12 Dynamic Systems

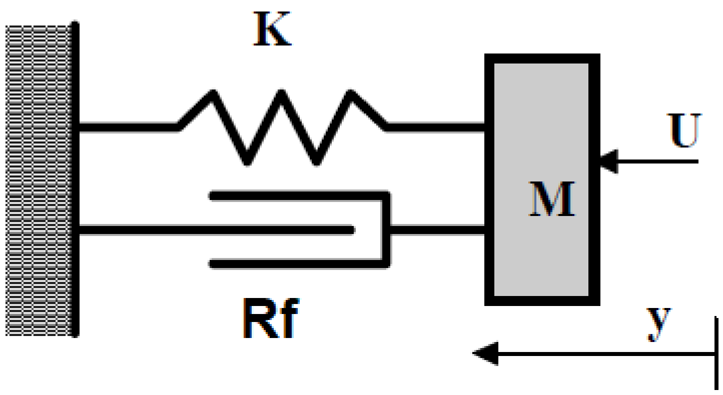
All physical systems that store and transform energy are all dynamic systems that cannot react instantaneously to an external solicitation. This because if an energy transformation were to be instantaneous, it would require an infinite power (P=dW/dt), which is impossible to deliver.

These systems are characterized by their capacity to act or not in a stable way, in a fast way, in an oscillating way, ...

When the underlying physics of dynamic systems is sufficiently known, we have mathematical models capable of describing their temporal behavior: differential equations. Matlab allows the integration of linear differential equations with constant coefficients (LTI: Linear Time Invariant), expressed in different forms (and thus their numerical solutions), but also of non-linear differential equations (for which the most appropriate simulation tool is Simulink but it will not be discussed here).

As an illustration, let's take the following example: if a force applied to a mass is at the origin of an acceleration, it is by integrating this force (divided by the mass) that we will obtain the velocity as a function of time and, by integrating the velocity, we will obtain the distance traveled. So we express the derivatives, Matlab integrates them. That's easy!

We will continue this tutorial with this well known example: the system mass (M), spring (K), damper (Rf), of position y, subjected to an external force u.



with M = 0.5 kg; Rf = 0.2 kg/s and K = 1 kg/s²

The first step in the analysis of a physical system is to write down the laws of physics that govern its operation.

Here, the dynamic equilibrium equation of the forces $\left(\sum_i F_i = M.a\right)$ on the mass M is written :

$M\frac{d^2y}{dt^2} + R_f\frac{dy}{dt} + Ky = u$ or, in a normalized way : $\frac{d^2y}{dt^2} + \frac{R_f}{M}\frac{dy}{dt} + \frac{K}{M}y = \frac{1}{M}u$

Recall that this differential equation can, **<u>only because it is an LTI equation</u>**, be transformed into a polynomial expression as a function of the variable "s" by passing in the Laplace domain.

This passage in the Laplace domain is done with the Laplace transform, and allows us :

- to make our life much easier : we will manipulate algebraic equations rather than differential equations
- to quickly judge the dynamics and stability of the system by identifying its poles (the roots of the denominator)
- to obtain directly the harmonic response of the system, i.e. in sinusoidal regime after the transient phase, by replacing "s" by " $j\omega$".

Our differential equation becomes, by the Laplace transform : $s^2Y(s)+\frac{R_f}{M}sY(s)+\frac{K}{M}Y(s)=\frac{1}{M}U(s)$

The transfer function is the ratio of the output y to the input u : $H(s)=\frac{Y(s)}{U(s)}=\frac{\frac{1}{M}}{s^2+\frac{R_f}{M}s+\frac{K}{M}}$.

This transfer function H(s) will serve as an illustrative example throughout this tutorial. As soon as we talk about the H(s) system, it will be good to remember that the physics behind this transfer function is this mass-spring-damper system.

NB: If the equation relating to this system were a non-linear differential equation, it would be necessary to work with the tools that are briefly presented at the end of this tutorial, or else it would be necessary to accept to work with small variations of the quantities around an operating point of this system to be able to apply the Laplace transform. This amounts to linearizing this system and allows to use all the theory related to transfer functions. This linearization approach is the one most frequently encountered in industry. Let's remember that the most frequent objective in industry is to control a system with a controller. And it turns out that the LTI tools allowing the implementation of this control are much more understandable and simple to implement than the non linear techniques, which explains why these solutions are preferred.

Transfer functions creation (continuous or discrete) in Matlab

Matlab knows the "s" (for the continuous systems) and "z" (for the discrete ones) variables. We can therefore create the transfer function as a fraction of polynomials of "s".

But first, the "s" variable should be created, which is done with the instruction **tf** :

```
clear
close all
s=tf('s')        % Création de la variable s comme étant une fonction de transfert
```

```
s =

  s

Continuous-time transfer function.
```

This variable (Laplace variable) can then be manipulated to write the polynomials of the transfer functions. In the case of the above example, by replacing the parameters by their values, this gives :

```
H = 2/(s^2+0.4*s+2)
```

```
H =

        2
  ---------------
  s^2 + 0.4 s + 2

Continuous-time transfer function.
```

Note that the instruction **tf** can be used in another way, without creating the variable s, by giving it the coefficients of the polynomials in s: tf(num,den) where **num** is the polynomial of the coefficients in s of the numerator and **den** that of the denominator.

It is also possible to create a transfer function in Matlab by providing the zeros, the poles and the gain, via the instruction **zpk**.

These ways of doing things being less intuitive, we will prefer the method based on the creation of the variable s, and refer you to the Matlab help for the rest.

In the same way, we can create a transfer function not in continuous but in discrete, by creating beforehand the variable z, as well as the sampling time of application :

```
z=tf('z',0.1)            % 0.1 is the sampling time, expressed in seconds
```

```
z =

  z

Sample time: 0.1 seconds
Discrete-time transfer function.
```

```
Hz=2/(z^2+0.4*z+2)     % Discrete transfer function, with the sample time
(Ts) = 0.1s, as the "z" variable was created specifiying this value
```

```
Hz =

        2
  ---------------
  z^2 + 0.4 z + 2

Sample time: 0.1 seconds
Discrete-time transfer function.
```

Pay attention, it is important to note that the transfer function Hz created above is not the equivalent of the transfer function H in discrete, even if the numerator and denominator are identical!

To discretize the system H and obtain the discrete version, we can use the "continue to discrete" command, specifying the continuous system to be discretized and the sampling period Ts : **Sysd = c2d(Sysc,Ts)**.

By default, the discretization method used is that of the zero-order blocker:

```
Hz=c2d(H,0.1)                % We get a discete transfert function (using
the z variable). The polynoms (numerator and denominator) are differents
from the ones we had in continuous H.
```

```
Hz =
 
  0.009852 z + 0.009721
  ---------------------
  z^2 - 1.941 z + 0.9608
 
Sample time: 0.1 seconds
Discrete-time transfer function.
```

We can see here that the global order of Hz is z^{-1}, , which expresses well that the influence of the input on the output will be perceived only after a sampling time.

Transfer functions properties in Matlab

The transfer functions obtained above are **objects** in Matlab. Like all Matlab objects (see the LiveScript on the subject), these objects all have a series of properties that can be modified :

```
get(H)
```

```
       Numerator: {[0 0 2]}
     Denominator: {[1 0.4000 2]}
        Variable: 's'
         IODelay: 0
      InputDelay: 0
     OutputDelay: 0
              Ts: 0
        TimeUnit: 'seconds'
       InputName: {''}
       InputUnit: {''}
      InputGroup: [1×1 struct]
      OutputName: {''}
      OutputUnit: {''}
     OutputGroup: [1×1 struct]
           Notes: [0×1 string]
        UserData: []
            Name: ''
    SamplingGrid: [1×1 struct]
```

```
get(Hz);
```

```
       Numerator: {[0 0.0099 0.0097]}
     Denominator: {[1 -1.9412 0.9608]}
        Variable: 'z'
         IODelay: 0
      InputDelay: 0
     OutputDelay: 0
              Ts: 0.1000
        TimeUnit: 'seconds'
       InputName: {''}
       InputUnit: {''}
      InputGroup: [1×1 struct]
      OutputName: {''}
      OutputUnit: {''}
     OutputGroup: [1×1 struct]
           Notes: [0×1 string]
        UserData: []
            Name: ''
    SamplingGrid: [1×1 struct]
```

The **'InputDelay'** property allows to add a dead-time to a previoulsy created transfer function, using the parameter TimeUnit to define the... time unit (by default : second) :

```
H1=H;
set(H1,'InputDelay',10) ;   % We add a 10 second dead-time on the existing 
H2 transfer function
H1
```

```
H1 =
 
                      2
  exp(-10*s) * ---------------
               s^2 + 0.4 s + 2
 
Continuous-time transfer function.
```

```
get(H1)
```

```
       Numerator: {[0 0 2]}
     Denominator: {[1 0.4000 2]}
        Variable: 's'
         IODelay: 0
      InputDelay: 10
     OutputDelay: 0
              Ts: 0
        TimeUnit: 'seconds'
       InputName: {''}
       InputUnit: {''}
      InputGroup: [1×1 struct]
      OutputName: {''}
      OutputUnit: {''}
     OutputGroup: [1×1 struct]
           Notes: [0×1 string]
        UserData: []
            Name: ''
    SamplingGrid: [1×1 struct]
```

One can get the same result by multiplying the transfer function by this exponential e^{-sT_m}, being a dead-time expression in Laplace's domain, having T_m as delay.

