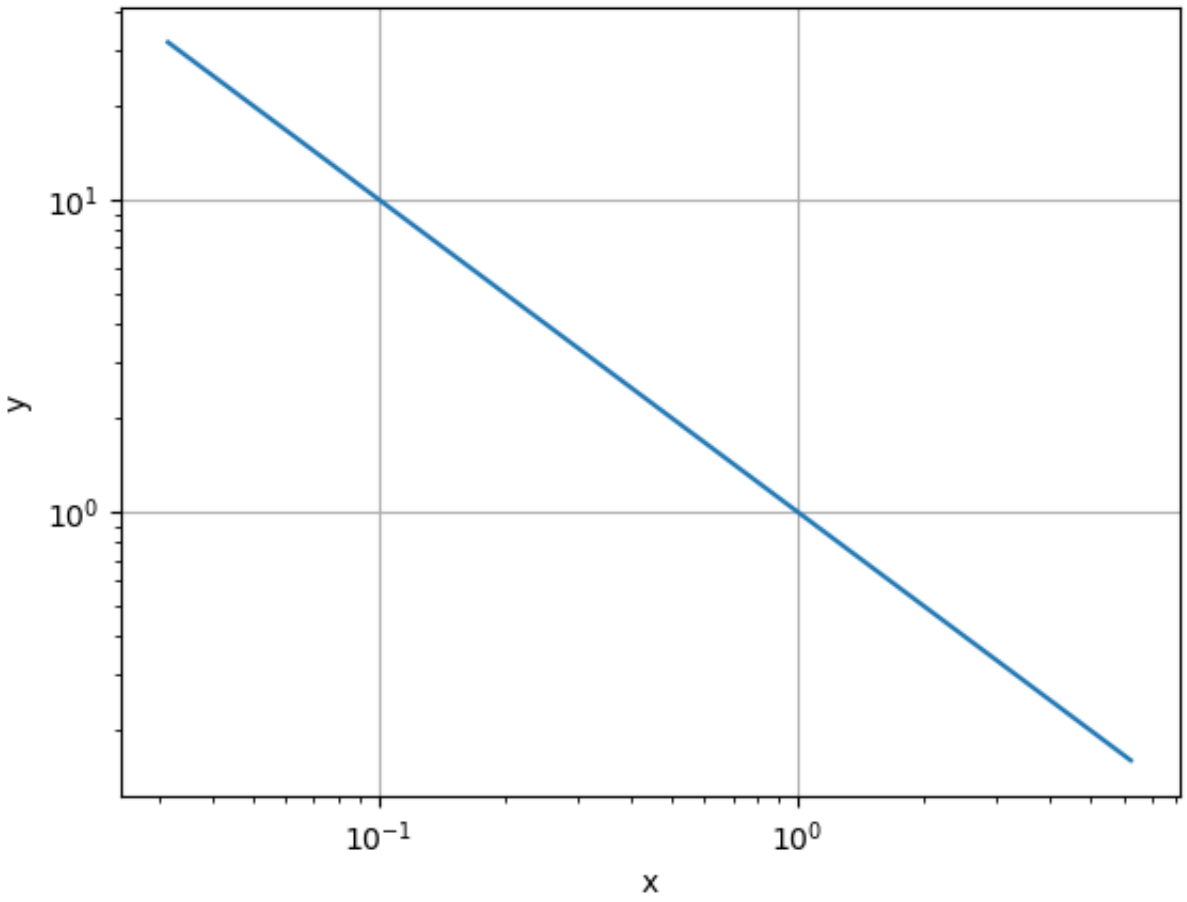
In [136…

```
y2=1 ./x
semilogx(x,y2);grid()
xlabel("x");ylabel("y");
```



In [137…

```
loglog(x,y2);grid()
xlabel("x");ylabel("y");
```



Condições e laços

Condições em *Julia* seguem o padrão if-then-else habitual em várias linguagens, como mostra o exemplo abaixo:

In [138…
```
a=1.0
if a<1
    print("menor")
elseif a==1
    print("igual")
else
    print("maior")
end
```

```
igual
```

In [139…
```
if a!=1
    print("diferente")
else
    print("igual")
end
```

```
igual
```

Repare que testes são feitos sempre sobre variáveis do tipo `Bool`. Por exemplo,

In [140…
```
a=true
if a
    print("Verdadeiro")
else
    print("Falso")
end
```

```
Verdadeiro
```

É possível encadear testes em uma única linha:

In [142…
```
a = 3
if 1 < a < 4
    println("a ∈ [1, 4]")
end
if 1 < a != 3
    println("a > 1 e a ≠ 3")
else
    println("a ≤ 1 ou a = 3")
end
```

```
a ∈ [1, 4]
a ≤ 1 ou a = 3
```

In [143…
```
a = 2
if 1 < a < 4
    println("a ∈ [1, 4]")
end
if 1 < a != 3
    println("a > 1 e a ≠ 3")
else
    println("a ≤ 1 ou a = 3")
end
```

```
a ∈ [1, 4]
a > 1 e a ≠ 3
```

Testes não podem ser feitos diretamente sobre variáveis numéricas, ao contrário do *Matlab*, só sobre variáveis do tipo `Bool`, que assumem os valores `true` ou `false`.

Laços `for` e `while`

Laços em *Julia* podem ser criados com os comandos `for` e `while`, como nos exemplos a seguir:

In [144…
```
soma=0
for i=1:10
    soma += i
end
print(soma)
```

```
55
```

É possível escrever `for i in 1:10` ou `for i ∈ 1:10` (`∈` é obtido com `\in[TAB]`).

In [145…
```
soma=0; i=0
while i<=10
    soma += i
    i+=1
end
print(soma)
```

```
55
```

Também é possível definir laços sobre vetores:

In [146…
```
a=[1 3 2 7 -8]
soma=0
for i in a
    soma+=i^2
end
print(soma)
```

```
127
```

Repare apenas que em *Julia* há formas de se calcular os resultados dos exemplos assim de maneira mais compacta - note que a maneira compacta não é necessariamente a mais rápida em *Julia*, mas torna o código mais simples de ler e escrever:

In [147…
```
sum(a.^2)
```

Out[147]:
```
127
```

In [148…
```
sum(1:10)
```

Out[148]:
```
55
```

Uma advertência quando usando laços `for` e `while` se você estiver usando *Julia* um *script* fora de uma função. Em um *script*, internamente a um laço `for` o programa não enxerga variáveis externas. Assim, nos exemplos anteriores, a variável *soma* não estaria disponível dentro dos laços. Aqui não houve problema porque este documento está sendo escrito usando um *notebook* do *Jupyter*, e as regras dentro de *notebooks* são um pouco diferentes.

Há duas soluções para isto: Se o laço `for` ou `while` estiver definido dentro de uma função ou diretamente na linha de comando do `REPL`, ou no ambiente do *Jupyter*, variáveis externas estarão disponíveis para o laço.

Se você estiver num *script*, e definir um laço diretamente sem criar uma função, é necessário modificar um pouco o código colocando a diretiva `global soma` dentro do laço:

```
soma=0
for i=1:10
    global soma
    soma+=i
end
```

Isto ocorre porque todas as variáveis criadas fora de funções e laços são do tipo `global`, e laços herdam do ambiente em que são definidos as variáveis locais, mas não as variáveis globais.

Isso é um pouco confuso no início, mas a regra geral é a seguinte: se você criar um laço, e ao tentar usar uma variável definida fora dele, receber uma mensagem de erro do tipo `ERROR: UndefVarError: soma not defined`, basta acrescentar o comando `global soma` no início do laço.

Funções

Você pode criar funções em *Julia* em qualquer lugar - no meio do código da linha de comando, ou num arquivo em separado.

Se você criar a função em um arquivo em separado, deve carregá-la antes de usá-la com a função `include("Nome_do_arquivo.jl")`. Arquivos em *Julia* normalmente levam a extensão `.jl`.