It is of course required to have previoulsy created the "s" variable as a transfer function.

```
set(H1,'InputDelay',0) ;      % The dead-time is resetted to 0.
H1=H1*exp(-s*10)              % We create a new dead-time of 10 seconds 
using this second method
```

```
H1 =
 
                      2
  exp(-10*s) * ---------------
               s^2 + 0.4 s + 2
 
Continuous-time transfer function.
```

Be careful, this operation of adding a pure dead time on an existing transfer function is not trivial: indeed, the use of the c2d command or the rational calculation using the tf object could bring some problems.

The instruction **tfdata** allows to recover the coefficients in "s" (or in z) of the numerator and denominator of an existing transfer function:

```
[num,den]=tfdata(H,'v')        % Note the use of 'v' to get the data into a
vector (by default in a cell)
```

```
num =
     0     0     2
den =
    1.0000    0.4000    2.0000
```

Finding and plotting a transfer function poles and zeros

As already mentioned, the poles of this transfer function correspond to the roots of the denominator (which can be real or complex conjugates), and characterize the dynamics of the system and its stability. Indeed, when we use the inverse Laplace transform to return to the time domain, the answer we obtain will be a sum of exponentials defined by the position of the poles in the Laplace domain:

- The real part of the poles will define the speed of decay or growth of the exponential : e^{t*p_r}, with p_r being the real part of the pole : One can understand that poles with a positive real part will give rise to increasing exponentials, and thus to a system which will not converge and will be unstable. A system will be stable if all its poles are located to the left of the imaginary axis in the complex plane. One can also observe that e^{t*p_r} can be classicaly written $e^{\frac{t}{\tau}}$ when dealing with differential equations, and that therefore the real part of a pole is the inverse of its time constant : the further the pole is from the imaginary axis, the faster the associated time constant will be, and the less its influence on the transient will be important. Conversely, a pole close to the imaginary axis will be considered as a dominant pole.
- The imaginary part of the poles will lead in the time domain to a complex exponential, or a sinusoid. This implies that if there are complex poles conjugated in the complex plane, it will lead us to have an oscillating response in the time domain. The imaginary part of the poles defines the pulsation of this sine. And the combination of the real and imaginary parts defines what the overshoot will be when responding to a step.

The poles can be obtained easily, with the **roots** command seen previously in the polynomial tutorial :

```
roots(den)
```

```
ans =
  -0.2000 + 1.4000i
  -0.2000 - 1.4000i
```

or via the **pole** command applied to the transfer function object :

```
pole(H)
```

```
ans =
  -0.2000 + 1.4000i
  -0.2000 - 1.4000i
```

Similarly, if the system had zeros, we could find them by looking for the roots of the numerator :

```
roots(num)              % We get an empty vector as the numerator has no zero
in this specific case.
```

```
ans =

  0×1 empty double column vector
```

or via the **zero** instruction which also returns the gain of the system.

```
[Z,G]=zero(H)
```

```
Z =

  0×1 empty double column vector
G = 2
```

Be careful, this gain is not necessarily the static gain of the system! For it to be, the transfer function should have been put in Bode form (the independent term of the denominator = 1)... We will talk about this later.

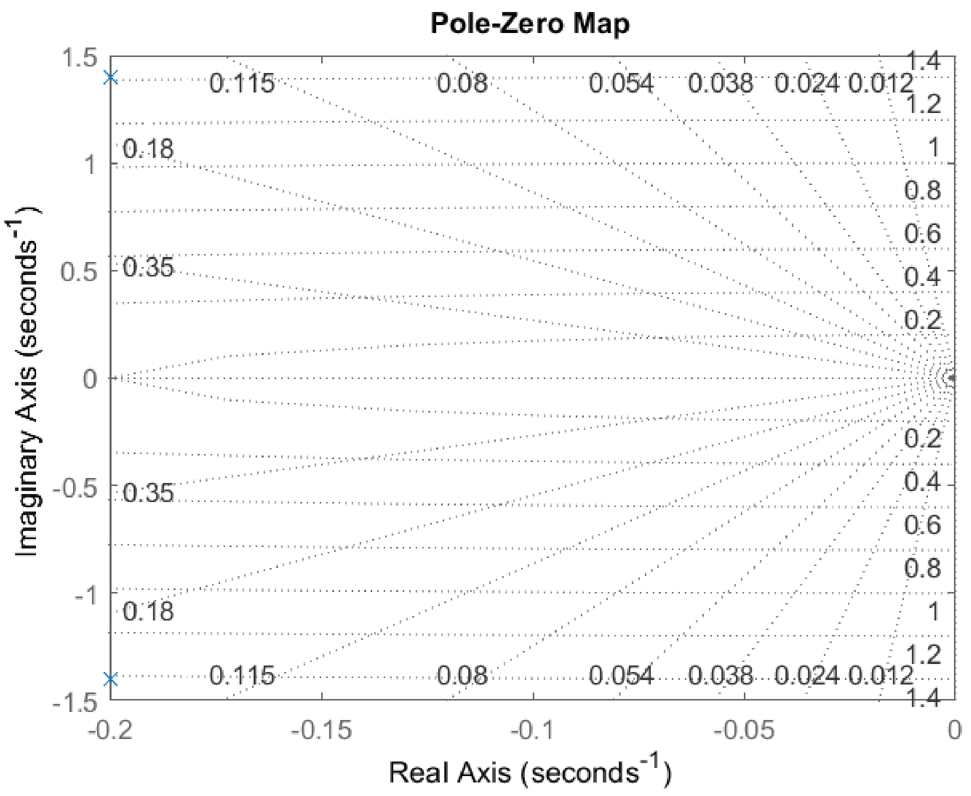
The instruction **zpkdata** allows in the same way to recover the zeros, poles and the gain of a transfer function :

```
[Z,P,K]=zpkdata(H,'v')
```

```
Z =

  0×1 empty double column vector
P =
  -0.2000 + 1.4000i
  -0.2000 - 1.4000i
K = 2
```

The **pzmap** instruction (or pzplot, which offers more customization options of the obtained graph) allows to draw the poles and zeros of a system in the complex plane :

```
pzmap(H);        % The previoulsy computed poles are shown here, with a "x"
mark.
grid;
```



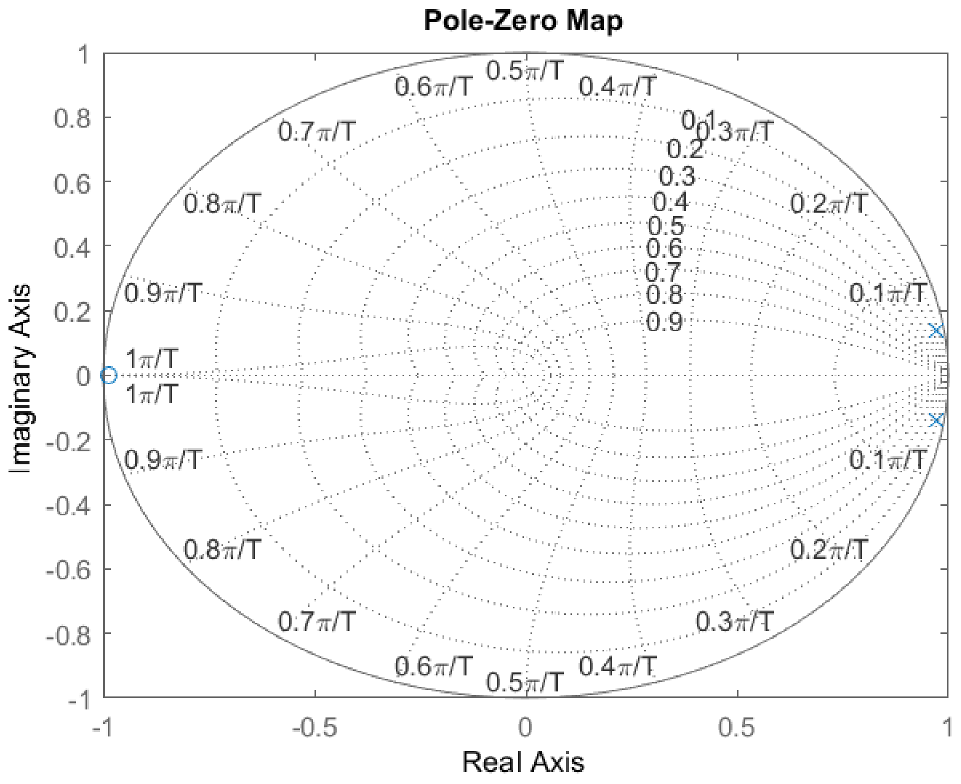
What if we work with a discrete system ?