Vamos criar uma função simples:

In [149…
```
function cosexp(x)
    y=cos(5x)*exp(-0.5x)
    return y
end
```

Out[149]:
```
cosexp (generic function with 1 method)
```

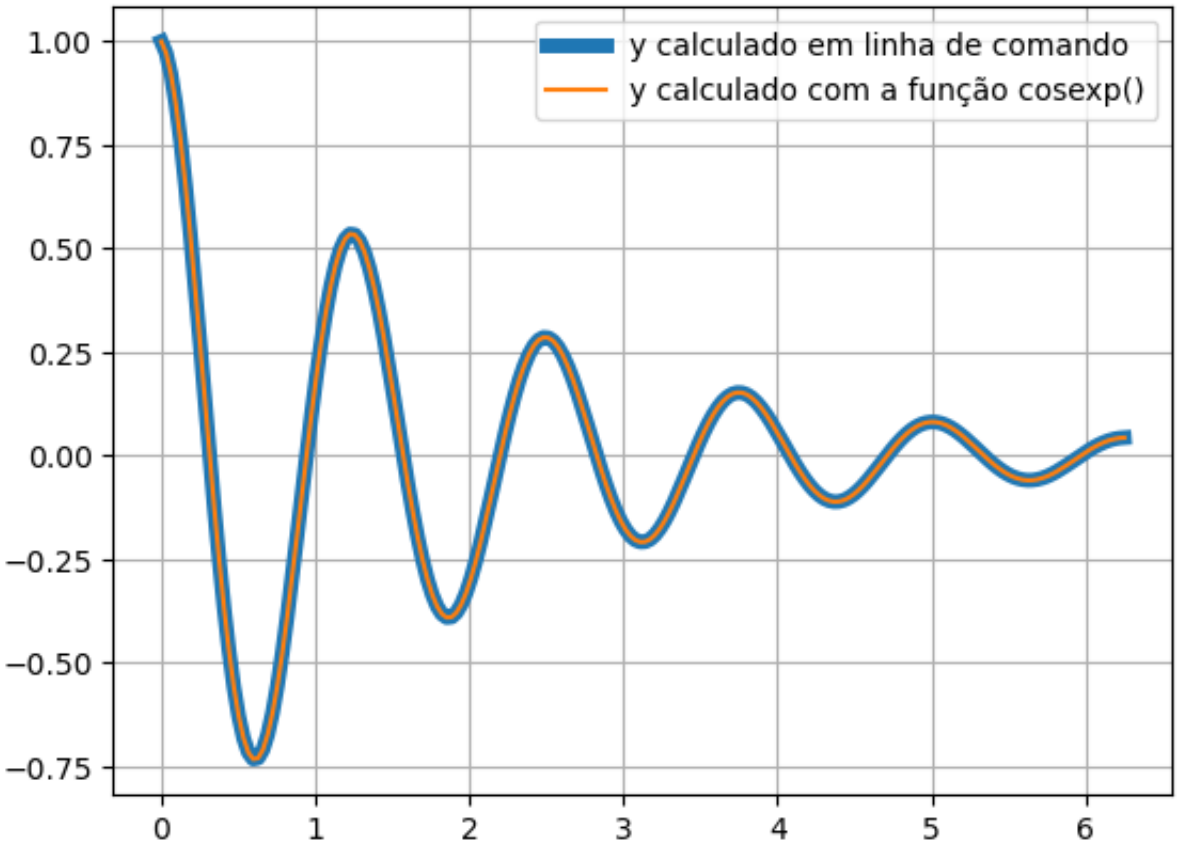
Uma vez definida, você pode usar a função normalmente:

In [150…
```
cosexp(2)
```

Out[150]:
```
-0.30867716521951294
```

Como a função `cosexp()` tem apenas um argumento escalar, você pode aplicá-la a todos os elementos de um vetor usando `cosexp.()`:

In [154…
```
y1=cosexp.(x)
plot(x,y, label="y calculado em linha de comando", lw=5) # Desenha com li
plot(x,y1, label="y calculado com a função cosexp()")
grid()
legend();
```



Repare que a saída da função é informada pelo operador `return`. Uma função pode ter mais de uma saída também, como por exemplo

In [155…
```
function fasor(V)
    r=abs(V)
    θ=angle(V)*180/π
    return r,θ
end
```

Out[155]:
```
fasor (generic function with 1 method)
```

As duas saídas são obtidas chamando-se a função como a seguir.

In [156…
```
p, ϕ = fasor(1+im)
```

Out[156]:
```
(1.4142135623730951, 45.0)
```

In [157…
```
p
```

Out[157]:
```
1.4142135623730951
```

In [158…
```
ϕ
```

Out[158]:
```
45.0
```

Variáveis aleatórias

Você pode gerar números (pseudo-)aleatórios aproximando uma distribuição gaussiana padrão usando o comando `randn()`, e números aproximando uma distribuição uniforme no intervalo $[0, 1]$ usando o comando `rand()`:

In [159…
```
randn()
```
Out[159]:
```
-1.9233738047200364
```

In [160…
```
randn()
```
Out[160]:
```
0.7659180183118443
```

In [161…
```
randn()
```
Out[161]:
```
1.5316759499104946
```

In [162…
```
randn()
```
Out[162]:
```
-0.6524340647417541
```

Também é possível gerar um vetor de números aleatórios:

In [163…
```
x=randn(5)
```
Out[163]:
```
5-element Vector{Float64}:
 -1.1285958851699476
 -0.31480574095019387
 -0.8086496500518047
  0.5281361225107603
  0.5011105460879153
```

In [164…
```
y=rand(5)
```
Out[164]:
```
5-element Vector{Float64}:
 0.7232208232282403
 0.08503849180179446
 0.169286842501708
 0.7418097506560867
 0.4760169193859686
```

Note que no *Matlab*, `rand(5)` gera uma matriz 5x5, e `rand(5,1)` gera um vetor. Em *Julia*, a matriz é criada com `rand(5,5)`.

Finalmente, pode-se criar matrizes aleatórias também:

In [165…
```
R=randn(3,4)
```
Out[165]:
```
3×4 Matrix{Float64}:
  2.50826    0.824542  -2.06794    0.934835
 -0.47709   -3.31282   -0.263293   0.899619
  0.11126    1.7033     0.283249   0.393496
```

O pacote `Distributions` permite que você calcule probabilidades de diversas distribuições.

In [166…
```
using Distributions
```

Agora você pode criar "variáveis aleatórias" com distribuições diversas:

In [167…
```
X=Normal(1, 2)
```

Out[167]:
```
Normal{Float64}(μ=1.0, σ=2.0)
```

A função `Normal(1, 2)` cria uma "variável aleatória" com valor esperado 1 e **desvio-padrão** 2 (cuidado que essa notação é diferente da que usamos habitualmente, em que $N(a, b)$ representa uma variável aleatória com valor esperado a e **variância** b).

Podemos achar a probabilidade de X ser menor do que 0.5 usando

In [168…
```
cdf(X, 0.5)
```

Out[168]:
```
0.4012936743170763
```

Obviamente, a probabilidade de $X \leq 1$ é 0.5:

In [169…
```
cdf(X, 1)
```

Out[169]:
```
0.5
```

A função `ccdf(x)` é `1-cdf(x)`, ou seja 'ccdf(x)`$= \text{Prob}(\{X > x\})$:

In [170…
```
ccdf(X,0.5)
```

Out[170]:
```
0.5987063256829237
```

In [171…
```
ccdf(X,0.5)+cdf(X,0.5)
```

Out[171]:
```
1.0
```

Outra função útil é `quantile(p),` que fornece o valor de x para o qual $\text{Prob}(\{X \leq x\}) = p$, para um valor de p escolhido:

In [172…
```
quantile(X, 0.4)
```

Out[172]:
```
0.49330579372840055
```

In [173…
```
quantile(X,0.5)
```

Out[173]:
```
1.0
```

A função `cquantile(p)` fornece o valor de x para o qual $\text{Prob}(\{X > x\}) = p$:

In [174…
```
cquantile(X, 0.4)
```
Out[174]:
```
1.5066942062715993
```

In [175…
```
cquantile(X, 0.6)
```
Out[175]:
```
0.49330579372840055
```

O pacote `Polynomials` é útil para trabalhar com polinômios. Você precisa carregá-lo com `using` (e, conforme o caso, instalá-lo antes com o comando `add` no modo de pacotes).

In [176…
```
using Polynomials
```

In [178…
```
p=Polynomial([2, 3, 1],:x)
```
Out[178]: $2 + 3 \cdot x + x^2$

A entrada `:x` no final informa qual a variável será usada para descrever o polinômio. Preste atenção que a ordem dos coeficientes é a inversa do Matlab: o primeiro coeficiente corresponde ao termo de menor grau, e o último ao termo de maior grau.

In [179…
```
roots(p)
```
Out[179]:
```
2-element Vector{Float64}:
 -2.0
 -1.0
```

In [180…
```
q=Polynomial([2;1],:x)
```
Out[180]: $2 + x$

In [181…
```
s=p*q
```
Out[181]: $4 + 8 \cdot x + 5 \cdot x^2 + x^3$

In [182…
```
roots(s)
```
Out[182]:
```
3-element Vector{Float64}:
 -2.000000057059736
 -1.9999999429402628
 -1.0000000000000013
```

Você pode criar polinômios com outras variáveis também:

In [183…
```
r=Polynomial([2;1],:z)
```
Out[183]: $2 + z$

mas operações com polinômios só funcionam se eles forem descritos em termos da mesma variável. Por exemplo, o produto `p*r` resultará em um erro.

A função `fromroots()` cria um polinômio a partir das suas raízes (atenção que o nome da variável usa uma sintaxe diferente - se não for definido, a variável usada será `x`):

In [184…
```
p1=fromroots([-2;-1]; var=:z)
```

Out[184]: $2 + 3 \cdot z + z^2$

In [185…
```
p2 = fromroots([-2,1])
```

Out[185]: $-2 + x + x^2$

In [186…
```
p-p2
```

Out[186]: $4 + 2 \cdot x$

Filtros digitais

Várias funções úteis para processamento de sinais estão no pacote `DSP`. Você precisa carregá-lo com o comando `using` (e se necessário antes instalá-lo com o comando `add` no modo de pacotes).