We have the same tools to find and draw the poles and to draw. Remember that for a discrete system, the stability condition is that the poles are located in a circle of unit radius centered at (0,0).

```
pole(Hz)
```

```
ans =
   0.9706 + 0.1368i
   0.9706 - 0.1368i
```

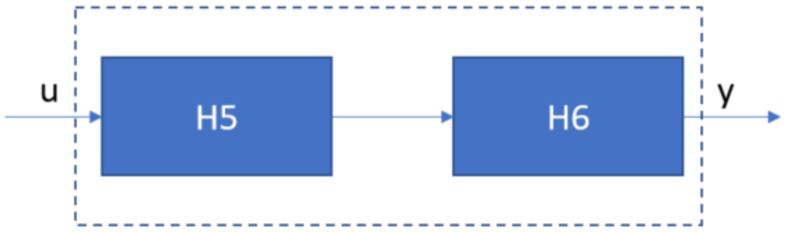
```
pzmap(Hz)             % We can observe that the poles are located inside a
unit circle centered at (0,0), so the discrete system is stable.
zgrid
```



Handling of LTI dynamic objects

When the output of the H1 system is connected to the input of the H2 system, the global transfer function is obtained by the series connection, thus mathematically by a multiplication of the 2 transfer functions.

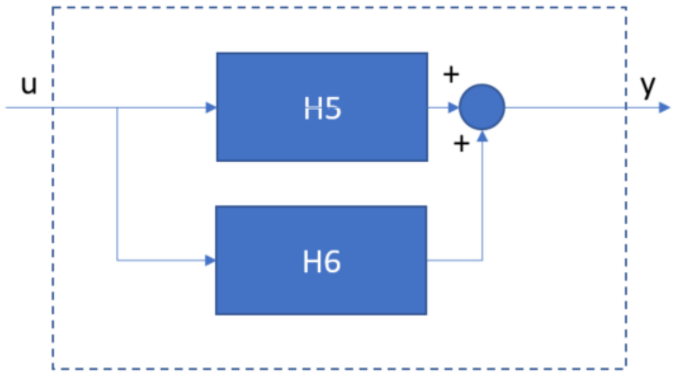
In the time domain, the convolution integral should have been used instead of this simple product in the Laplace domain.



```
H5=1/(5*s+1);
H6=1/(10*s+1);
H5*H6
```

```
ans =
 
          1
  ----------------
  50 s^2 + 15 s + 1
 
Continuous-time transfer function.
```

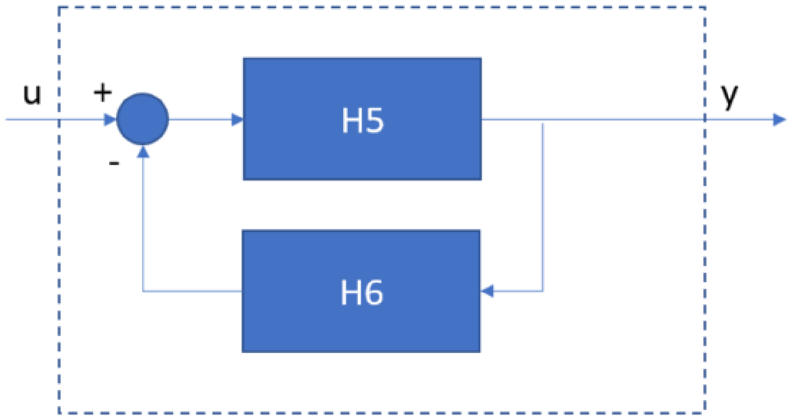
... their parallelism is obtained with the " + ":



```
H5+H6
```

```
ans =

      15 s + 2
  -----------------
  50 s^2 + 15 s + 1

Continuous-time transfer function.
```

When a feedback loop is closed, the Matlab **feedback** command is used to obtain the transfer function of the looped system:



```
feedback(H5,H6)
```

```
ans =

      10 s + 1
  -----------------
  50 s^2 + 15 s + 2

Continuous-time transfer function.
```

or without the feedback function :

```
H5/(1+H5*H6)    % ... same as the previous one after simplification.
```

```
ans =

       50 s^2 + 15 s + 1
  ----------------------------
  250 s^3 + 125 s^2 + 25 s + 2

Continuous-time transfer function.
```

When manipulating these transfer functions, Matlab does not simplify the poles and zeros on its own initiative when possible.

```
H2=(s+2)/((s+3)*(s+5));
H3=(s+3)/(s+4);
H3*H2
```

```
ans =

       s^2 + 5 s + 6
  ------------------------
  s^3 + 12 s^2 + 47 s + 60

Continuous-time transfer function.
```

This often occurs when multiple operations are performed on transfer functions. We can therefore simplify the expression, which is done with the **minreal** command:

```
minreal(H3*H2)
```

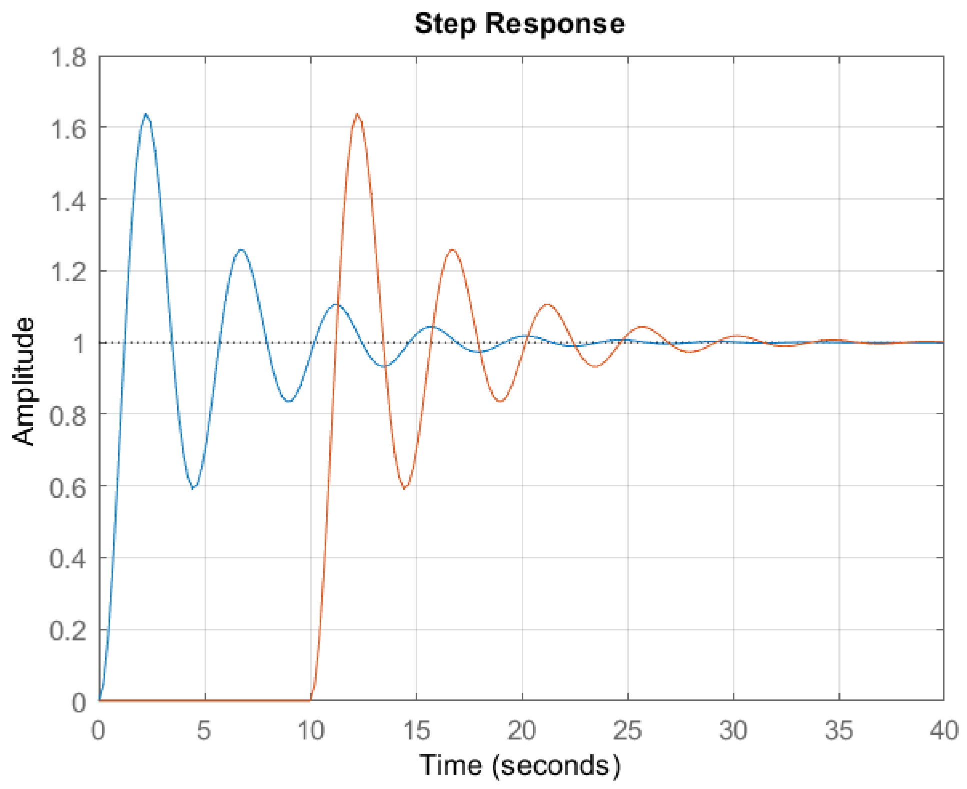
```
ans =

      s + 2
  --------------
  s^2 + 9 s + 20

Continuous-time transfer function.
```

Temporal responses to a step, an impulse and any input signal (lsim)

With the **step** command, the step response of the system will be calculated from t =0s until the transients are done, regardless of which tool the dynamic object was created with (zpk, tf, ...)

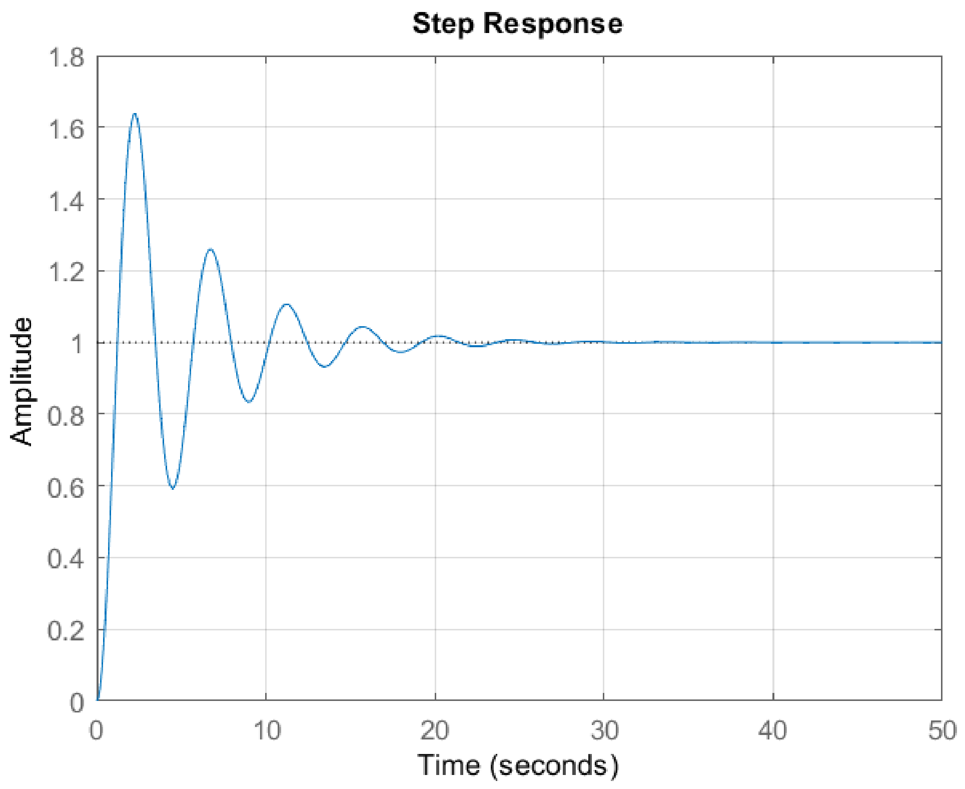
It is worth noting that the step command executes **a unitary amplitude step at time t=0s.**

```
step(H)          % Plotting the system step response without dead-timela
réponse indicielle sur un graphique du système sans temps mort
hold on
step(H1)         % The same system with a previously added dead-time of 10
seconds.
hold off
grid
```



By default, Matlab defines the length of the time vector itself, so as to show the whole transient. It is also possible to impose the time over which the step response must be calculated and plotted. The units that will be taken into account by Matlab for the time vector depend on the "TimeUnit" property of the transfer function object, which is by default expressed in seconds.

```
t=0:0.1:50;
step(H,t), grid
```



One can also get the data of the step response in 2 vectors t & y, instead of seeing the step response beeing plotted :

```
[y,t]=step(H)
```

```
y =
         0
    0.0475
    0.1802
    0.3784
    0.6180
    0.8730
    1.1182
    1.3315
    1.4956
    1.5993
      ⋮
t =
         0
    0.2221
    0.4443
    0.6664
    0.8886
    1.1107
    1.3329
    1.5550
    1.7772
    1.9993
      ⋮
```

NB: You will have noticed that the static gain of the system is 1, whereas the zero() instruction had announced a gain (not static) of 2: this gain was that of the numerator for the transfer function written in the Evans form (unit co-efficient for the most important power in s of the denominator).

It is good to remember that the static gain, no matter in which form the system is presented, can always be calculated by replacing s by 0 in the transfer function.

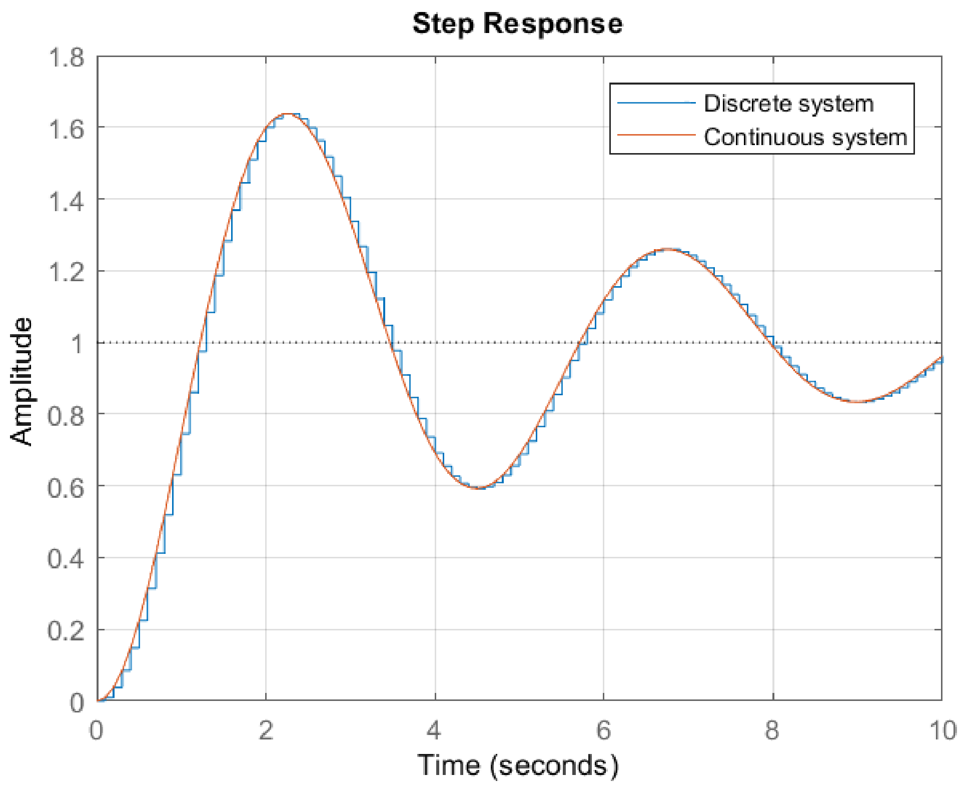
What about the discrete time response?

The step response of a discrete system can also be plotted using the same **step** command :

```
Hz
```

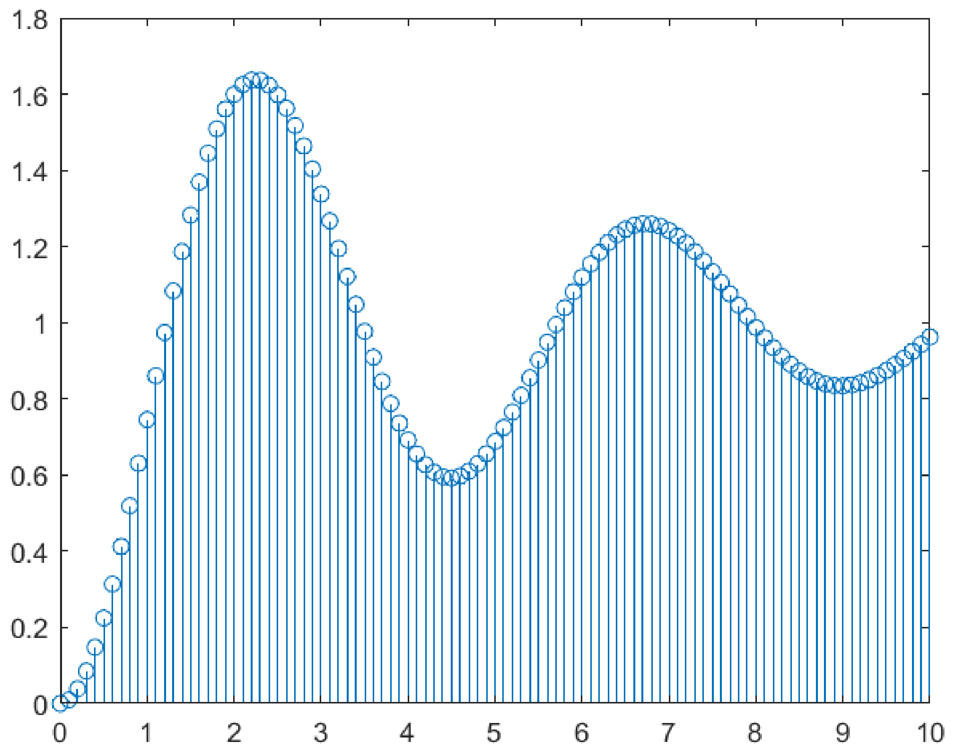
```
Hz =
 
  0.009852 z + 0.009721
  ----------------------
  z^2 - 1.941 z + 0.9608
 
Sample time: 0.1 seconds
Discrete-time transfer function.
```

```
t1=(0:0.1:10)';     % Remember the sampling time used to pass this system
from continous to discrete was 0.1 second
step(Hz,H,t1), grid, legend('Discrete system','Continuous system')
```



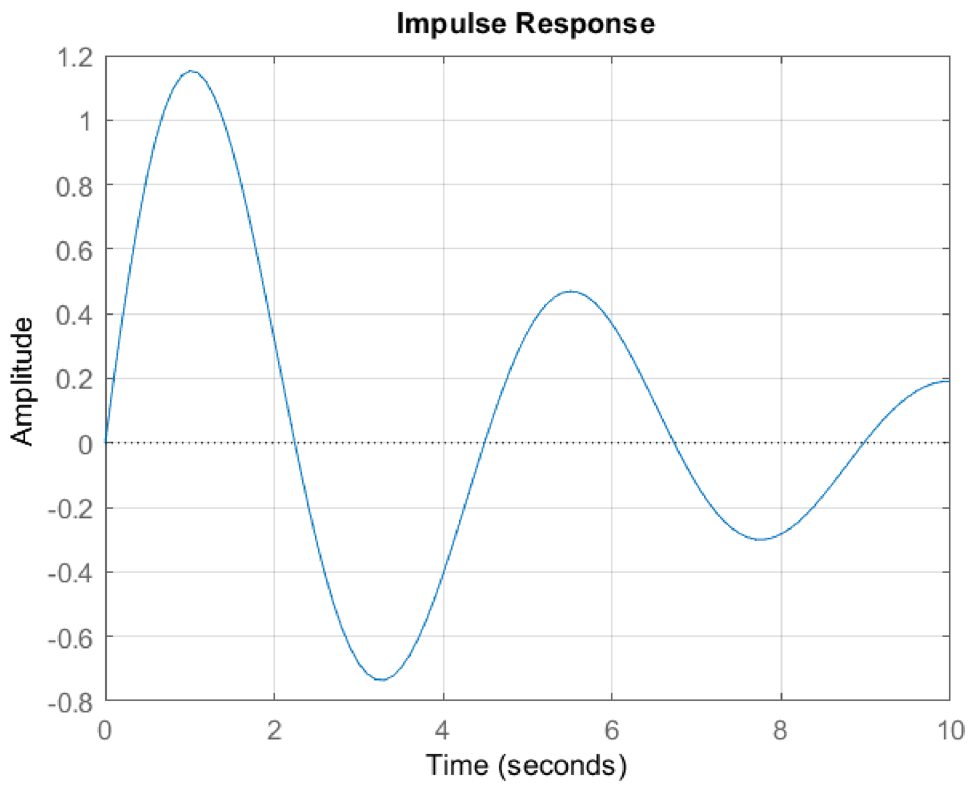
One can see the sampling time (0.1 second) on the step response, but also the similarities between the continuous and discrete systems. Note that the discrete response is late compared to the continuous one, the delay beeing the half of the sampling period. It should also be noted that the values are not known between 2 sampling times. A more correct representation of the actually known information would be obtained by showing the values at the sampling times only, using the **stem** command:

```
yz=step(Hz,t1);
stem(t1,yz)
```