In [187…
```
using DSP
```

Podemos definir um filtro FIR através do vetor de coeficientes usando os comandos a seguir:

In [188…
```
ωc=π/3
```

Out[188]:
```
1.0471975511965976
```

In [189…
```
N=21; L=(N-1)/2
```

Out[189]:
```
10.0
```

Lembre que um filtro passa-baixas ideal com corte na frequência ω_c tem resposta ao impulso

$$h_d[n] = \frac{\omega_c}{\pi}\text{sinc}\left(\frac{\omega_c}{\pi}(n-L)\right),\ 0 \leq n \leq N-1.$$

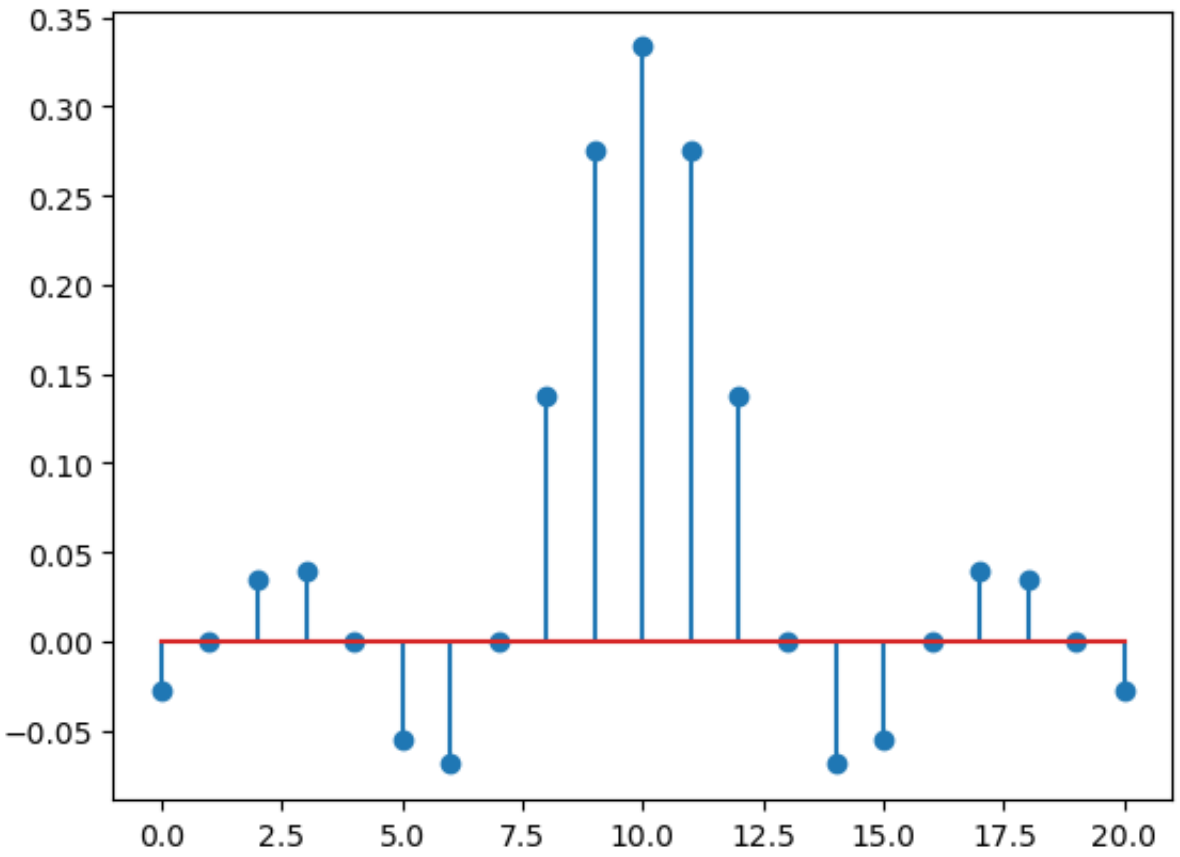
Para gerar essa resposta ao impulso, criamos um vetor n com valores entre 0 e $N-1$:

In [190…
```
n=0:N-1;
```

Para criar a resposta ao impulso, usamos a função sinc. Cuidado apenas com o cálculo de $n - L$: como n é um vetor, a operação $n - L$ irá gerar um erro (pois estamos tentando subtrair um escalar de um vetor). Precisamos então usar " .- ", para indicar que queremos subtrair o escalar de todos os elementos do vetor:

In [191…
```
hd=(ωc/π)*sinc.((ωc/π)*(n .- L));
```

In [192…
```
stem(n,hd);
```



Podemos calcular a resposta em frequência do filtro usando o comando `freqz`. No pacote `DSP`, o comando `freqz` é um pouco diferente do correspondente no *Matlab*: as entradas são uma variável do tipo `filter`, e um vetor com as frequências em que se deseja calcular a saída.

Primeiro definimos uma variável do tipo `filter` através da função `PolynomialRatio`, que pede como entrada dois vetores, com os coeficientes do numerador e do denominador do filtro desejado. Como o nosso filtro é FIR, o denominador é apenas 1. Cuidado porque aqui há uma pegadinha: a função `PolynomialRatio(b,a)` precisa que tanto a quanto b sejam vetores. Se você escrever

```
hdf=PolynomialRatio(hd,1)
```

vai receber uma mensagem de erro, porque 1 é um escalar, não um vetor. Para não ter problemas, você deve escrever

```
hdf=PolynomialRatio(hd, [1])
```

como abaixo:

In [193…
```
hdf=PolynomialRatio(hd,[1.0]);
```

Agora podemos definir o vetor de pontos. Para simplificar, vamos usar a função `range`:

In [194…
```
ω=range(0,π, length=500);
```

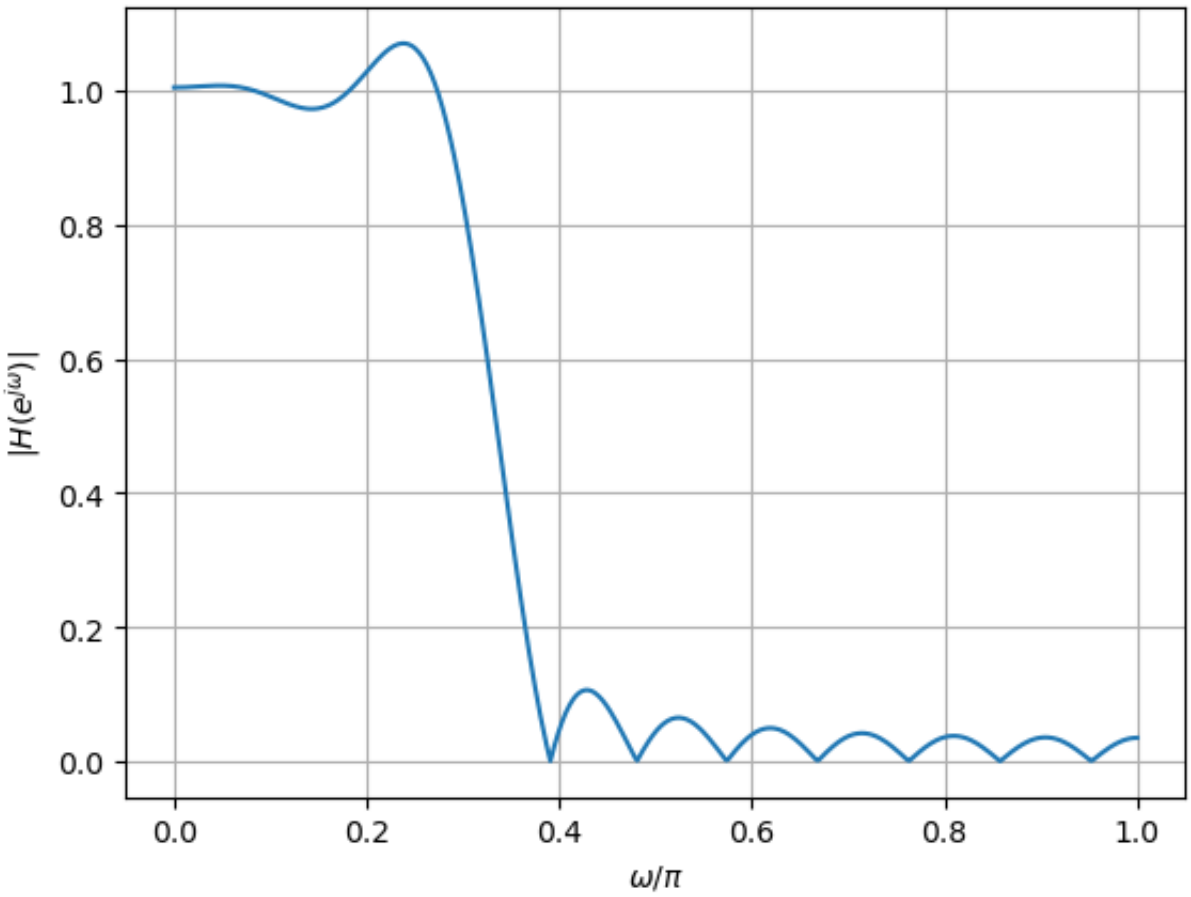
A função `range(a, b, length=N)` retorna um vetor com N elementos. O primeiro elemento é a, e o último é b.