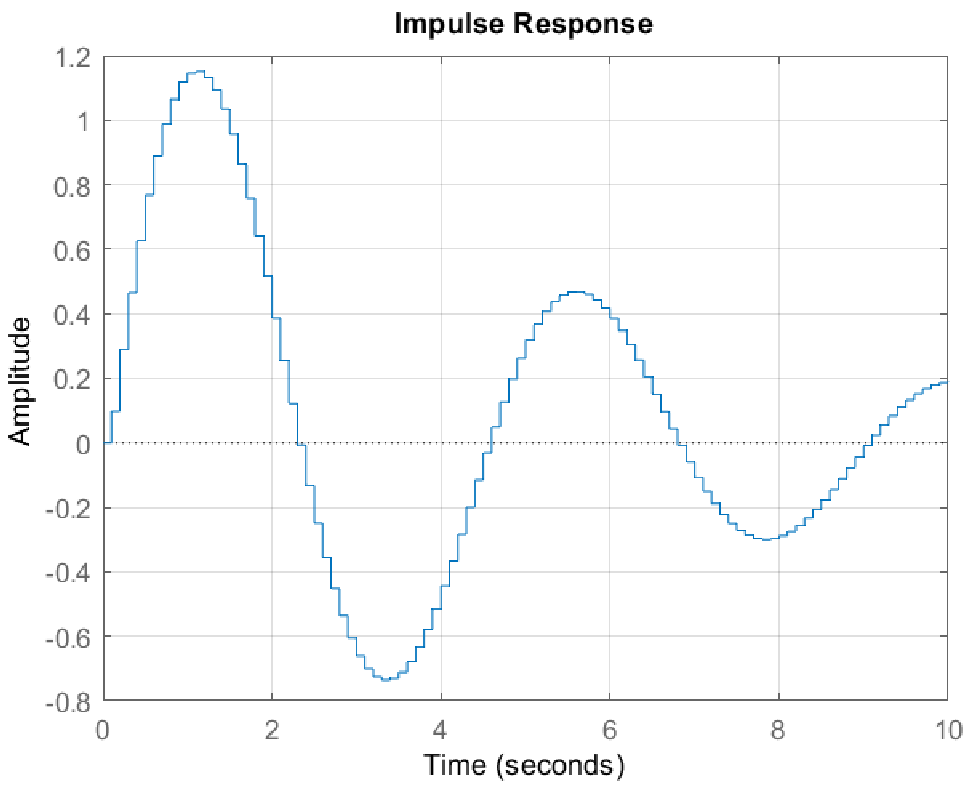
The **impulse response** is obtained with the **impulse** command :

```
impulse(H,t1), grid
```



and with the same instruction for a discrete system :

```
impulse(Hz,t1), grid
```

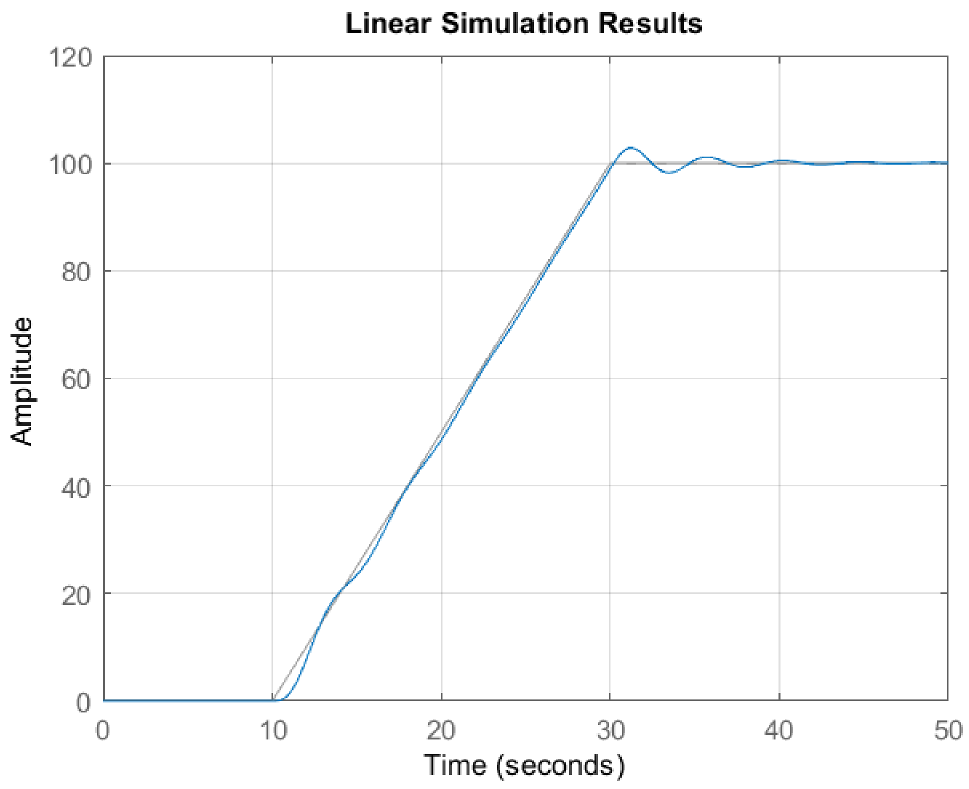


Time-related response to any signal

Matlab integrates functions to obtain the temporal responses to Dirac impulse or unit step solicitations. But how to obtain the response to any solicitation?

We will have to use the **lsim** instruction, having previously created the input signal manually. This way of doing things can be used to calculate the response to any signal

```
t=0:.1:50;        % creation of a ramp of slope 5 starting at t=10s and
maintaining its value from t=30s
u=5*t-50;
u(t<10)=0;
u(t>=30)=5*30-50;

lsim(H,u,t), grid
```



Exercise

We now suggest to solve the next exercise that deals with the creation and manipulation of transfer functions.

```
open 12_1_FT_Manipulation
```

Exercise

We now propose to do an exercise that deals with the identification of a system according to a model (a transfer function) imposed by its step response.

```
open 12_2_idf_FT_VDG
```

Frequency response: the Bode plot

In addition to the temporal analysis of systems, the harmonic (or frequency) analysis allows to obtain information about the behavior of a system that the temporal analysis can not give directly:

- The frequency analysis of the system in open loop is frequently used to deduce the behavior it will have in closed loop,
- The frequency analysis allows to quantify the stability of a system constituted with a feedback (closed loop),

- Knowing that any signal can be reconstructed as a sum of sines of different frequencies, phase and amplitude, the frequency analysis allows to understand some temporal behaviors.

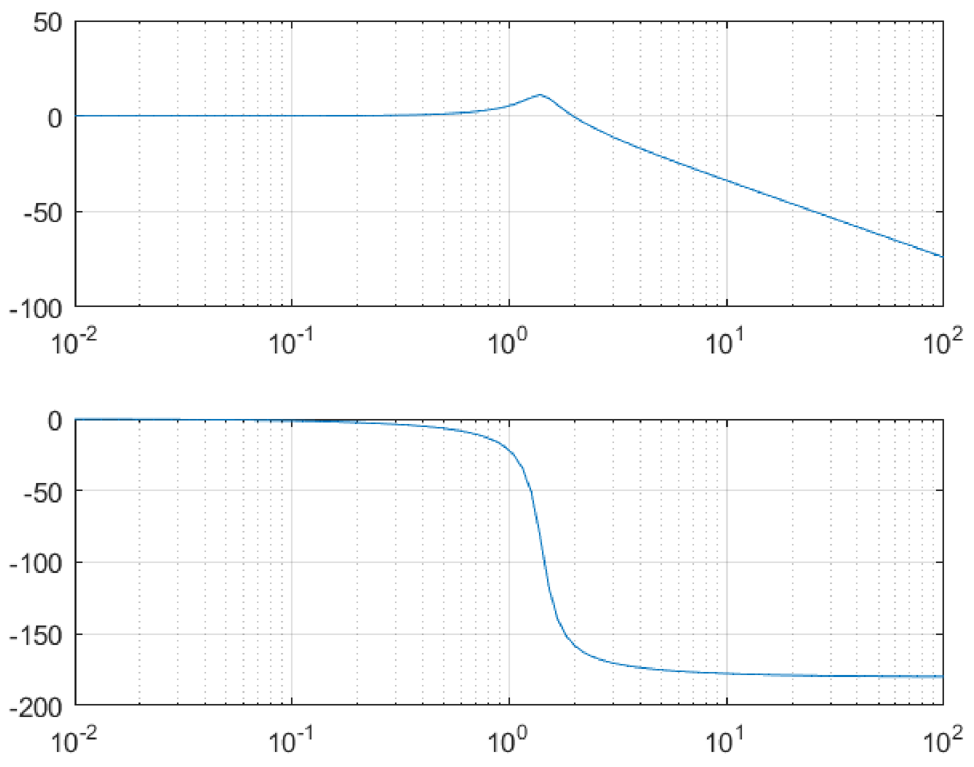
The principle of frequency analysis of a system is as follows: the system is solicited with a sinusoidal input. Knowing that the system is LTI, we know that we will obtain a sinusoidal signal (oscillating at the same frequency as the input signal) at the output of this same system, but that it may have a different amplitude from the input signal, and that a phase shift may appear between this output signal and the input signal. These 2 phenomena are directly linked to the time constants of the system, which, as a reminder, are an image of its inertia. We will therefore analyze how the amplitude ratio of the signals, called the gain (amplitude of the output signal compared to that of the input signal), and how the phase shift evolves when we vary the frequency of the input signal while keeping its amplitude constant.