Agora podemos calcular a resposta em frequência do filtro:

In [197…
```
Hd=freqresp(hdf, ω);
```

Você também pode usar o comando `Hd,ω = freqresp(hdf)` - neste caso o vetor ω não precisa ser criado antes.

In [205…
```
plot(ω/π, abs.(Hd))
xlabel(L"\omega/\pi")
ylabel(L"|H(e^{j\omega})|")
grid();
```



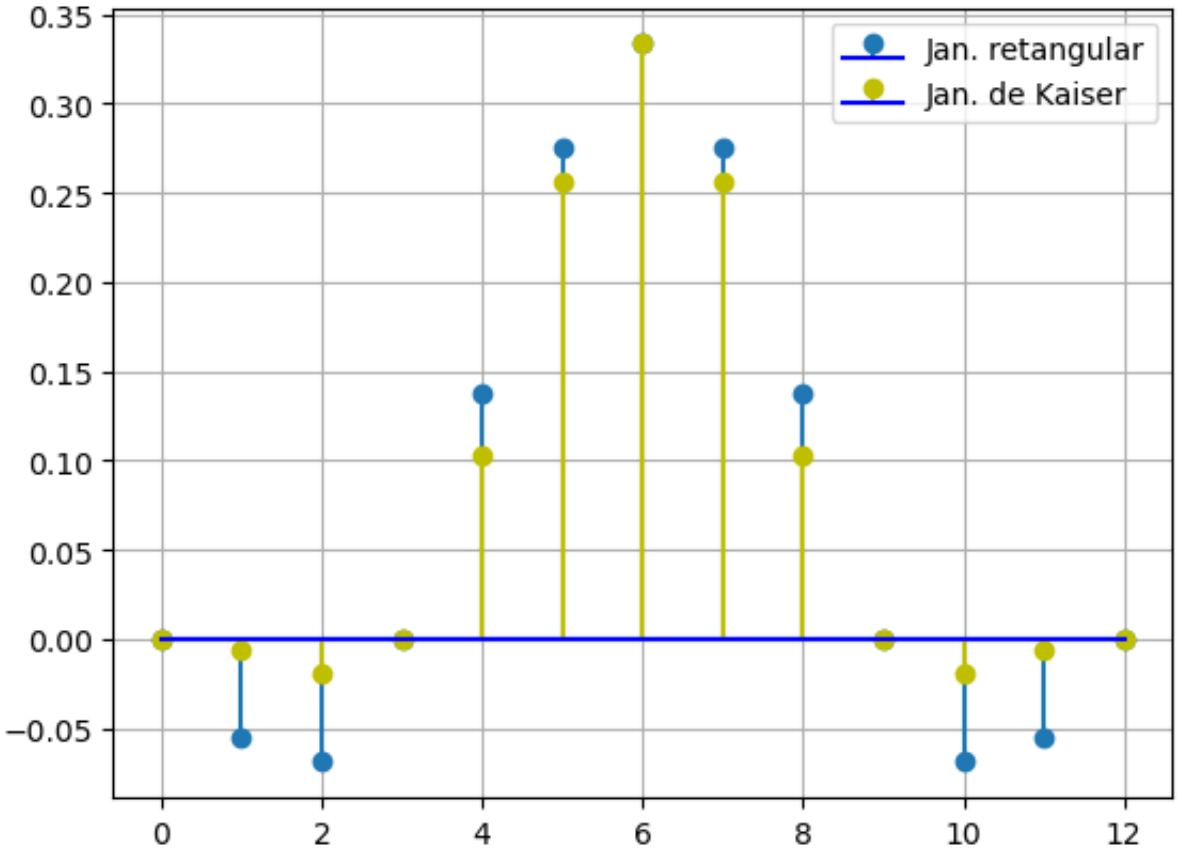
Para evitar as oscilações, podemos usar uma janela de Kaiser. Suponha que desejemos uma atenuação de 60dB e uma faixa de transição de largura $\Delta\omega = \pi/5$. Então podemos calcular os parâmetros da janela de Kaiser usando a função `kaiserord()` (cuidado que a definição do parâmetro β em *Julia* é diferente da do *Matlab* por um fator de π)

In [209…
```
N,α=kaiserord(π/5, 60)
```

Out[209]:
```
(13, 1.7994885471673767)
```

Agora recalculamos a resposta ao impulso ideal e a multiplicamos pela janela de Kaiser adequada:

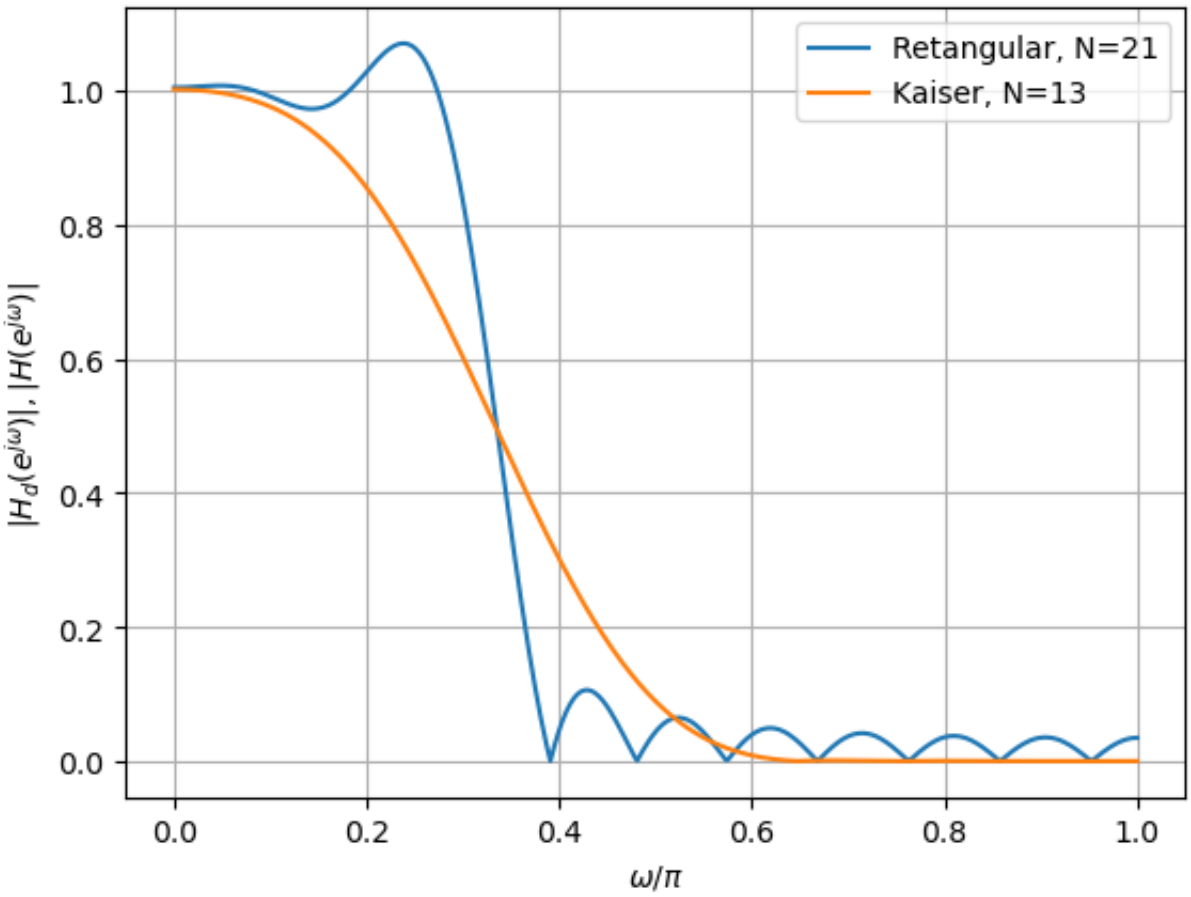
In [210…
```
L=(N-1)/2
n=0:N-1
hd=(ωc/π)*sinc.((ωc/π)*(n .- L))
h=hd.*kaiser(N,α)
stem(n,hd,label="Jan. retangular",basefmt="b")
stem(n,h,label="Jan. de Kaiser",linefmt="y",markerfmt="oy",basefmt="b")
grid();legend();
```



A nova resposta ao impulso fica

In [208…

```
hf=PolynomialRatio(h,[1])
H=freqz(hf,ω)
plot(ω/π,abs.(Hd),label="Retangular, N=21")
plot(ω/π,abs.(H),label="Kaiser, N=13");grid()
xlabel(L"\omega/\pi")
ylabel(L"|H_d(e^{j\omega})|, |H(e^{j\omega})|")
legend();
```



Atenção: o valor do parâmetro β da janela de Kaiser visto no curso deve ser dividido por π para usar no comando `kai` ser deve ser dividido por π: `kaiser(N,β/π)`

Também é possível criar filtros IIR. Por exemplo, o filtro

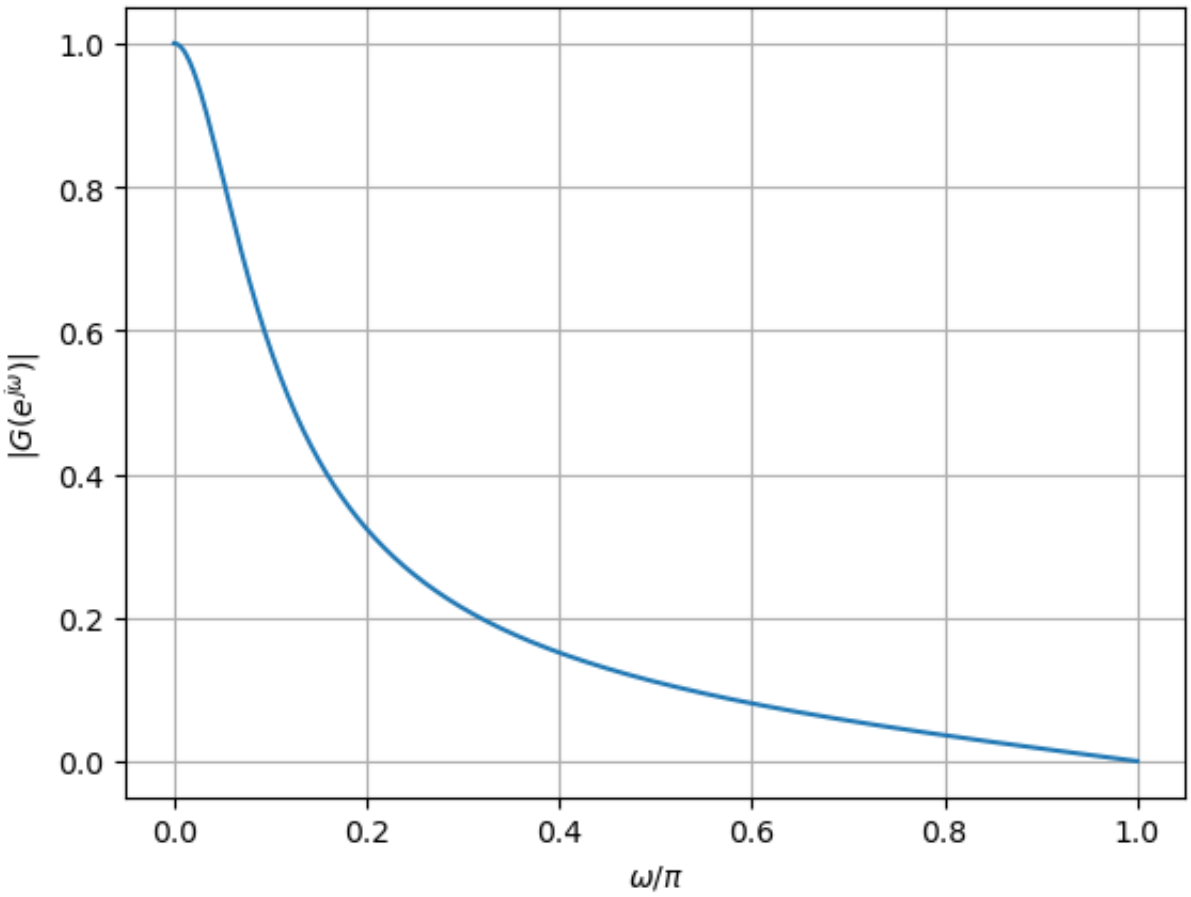
$$G(z) = 0,01\frac{1+z^{-1}}{1-0.99z^{-1}}$$

pode ser criado usando tanto a função `PolynomialRatio()` (cujas entradas são os coeficientes do numerador e do denominador) quanto a função `ZeroPoleGain()`, cujas entradas são os zeros, pólos e o ganho. Veja os exemplos:

In [211…
```
g=PolynomialRatio(0.1*[1;1],[1;-0.8])
```

Out[211]:
```
PolynomialRatio{:z, Float64}(LaurentPolynomial(0.1*z⁻¹ + 0.1), LaurentPo
lynomial(-0.8*z⁻¹ + 1.0))
```

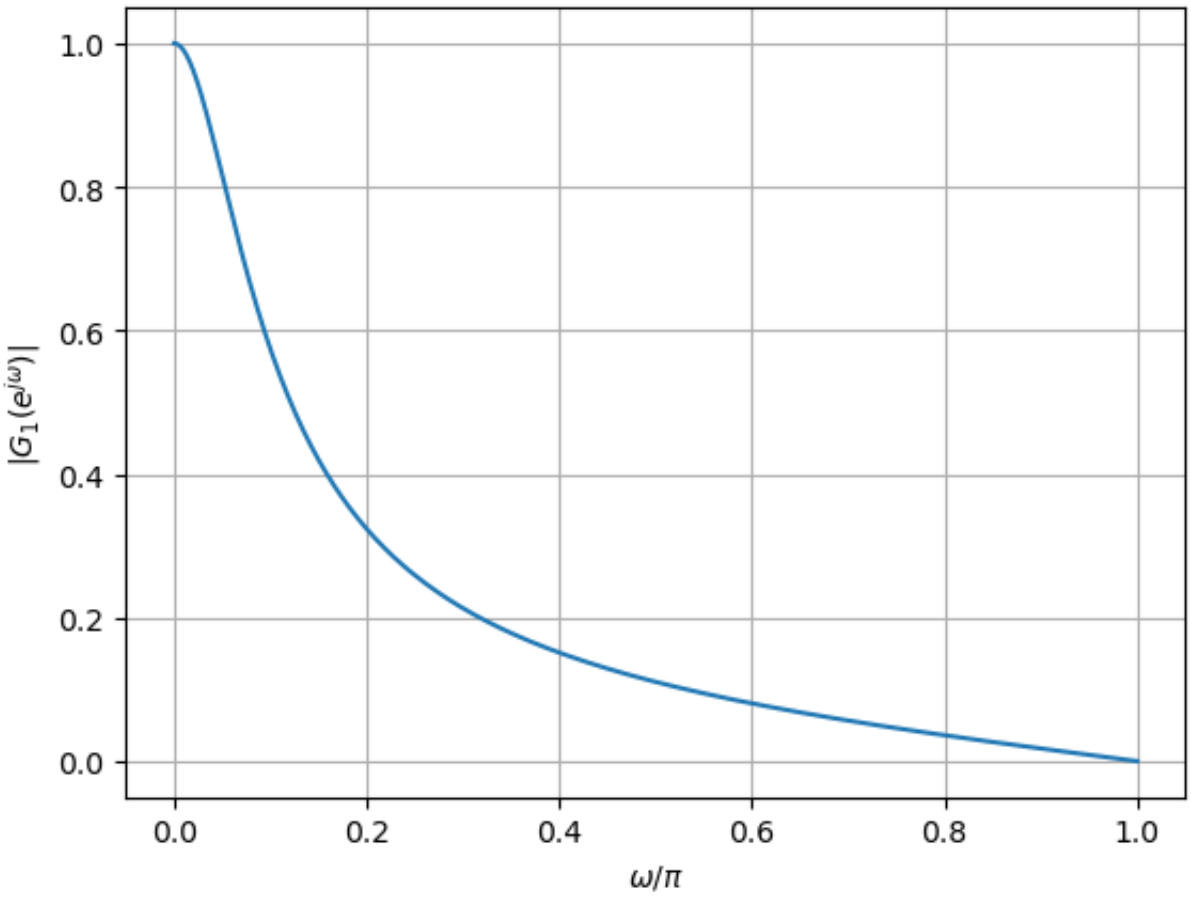
In [212…
```
G=freqz(g,ω);
plot(ω/π, abs.(G))
xlabel(L"\omega/\pi")
ylabel(L"|G(e^{j\omega})|")
grid();
```



In [213…
```
g1=ZeroPoleGain([-1],[0.8],0.1)
```

Out[213]:
```
ZeroPoleGain{:z, Int64, Float64, Float64}([-1], [0.8], 0.1)
```

In [214…
```
G1=freqz(g1,ω);
plot(ω/π, abs.(G1))
xlabel(L"\omega/\pi")
ylabel(L"|G_1(e^{j\omega})|")
grid();
```



Repare que se você quiser definir um filtro sem nenhum zero finito, por exemplo

$$G_2(z) = 0.2\frac{1}{1-0.99z^{-1}}$$

usando a função ZeroPoleGain, você precisa usar um vetor "vazio" (com zero elementos), como