It is important to note that this analysis is done in "steady state": we consider that the sinusoidal input signal of the system is present for a sufficiently long time so that all the transients of the system are over.

This analysis is made easier by the use of transfer functions: indeed, it is enough to replace "s" by " $j\omega$", which generates a complex fraction. Then to see how the norm and the phase of this one evolve when ω varies. Let's do it for our example:

Replacing "s" by " $j\omega$" in the transfer function, we get the next complex fraction : $\frac{2}{(j\omega)^2 + 0.4\,(j\omega) + 2}$.

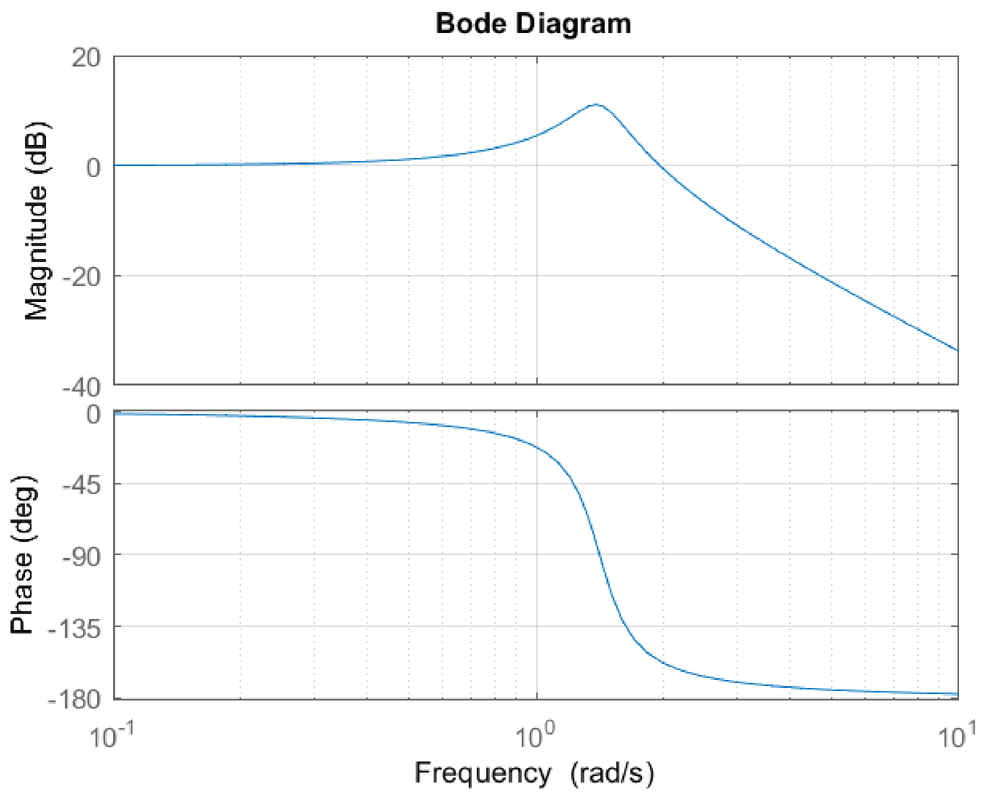
```
w=logspace(-2,2,100);              % Vector of the frequencies for which I
want to know the behavior of the system
c=2./((1j*w).^2+0.4*(1j*w)+2);    % Matrix of complex fractions for the
corresponding frequencies
m=rad2deg(angle(c));               % Phase in degrees
g=20*log10(abs(c));                % Gain in dB
subplot(2,1,1); semilogx(w,g), grid
subplot(2,1,2); semilogx(w,m), grid
```



We have manually rebuild the Bode diagram, which represents the way the gain and phase of the output signal of a system evolve as a function of the frequency of the input signal.

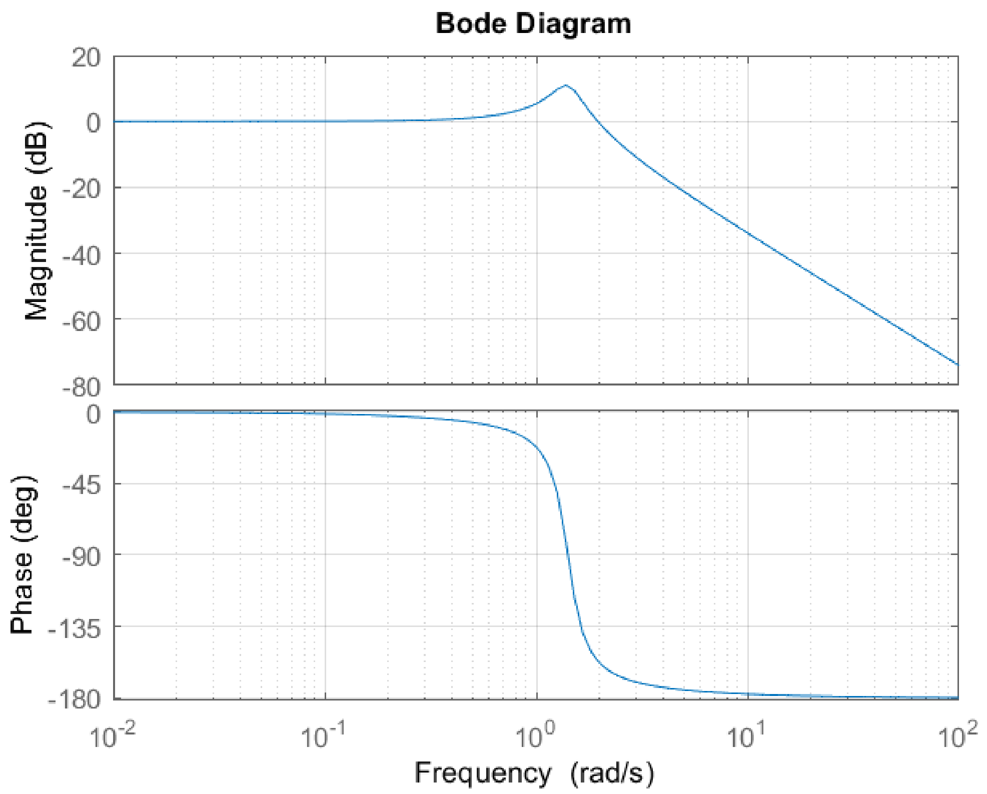
In Matlab, you can use the **bode()** command to directly get the same result:

```
figure
bode(H)          % the Bode diagram, in gain and phase.   Try bodeplot(H),
more versatile
grid             % as a reminder, the gain is in dB (20*log10(y)) and the
horizontal axis is logarithmic
```



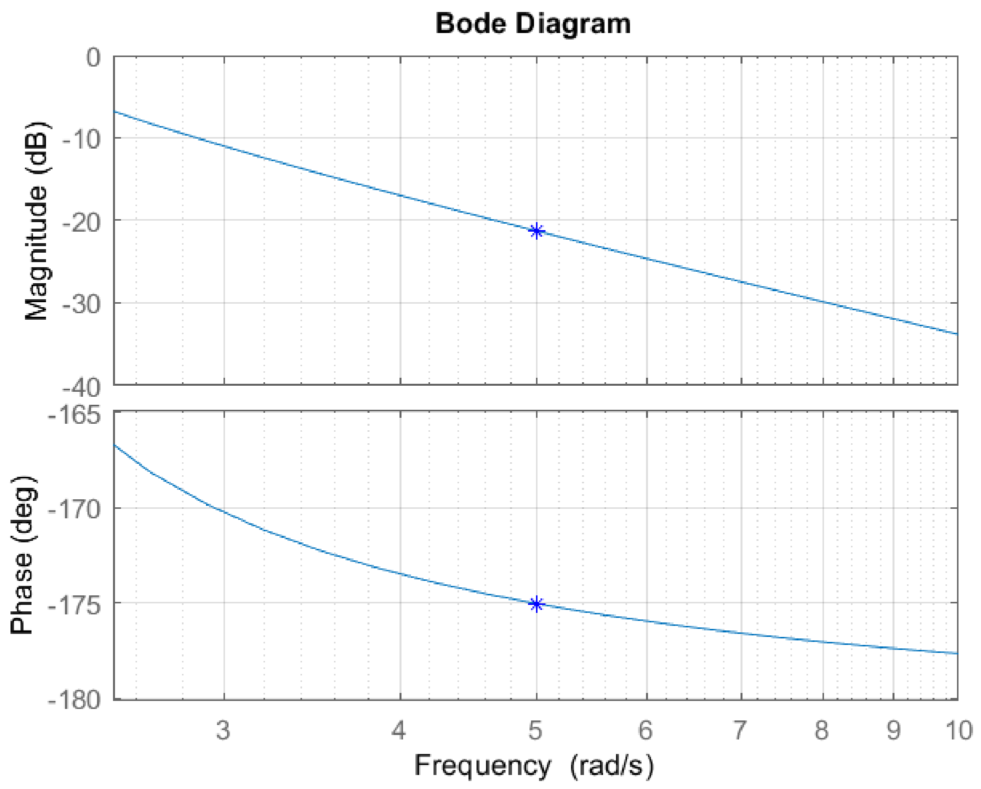
As for the temporal responses, it is possible to impose the frequencies at which we want to plot the Bode diagram:

```
bode(H,w), grid
```



And also to recover the phase and gain for a given frequency:

```
w_spec = 5;
bode(H,w), grid, hold on
bode(H,w_spec,'*')
hold off
```



```
[mag,phase1] = bode(H,w_spec)  % The gain is not given in dB! NB: we are
advised not to use as variable "phase" which is a native function of Matlab,
otherwise we lose it!
```

```
mag = 0.0866
phase1 = -175.0303
```

```
magdB=20*log10(mag)            % Verification by expressing it in dB
```

```
magdB = -21.2467
```

This allows us to read the value of the static gain of the system, which is read at the zero frequency :

```
[mag,phase1] = bode(H,0)
```

```
mag = 1.0000
phase1 = 0
```

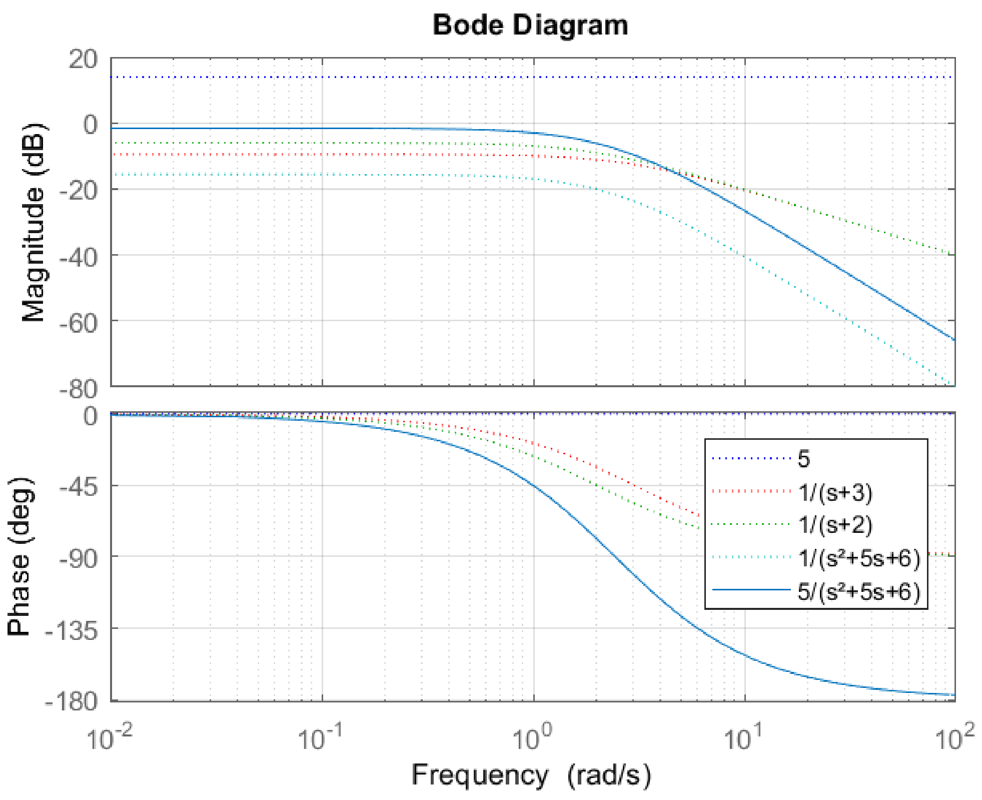
We can see that the static gain of the system is equal to 1 in our case. Remember that it is because the Bode diagram is drawn for a sinusoidal **steady state** regime that the fact of imposing a zero frequency can read the static gain.

One of the great interests of the representation of systems in the Bode diagram is that it is possible to simply sum the curves of the different elements composing the direct chain (the open loop) of the complete system. This is made possible by the use of the logarithmic axis in this Bode diagram. Indeed, we remember that log(ab)

= log(a) + log(b). This property applies only to the Bode diagram, not to the Nyquist diagram which will be described below.

Thus, the frequency response of a transfer function $G(s) = \frac{5}{s^2 + 5s + 6}$ which can be written as the product of 3 independent transfer functions 5, $\frac{1}{s+3}$ and $\frac{1}{s+2}$ can be easily plotted by drawing the diagrams corresponding to the 3 simple functions and summing them:

```
bode(tf(5,1),':b'), hold on
bode(tf(1,[1 3]),':r')
bode(tf(1,[1 2]),':g')
bode(tf(1,[1 5 6]),':c')
bode(tf(5,[1 5 6]))
legend('5','1/(s+3)','1/(s+2)','1/(s²+5s+6)','5/(s²+5s+6)')
grid
hold off
```



Time for an exercise using the Bode plot and illustrating that it provides the harmonic response only in steady state:

```
open 12_3_sine_wave_sollicitation
```

Let's start a new exercise: solving an electric circuit in steady state and transient:

```
open 12_4_AC_Electrical_circuit_2
```

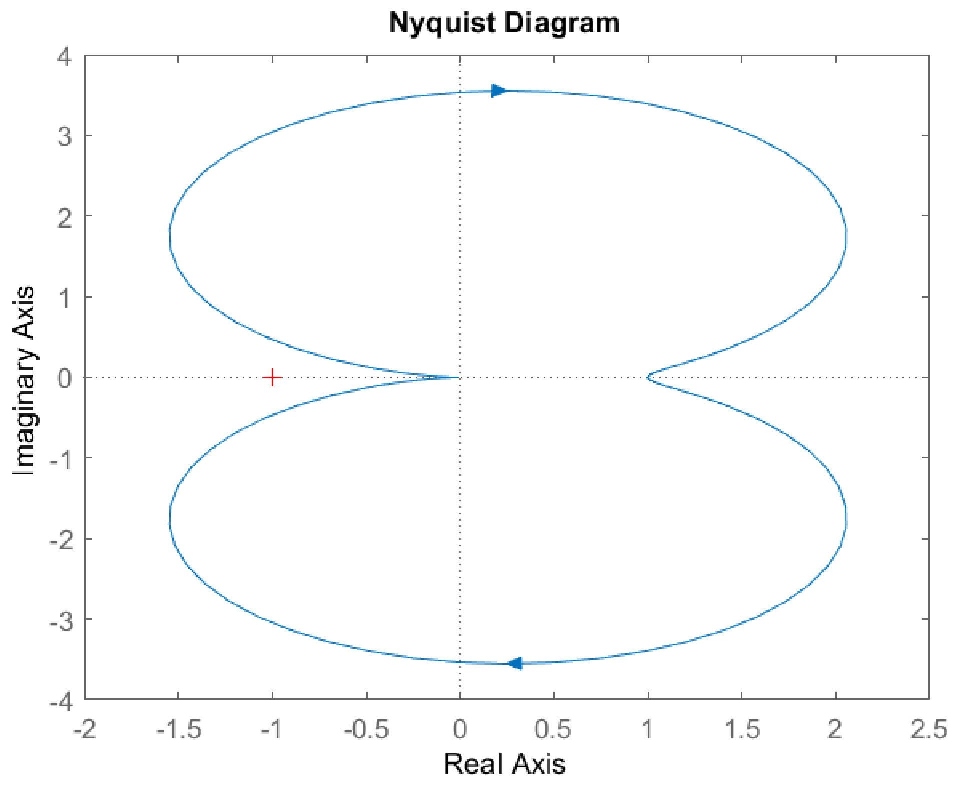
Nyquist plot :

The Nyquist diagram represents exactly the same thing as the Bode diagram, but instead of representing the gain and phase as a function of frequency, it is a polar representation on a single curve.

We can imagine that it is the plot of the end of a rotating vector whose origin is placed in [0;0] and whose amplitude and phase vary with frequency. We specify with an arrow on the curve obtained in which direction to follow a growing ω. It should be noted that the gain is not calculated in dB, unlike the Bode diagram.