In [215…
```
zvazio = Array{Float64,1}()
```

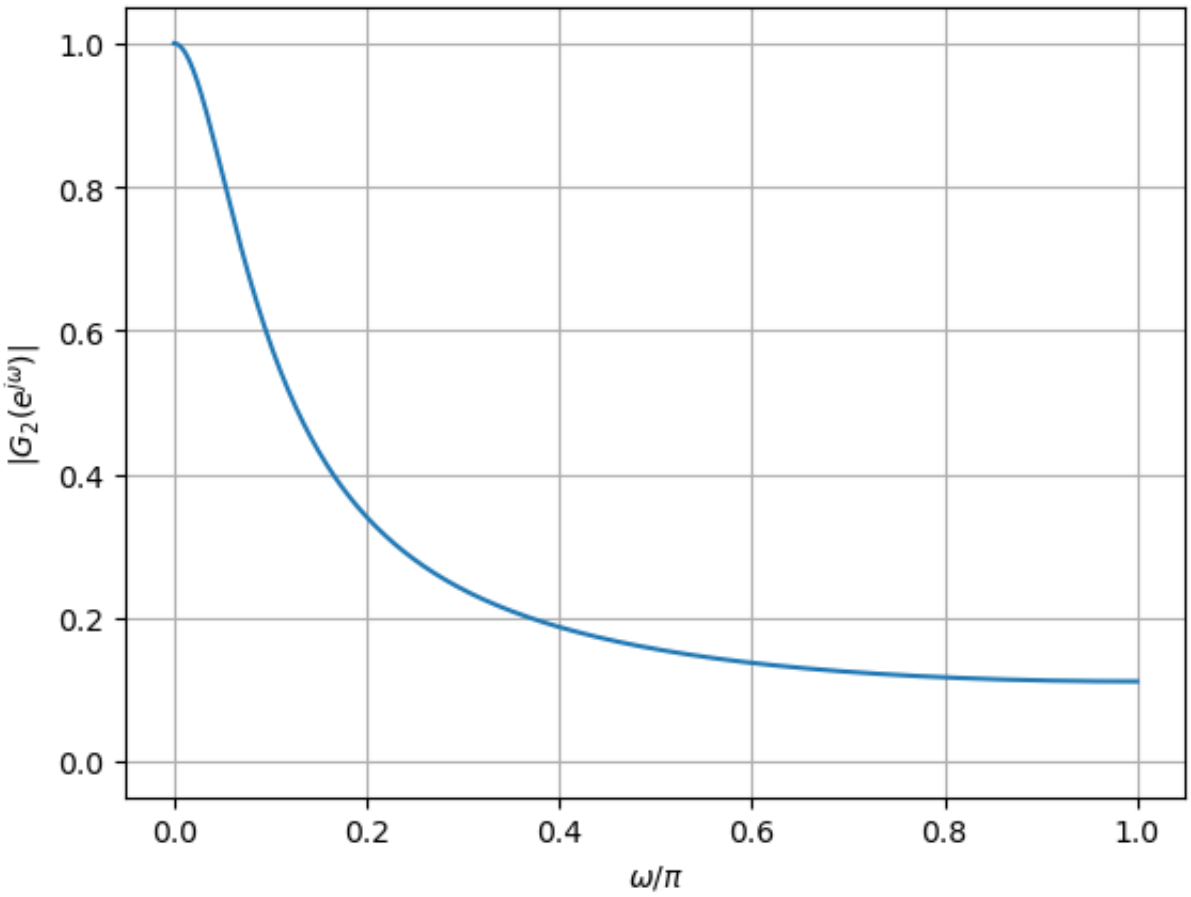
Out[215]:
```
Float64[]
```

Veja mais detalhes sobre tipos de variáveis na seção Tipos de variáveis.

In [216…
```
g2 = ZeroPoleGain(zvazio, [0.8], 0.2)
```

Out[216]:
```
ZeroPoleGain{:z, Float64, Float64, Float64}(Float64[], [0.8], 0.2)
```

In [217…
```
G2=freqz(g2,ω);
plot(ω/π, abs.(G2))
xlabel(L"\omega/\pi")
ylabel(L"|G_2(e^{j\omega})|")
grid();
axis([-0.05, 1.05, -0.05, 1.05])
```

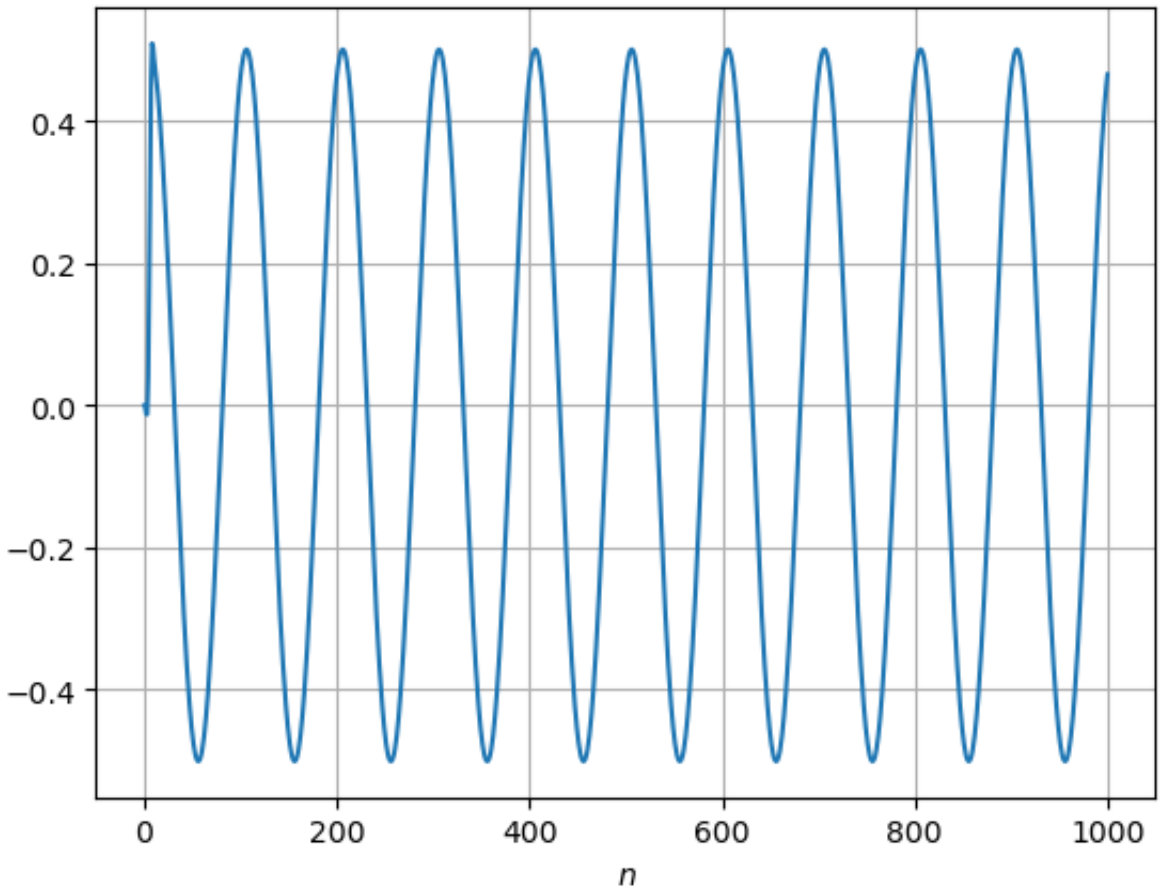


Out[217]: (-0.05, 1.05, -0.05, 1.05)

A saída de um filtro para uma entrada qualquer pode ser calculada usando-se a função `filt()`

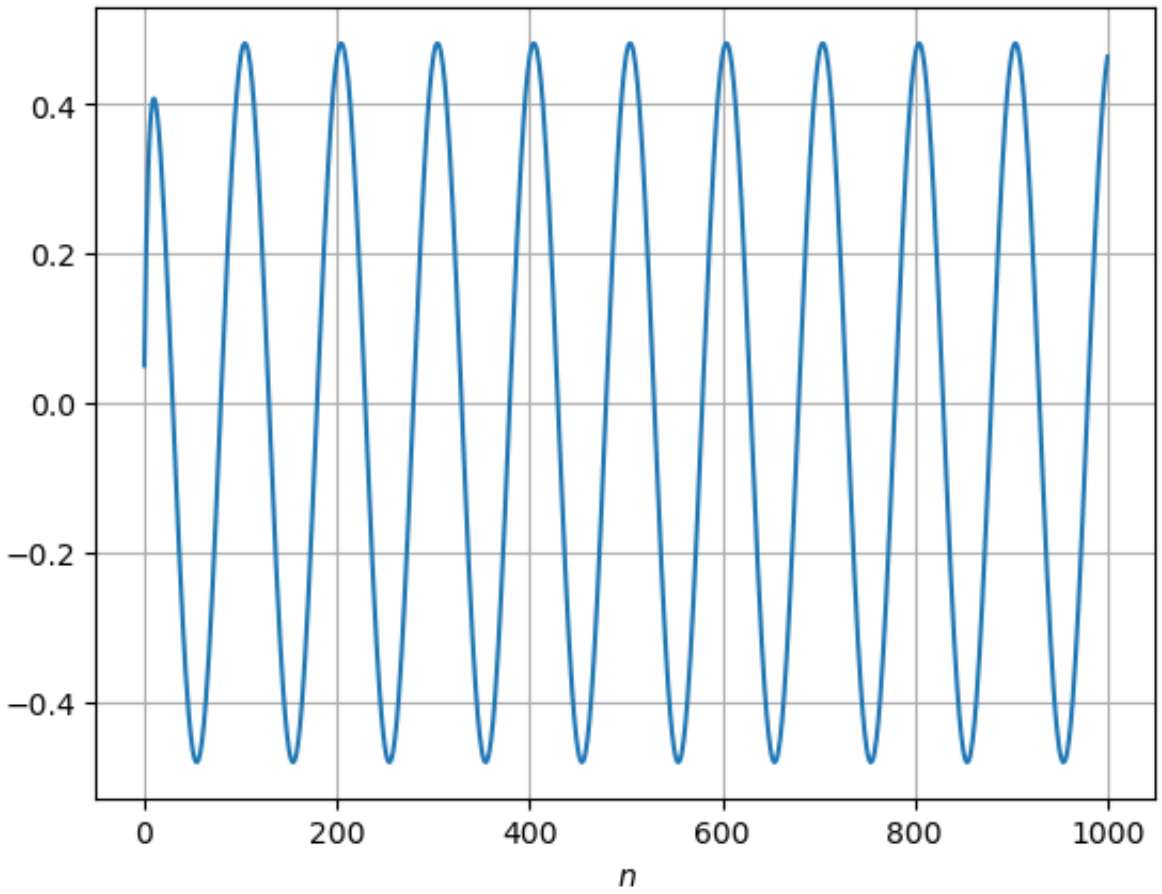
In [218…
```
n1=0:1000
x=0.5*cos.((π/50)n1)
y=filt(hf,x);
```

In [219…
```
plot(n1,y);grid();xlabel(L"n");
```



In [220…

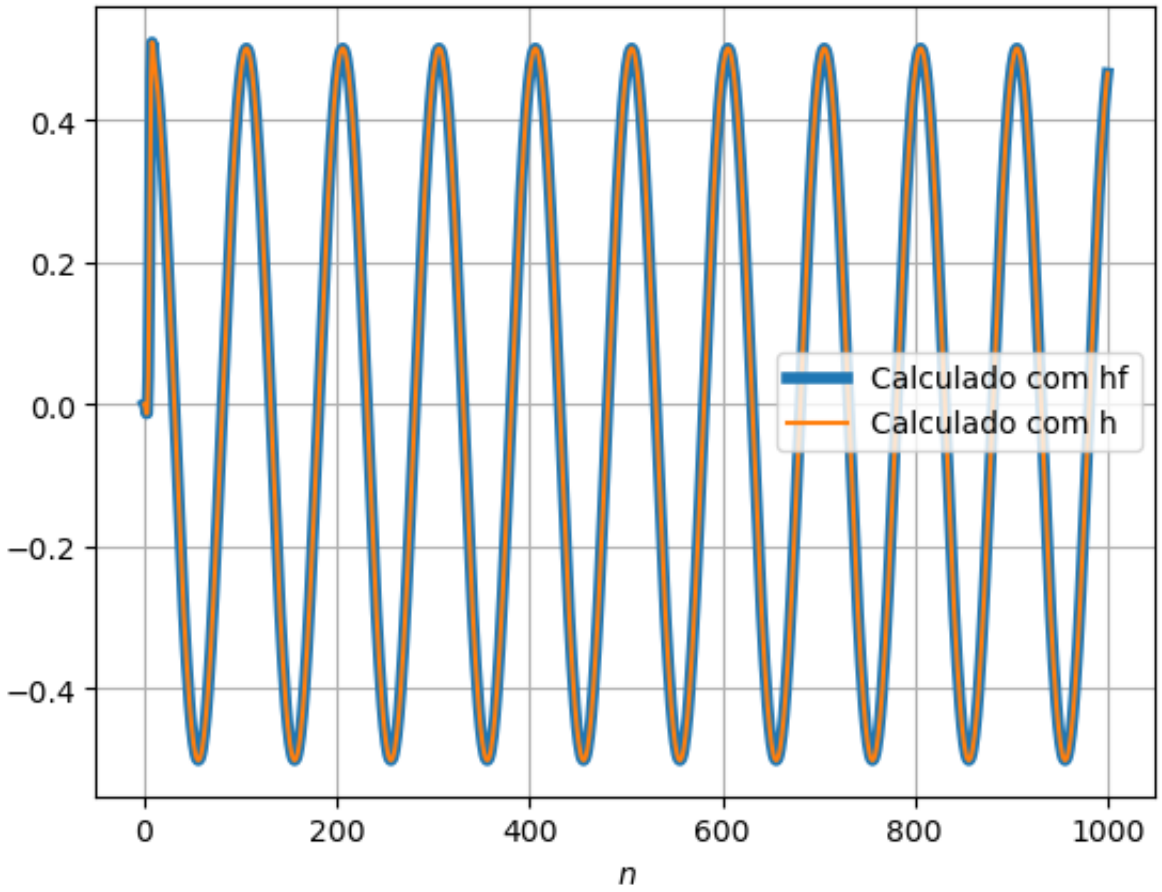
```
y1=filt(g,x);
plot(n1,y1);grid();xlabel(L"n");
```



A função `filt()` também pode ser usada definindo o filtro diretamente pelos coeficientes do numerador e do denominador (repare que a função `filt()` aceita que a segunda entrada seja um escalar):

In [222…

```
yalt=filt(h,1,x);
plot(n1,y,label="Calculado com hf",lw=4);grid();xlabel(L"n");
plot(n1,yalt,label="Calculado com h")
legend();
```



Tipos de variáveis em *Julia*

Se você estiver escrevendo um programa simples só para resolver um exercício, em geral não é importante prestar atenção se uma variável é um inteiro, um número em ponto flutuante, etc. No entanto, isso é importante quando você quer gerar código eficiente, para resolver um problema real. Também é importante para entender as mensagens de erro de *Julia*. Como a linguagem foi projetada para ser eficiente, para aplicações em cálculo numérico, em algumas situações a conversão de variáveis de um tipo para outro não é automática.

Os dois tipos de variáveis mais comuns são inteiros e "reais" (ponto flutuante). Você pode escolher entre um tipo e outro ao entrar uma variável escrevendo ou não o ponto decimal. Por exemplo,

In [223…
```
n=2
```

Out[223]:
```
2
```

é um inteiro, enquanto

In [224…
```
x=2.
```

Out[224]:
```
2.0
```

Você pode verificar o tipo de uma variável usando a função `typeof()`

In [225…
```
typeof(n)
```

Out[225]:
```
Int64
```

In [226…
```
typeof(x)
```

Out[226]:
```
Float64
```

In [227…
```
q=2//3
typeof(q)
```

Out[227]:
```
Rational{Int64}
```

Ao fazer uma conta contendo números de diferentes tipos, quando possível *Julia* converte todos os números para o tipo mais geral, e faz a conta considerando esse tipo.

In [228…
```
n+x
```

Out[228]:
```
4.0
```

In [229…
```
n+q
```

Out[229]:
```
8//3
```

In [230…
```
x+q
```

Out[230]:
```
2.6666666666666665
```

In [231…
```
z=1+0im
```

Out[231]:
```
1 + 0im
```

In [232…
```
z+n
```

Out[232]:
```
3 + 0im
```

In [233…
```
z+q
```

Out[233]:
```
5//3 + 0//1*im
```

In [234…
```
z+x
```

Out[234]:
```
3.0 + 0.0im
```

Apenas inteiros podem ser usados para indexar vetores e matrizes:

In [235…
```
v=[1,2,3].^2
```

Out[235]:
```
3-element Vector{Int64}:
 1
 4
 9
```

In [236…
```
v[n]
```

Out[236]:
```
4
```

In [237…
```
v[x]
```

```
ArgumentError: invalid index: 2.0 of type Float64

Stacktrace:
 [1] to_index(i::Float64)
   @ Base ./indices.jl:300
 [2] to_index(A::Vector{Int64}, i::Float64)
   @ Base ./indices.jl:277
 [3] to_indices
   @ ./indices.jl:333 [inlined]
 [4] to_indices
   @ ./indices.jl:325 [inlined]
 [5] getindex(A::Vector{Int64}, I::Float64)
   @ Base ./abstractarray.jl:1218
 [6] top-level scope
   @ In[237]:1
 [7] eval
   @ ./boot.jl:373 [inlined]
 [8] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::S
tring, filename::String)
   @ Base ./loading.jl:1196
```

Isto pode gerar alguns erros estranhos quando você calcula o valor de um índice. Por exemplo, digamos que você queira pegar o elemento do meio de um vetor:

In [238…
```
N=length(v)
```

Out[238]:
```
3
```

In [239…
```
i=N/2
```

Out[239]:
```
1.5
```

Como o resultado não é um número inteiro, podemos arredondar o resultado para cima:

In [240…
```
i=ceil(i)
```

Out[240]:
```
2.0
```

No entanto, o resultado não é inteiro, e você vai receber um erro se tentar acessar `v[i]`:

In [241…
```
v[i]
```

```
ArgumentError: invalid index: 2.0 of type Float64

Stacktrace:
 [1] to_index(i::Float64)
   @ Base ./indices.jl:300
 [2] to_index(A::Vector{Int64}, i::Float64)
   @ Base ./indices.jl:277
 [3] to_indices
   @ ./indices.jl:333 [inlined]
 [4] to_indices
   @ ./indices.jl:325 [inlined]
 [5] getindex(A::Vector{Int64}, I::Float64)
   @ Base ./abstractarray.jl:1218
 [6] top-level scope
   @ In[241]:1
 [7] eval
   @ ./boot.jl:373 [inlined]
 [8] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::S
tring, filename::String)
   @ Base ./loading.jl:1196
```

Para resolver o problema, você pode converter o resultado para um inteiro: isso é fácil, usando as funções `Int()` ou `Int64()` - convertem o argumento para um inteiro de 64 bits.