In the case of our system, for a zero frequency, the gain is unity and the phase zero. And for an infinite frequency, the gain is zero and the phase is -180°. This can be seen on the following diagram.

```
nyquist(H)        % diagram of the gain versus the phase (where the
frequency does not appear)
```



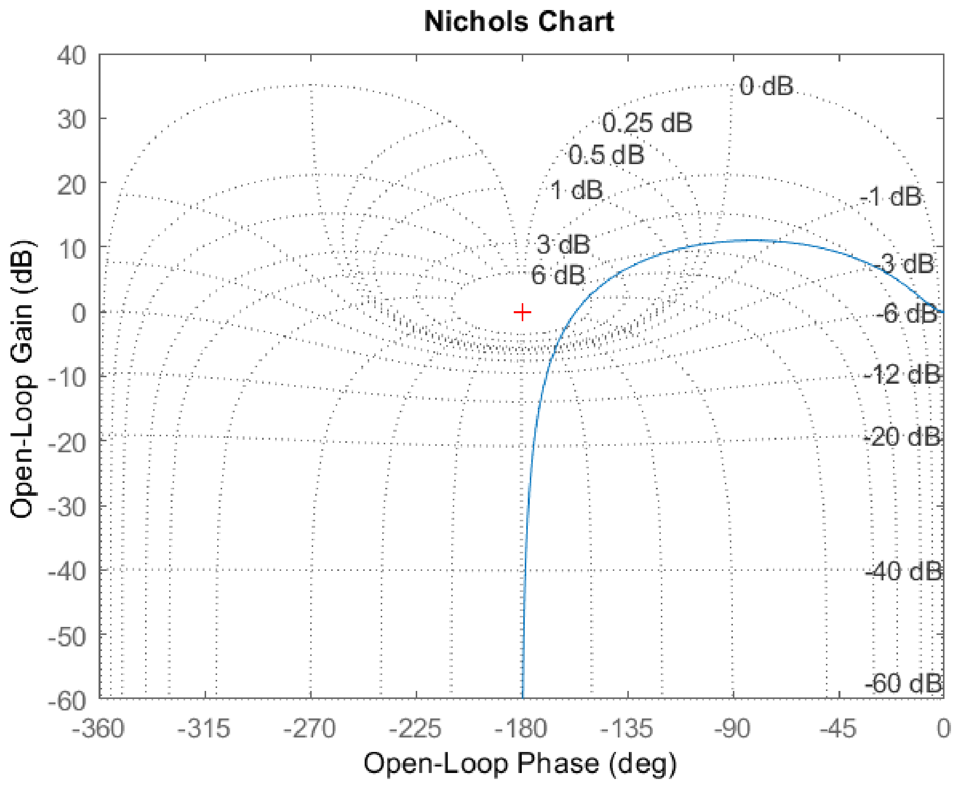
Time for a new exercice combining Bode and Nyquist plots :

```
open 12_5_Nyquist
```

Black-Nichols plot

Last representation of the frequency response of a system, the Black-Nichols diagram is drawn in a Cartesian graph with in abcissa the phase of the system, and in ordinate the gain expressed in dB, for an increasing pulse.

```
nichols(H), grid
```



Example: Study of the current transient of a non-ideal choke connected to the network, at a phi phase of the sinusoid

```
f=50;               % Network frequency
w=2*pi*f;
L=0.005;            % Self [F]
R=0.05;             % Resistance [R]
Zself=R+1i*w*L;     % Impedance for w = 2*pi*f
Yself=1/Zself;
U=240;              % Voltage
I=U*Yself           % Current ==> Almost only reactive.
```

```
I =
   4.8585e+00 - 1.5263e+02i
```

```
abs(I)              % Its a complex number, let's compute its norm
```

```
ans = 152.7114
```

This value could have been obtained by creating the transfer function corresponding to this system, and looking at the harmonic response for the desired pulse:

```
G=tf(1,[L R])
```

```
G =

        1
  --------------
  0.005 s + 0.05

Continuous-time transfer function.
```

```
[mag,phase1] = bode(G,100*pi)
```

```
mag = 0.6363
phase1 = -88.1768
```

The current amplitude is therefore the gain multiplied by the voltage amplitude:

```
U*mag
```

```
ans = 152.7114
```

This current is correct but only in the steady state, i.e. after the inrush current. As mentioned earlier, the harmonic analysis via Bode, Nyquist, etc. tools only concerns the steady state.

However, the question arises as to how long it takes to reach the steady state?

Let's study the Inrush current for a phase pi of the voltage at the onset by means of the transfer function, but in time then, by means of the **lsim** function (it is not possible to do otherwise this time)

```
t=linspace(0,.3,400);
phi=pi;                                  % the phase of the voltage at the time
of switching on

u=240*sqrt(2)*sin(100*pi*t+phi);    % the instantaneous network voltage

s=tf('s');
Yself=1/(R+s*L);
a=lsim(Yself,u,t);                       % a = the current = u*Y

figure
plotu=plot(t,u,'b'); hold on        % Blue color choosen for the voltage
plotu                                    % plotu is an object of type "line",
having a few properties
```

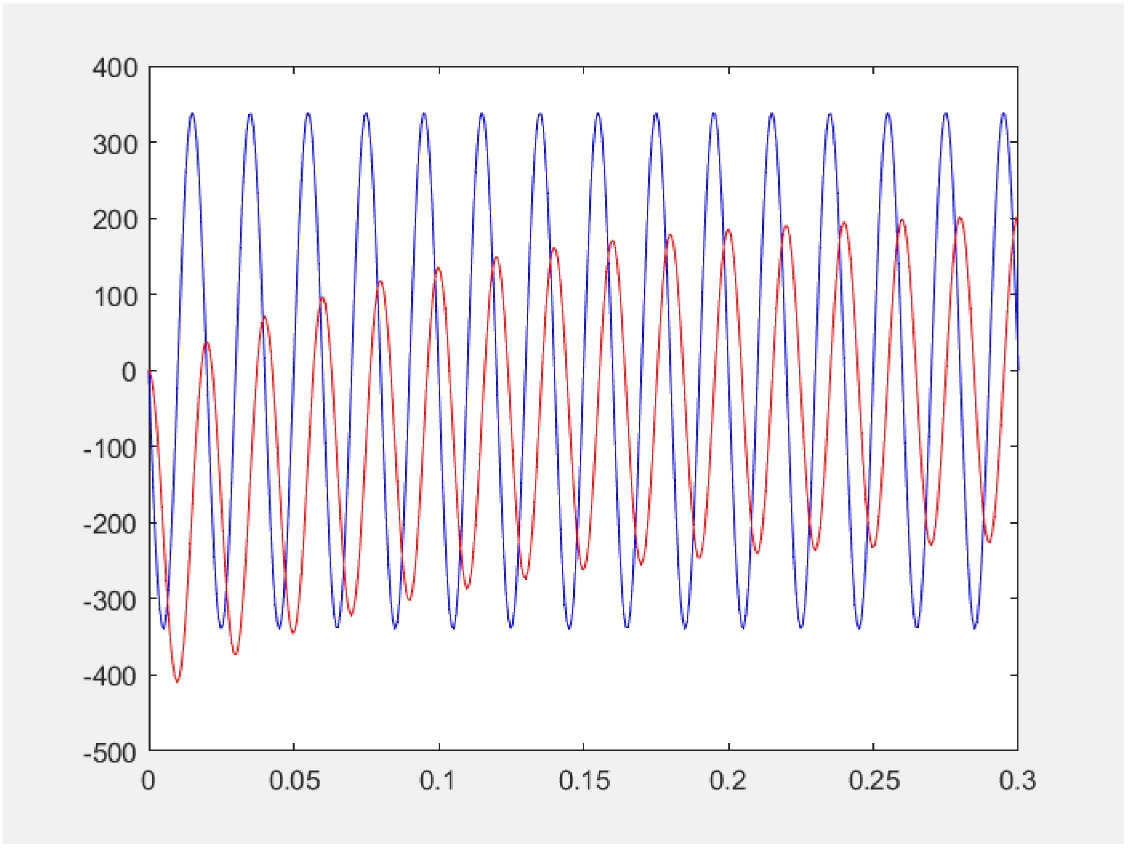
```
plotu = 
  Line with properties:

              Color: [0 0 1]
          LineStyle: '-'
          LineWidth: 0.5000
             Marker: 'none'
         MarkerSize: 6
    MarkerFaceColor: 'none'
              XData: [1×400 double]
              YData: [1×400 double]
              ZData: [1×0 double]
```

```
Show all properties
```

```
plota=plot(t,a,'r'); hold off,       % Red color for the current
```

The current undergoes here a transient which gives it a continuous component during a certain time.

```
set(gcf,'visible','on')             % To plot on an external figure
```



Animation for phi varying from -pi to pi:

What would happen if we connected the self at another time (another phase of u(t))?

Let's animate this:

```
% for th=linspace(2*pi,-2*pi,600)                        % we vary the phase
from -2*pi to 2*pi
%     u=240*sqrt(2)*sin(100*pi*t+th);                    % computing the
instaneous values of the voltage
%     set(plotu,'ydata',u)
%     a=lsim(Yself,u,t);                                 % computing the
instaneous values of the current
%     set(plota,'ydata',a)
%     grid, xlabel('t [s]'), ylabel('u[v] i[A]')         %
%     drawnow                                            % To force
theupdate of the plotted figure
%     figure(gcf)
% end
%close all
```

NB: for help on handling objects as done in this example, see the appendices

Matlab (but not Octave) proposes a rather interesting tool to visualize and study the different typical answers : **ltiview** (to try ... at home)

© JM Rousseau - D Rouchard