Também há inteiros de 128 (`Int128`), 32 (`Int32`), 16 (`Int16`) e 8 (`Int8`) bits, e inteiros sem sinal (`UInt`).

In [242…
```
v[Int(i)]
```

Out[242]:
```
4
```

In [243…
```
typeof(Int(i))
```

Out[243]:
```
Int64
```

Você também poderia usar o operador ÷, a divisão inteira, ou a versão completa do comando `ceil`:

In [246…
```
i = N÷2 + 1
v[i]
```

Out[246]:
```
4
```

In [247…
```
i=ceil(Int,N/2)
v[i]
```

Out[247]:
```
4
```

Uma outra situação em que você pode ter um resultado inesperado é se tentar usar a função `sqrt()` com um número negativo:

In [248…
```
sqrt(2)
```

Out[248]:
```
1.4142135623730951
```

In [249…
```
sqrt(-2)
```

```
DomainError with -2.0:
sqrt will only return a complex result if called with a complex argument.
Try sqrt(Complex(x)).

Stacktrace:
 [1] throw_complex_domainerror(f::Symbol, x::Float64)
   @ Base.Math ./math.jl:33
 [2] sqrt
   @ ./math.jl:567 [inlined]
 [3] sqrt(x::Int64)
   @ Base.Math ./math.jl:1221
 [4] top-level scope
   @ In[249]:1
 [5] eval
   @ ./boot.jl:373 [inlined]
 [6] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::S
tring, filename::String)
   @ Base ./loading.jl:1196
```

Para aumentar a eficiência da linguagem, a função `sqrt()` não testa se a entrada é positiva ou não, e escolhe se a saída é real ou complexa de acordo com o valor da entrada - na verdade existe uma versão da função `sqrt()` para entradas reais e outra para entradas complexas:

In [250…
```
sqrt(-2+0im)
```

Out[250]:
```
0.0 + 1.4142135623730951im
```

O mesmo efeito pode ser obtido convertendo o argumento de `sqrt()` para complexo antes de chamar a função:

In [251…
```
sqrt(Complex(-2))
```

Out[251]:
```
0.0 + 1.4142135623730951im
```

Saber um pouco sobre tipos de variáveis ajuda a entender as mensagens de erro da linguagem. Por exemplo, se você definir

In [252…
```
v=[1,2]
```

Out[252]:
```
2-element Vector{Int64}:
 1
 2
```

e tentar calcular `cos(v)` (sem usar o `.` antes dos parênteses), vai receber uma mensagem de erro

In [253…
```
cos(v)
```

```
MethodError: no method matching cos(::Vector{Int64})
Closest candidates are:
  cos(::T) where T<:Union{Float32, Float64} at ~/julia/usr/share/julia/base/special/trig.jl:98
  cos(::DualNumbers.Dual) at ~/.julia/packages/DualNumbers/5knFX/src/dual.jl:311
  cos(::Hermitian{var"#s858", S} where {var"#s858"<:Complex, S<:(AbstractMatrix{<:var"#s858"})}) at ~/julia/usr/share/julia/stdlib/v1.7/LinearAlgebra/src/symmetric.jl:761
  ...

Stacktrace:
 [1] top-level scope
   @ In[253]:1
 [2] eval
   @ ./boot.jl:373 [inlined]
 [3] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String, filename::String)
   @ Base ./loading.jl:1196
```

O que essa mensagem está dizendo é que a função `cos()` recebeu como entrada um vetor, mas não está definida para entradas do tipo vetor. Para resolver o problema, use `cos.(v)`:

In [254…
```
cos.(v)
```

Out[254]:
```
2-element Vector{Float64}:
  0.5403023058681398
 -0.4161468365471424
```

Outra situação sutil é a diferença entre uma matriz com uma coluna e um vetor. Veja, por exemplo, as definições a seguir. Veja a descrição do tipo de um vetor ou de uma matriz:

In [255…
```
a=[1,2]
```

Out[255]:
```
2-element Vector{Int64}:
 1
 2
```

In [256…
```
a=[1 2]
```

Out[256]:
```
1×2 Matrix{Int64}:
 1  2
```

Para definir um vetor coluna, você pode separar os elementos com `,` ou `;`. No entanto, no caso de matrizes é necessário usar `;`.

In [257…
```
B=[1 2;3 4]
```

Out[257]:
```
2×2 Matrix{Int64}:
 1  2
 3  4
```

Repare que `a` é `Array{Int64,1}`, ou seja: uma lista unidimensional de números inteiros, enquanto que `B` é `Array{Int64,2}`, ou seja: um arranjo bidimensional de números inteiros. Em outras palavras, `a` é um vetor e `b` é uma matriz.

A transposta de `a` é um vetor-linha:

In [258…
```
a'
```

Out[258]:
```
2×1 adjoint(::Matrix{Int64}) with eltype Int64:
 1
 2
```

Você pode criar um vetor com *randn* usando um único argumento:

In [259…
```
x=randn(3)
```

Out[259]:
```
3-element Vector{Float64}:
  0.5096644749530393
 -0.42948902400942096
 -0.9356756216140668
```

No entanto, se fizer como no *Matlab*, e usar dois argumentos, o que você obtém sempre é considerado uma matriz:

In [260…
```
y=randn(3,1)
```

Out[260]:
```
3×1 Matrix{Float64}:
 -0.07500411617949003
  1.2598234698456476
  0.05876428302825961
```

Veja que a descrição mudou: no primeiro caso, `x` é um `3-element Vector{Float64}`, que é equivalente a um `Array{Float64,1}`, enquanto que no segundo caso, `y` é um `3x1 Matrix{Float64}`, que é equivalente a um `Array{Float64,2}`. `x` é vetor, enquanto que `y` é uma matriz que acontece de ter uma única coluna.

Na maior parte das vezes você não vai ter problema nenhum com isso, nem vai reparar na diferença. Você pode ter problemas no entanto com algumas operações, vamos ver uns exemplos a seguir e como consertar. Imagine que você queira calcular os produtos escalares de `x` por `x`, `x` por `y`, `y` por `y`. Como vimos antes, você pode usar tanto a expressão `x'*y` quanto `x⋅y` (o operador `⋅` é obtido digitando-se `\cdot[TAB]`:

In [261…
```
x'*x
```

Out[261]:
```
1.3197075676565926
```

In [262…
```
x'*y
```

Out[262]:
```
1×1 adjoint(::Vector{Float64}) with eltype Float64:
 -0.6342915930312778
```

In [263…
```
y'*y
```

Out[263]:
```
1×1 Matrix{Float64}:
 1.5962340335776193
```

Repare que cada caso resultou numa variável de tipo diferente: `x' * x` resultou em um escalar, `x' * y` resultou em um "vetor"-linha com um único elemento, e `y' * y` resultou em uma "matriz" 1x1.

Novamente, em geral isso não vai causar muito problema, a não ser que você queira usar o resultado da operação como entrada de uma função que só aceita escalares:

In [264…
```
exp(x'*x)
```

Out[264]:
```
3.742326839822023
```

In [265…
```
exp(x'*y)
```

```
MethodError: no method matching exp(::Adjoint{Float64, Vector{Float64}})
Closest candidates are:
  exp(::Union{Float16, Float32, Float64}) at ~/julia/usr/share/julia/base
/special/exp.jl:296
  exp(::StridedMatrix{var"#s859"} where var"#s859"<:Union{Float32, Float6
4, ComplexF32, ComplexF64}) at ~/julia/usr/share/julia/stdlib/v1.7/Linear
Algebra/src/dense.jl:560
  exp(::StridedMatrix{var"#s859"} where var"#s859"<:Union{Integer, Comple
x{<:Integer}}) at ~/julia/usr/share/julia/stdlib/v1.7/LinearAlgebra/src/d
ense.jl:561
  ...

Stacktrace:
 [1] top-level scope
   @ In[265]:1
 [2] eval
   @ ./boot.jl:373 [inlined]
 [3] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::S
tring, filename::String)
   @ Base ./loading.jl:1196
```

Há duas maneiras de resolver o problema: uma é usar a função com ".()":

In [266…
```
exp.(x'*y)
```

Out[266]:
```
1×1 Matrix{Float64}:
 0.5303110313133983
```

A outra, mais elegante (porque resulta em variáveis mais simples e com melhor uso da memória) é usar o operador `·` (`\cdot[TAB]`):

In [267…
```
x·x
```

Out[267]:
```
1.3197075676565926
```

In [268…
```
x⋅y
```

Out[268]:
```
-0.6342915930312777
```

In [269…
```
exp(x⋅y)
```

Out[269]:
```
0.5303110313133983
```

In [270…
```
y⋅y
```

Out[270]:
```
1.5962340335776193
```

Desta vez o resultado sempre foi um escalar, como desejado!

Velocidade

Julia pode criar código eficiente, mas para isso é necessário tomar alguns cuidados. Se você tiver curiosidade de entender melhor como usar a linguagem bem, veja esta página e esta outra.

In [ ]:
```

```