

Introduction to Computer Control Systems

MATLAB Mini-tutorial

Rubén Cubo

October 2014

Contents

1	Introduction	2
1.1	What is MATLAB?	2
1.2	Interface	2
2	Basic elements and operations	4
2.1	Numbers, vectors and matrices	4
3	Basic commands	6
3.1	Mathematics	6
3.2	Plotting	7
3.3	Logic: if-else	9
3.4	Loops: for-while	11
4	Control applications	12
4.1	Defining a system	13
4.2	Responses to different signals	14

Foreword

This document is meant as a small tutorial to get the reader started with MATLAB, starting with the basics. No experience with MATLAB or any other programming language will be assumed during this tutorial, but some knowledge about mathematics (matrix algebra in particular) will be assumed. It will be also assumed that MATLAB is already installed in your computer. If you need help to get it started, you can go to <http://www.mathworks.se/help/install/index.html>. I encourage you to, even though there is an example at the end where most of the contents will be used, use the program to practice everything step by step by yourselves.

Its aim is for the students of Introduction to Computer Control Systems, so applications to control theory will be described. Please take into account that this is just to get the reader started and is in no way a substitute for MATLAB's extensive documentation.

1 Introduction

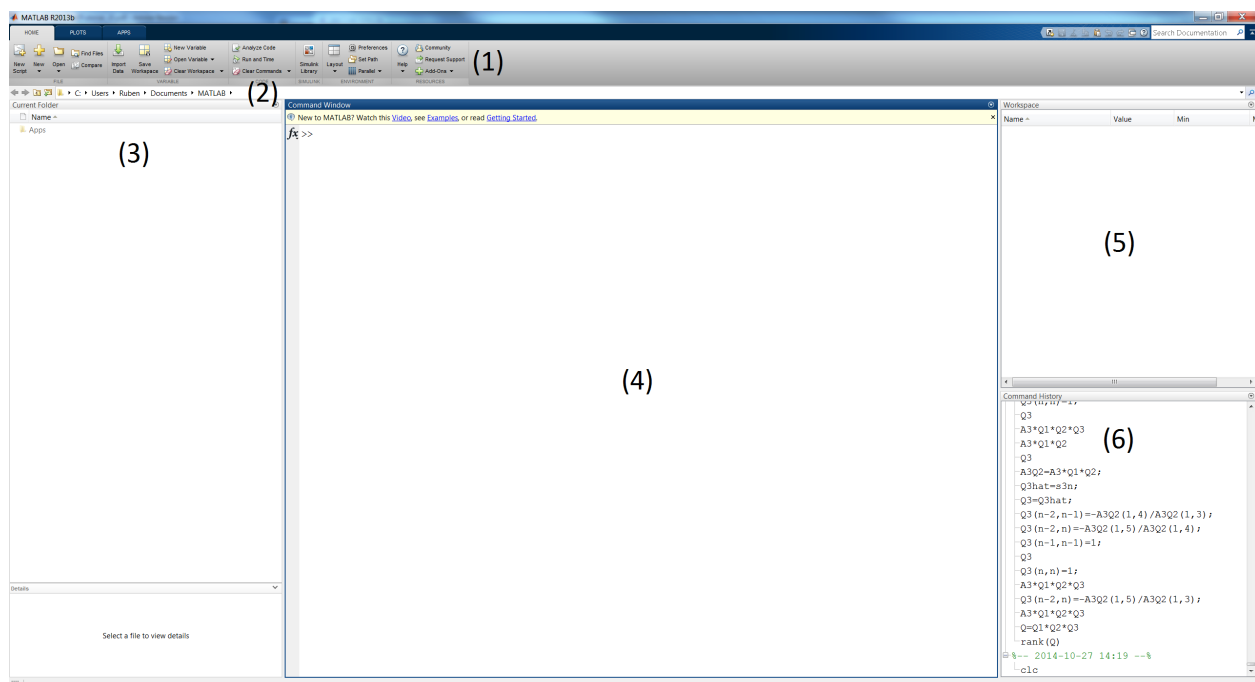
1.1 What is MATLAB?

MATLAB is a program suite used for scientific computing and engineering. One can think of it as a very capable calculator, with lots of packages which can do lots of cool stuff.



Using MATLAB can be difficult at the beginning. The next sections of this tutorial will show the basics of MATLAB, main commands, functions and examples which will hopefully ease the learning curve, turning MATLAB into a powerful tool for your purposes.

1.2 Interface

The first time it's started, you will see something like this:



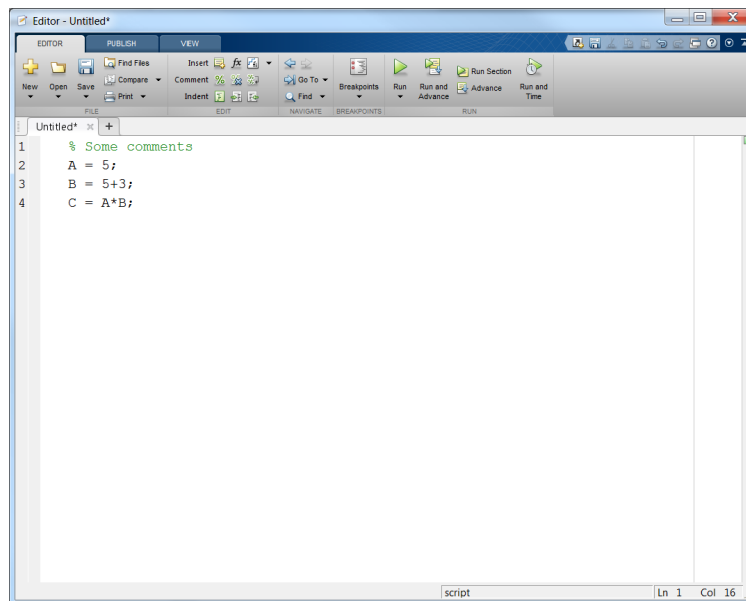
where:

1. **Ribbon:** This was added recently to MATLAB and lets you access the most common features of the program, such as creating new files, new variables, plotting or importing data. However for most of them you'll write commands instead, since it's faster that way.
2. **Active path:** This is the directory where MATLAB can see user-made files. You can change it by hand by clicking on the blank space, go up with  (third button to the left of the path) and browse other directories with  (fourth button to the left of the path).
3. **Current folder:** Shows what's inside the folder in the Active Path. You can access your MATLAB scripts or data files from there

4. **MATLAB Prompt:** You can introduce commands here and will be executed by MATLAB instantly. You can also see the results of your computations here if you want to. Its use for extensive sets of instructions (i.e. more than a few lines) is discouraged however.
5. **Workspace:** Where the variables that you will generate will be showed. You can see its name, its value (or type, depending on the situation) and its minimum and maximum.
6. **Command History:** You can see here which commands have you used in the past. You can execute them by double-clicking or you can copy-paste them in the prompt or the editor.

MATLAB uses an interpreted language, that is, you don't need to compile it unlike other languages like C. You can access the help of a function or command by typing **help function**. This is extremely useful, since usually one does not remember exactly the syntax of a function. You can also clear variables by using **clear variable_name**, or just clear all the workspace with **clear all**. Be careful when clearing since you cannot undo it.

To start making a MATLAB script, you can press the button *New Script* in the ribbon. A window will then open:



This is what you will use to write most of your code. It can be changed easily and will not vanish if the program freezes (happens more often than you would think...).

In the example above you will notice that there are semicolons with the command lines. These are used to prevent MATLAB from showing the results of every step in the Prompt, which can be annoying and even crippling (think about for example showing a matrix of 2 million elements).

To introduce comments in your code (probably not needed in this course, but it will make your life and other people's lives easier in the future), just use a % sign before the line

you want to comment. If you want to write more than one line of comments, it's better to write it as it is, select it, and then press *Ctrl+R* to comment that block. This also works if, for example, you want to block some code from executing. You can take out the comments by selecting what you want to uncomment and press *Ctrl+T*. Alternatively, you can use the ribbon buttons.

You can run whole scripts by pressing the *Run* button or F5 (note that MATLAB will save the file automatically if you do this). You can evaluate pieces of code by selecting them and pressing F9, which is particularly useful for testing and diagnostics.

2 Basic elements and operations

2.1 Numbers, vectors and matrices

The most basic element in MATLAB is a scalar. You can assign a scalar to a variable with an equality sign, for example, `A=5` will assign the variable A the value 5. The next element is a vector, which is a 1-D array of numbers. You can assign them with square brackets `[,]`. For example, `A=[3 8]` will assign the vector (3, 8) to the variable A. Finally, to define a matrix, you can separate its rows with semicolons, for example, the code `A=[0 1; 4 7]` will assign the following matrix to A:

$$A = \begin{pmatrix} 0 & 1 \\ 4 & 7 \end{pmatrix}$$

Special Matrices

There are some special matrices that can be generated by MATLAB easily, such as:

- `zeros(m,n)` will generate a $m \times n$ matrix full of zeros
- `eye(m,n)` will generate a $m \times n$ identity matrix
- `ones(m,n)` will generate a $m \times n$ matrix full of ones
- `rand(m,n)` will generate a $m \times n$ matrix with random numbers between 0 and 1 following an uniform distribution
- `randn(m,n)` will generate a $m \times n$ matrix with random numbers following a gaussian (or normal) distribution of mean 0 and standard deviation 1

Note that you can specify one of the numbers as well, for example by writing `zeros(n)`, but it will yield a $n \times n$ matrix instead of a vector. To generate a vector, you should use for example `zeros(1,n)`

Basic operations

All matrix algebra can be done in MATLAB. You can sum, subtract, multiply matrices with the usual signs `(+,-,*)`. Be aware that the dimensions of the involved matrices must

be adequate, so for example you can multiply a 3×2 matrix with a 2×4 , but not a 3×2 matrix with a 4×2 ! Other operations are elevating a matrix to a number (\wedge) and dividing ($/$).

Some other matrix operations are:

- Transposing: You can do it by adding an apostrophe (') to the variable. For example:

```
A=[0 1; 4 7]';
A
A =
     0     4
     1     7
```

- Inverting: Can be done by using the inv command, so `inv(A)` will yield A^{-1} . You can also use A^{-1} .
- Element-wise operations: These are useful if you want to, for example, compute the square of the elements of a matrix. You only have to add a dot (.) before the corresponding operand. For example (note the difference):

```
A=[0 1; 4 7]';
A.^2
ans =
     0    16
     1    49
A^2
ans =
     4    28
     7    53
```

This can be used to multiply or divide element-wise two matrices (instead of the usual matrix product)

Element access and modifications

To access an element of a matrix (let's say $A_{i,j}$), just type `A(i,j)`. For a vector, you only need to specify one of the numbers so for example if you want to access the element x_k , you can type `x(k)`. This will be important when you make loops, since you can also assign values to elements of a matrix.

For example:

```
A=[0 1; 4 7];
```

```
B=[5 7 9];
```

```
A(2,2)=4;
```

```
B(3)=A(2,1);
```

```
A
```

```
A =
```

```
    0    1
    4    4
```

```
B
```

```
B =
```

```
    5    7    4
```

Alternatively, if you want to access all the rows or all the columns, you have to substitute the elements inside the parenthesis with `..`. For example:

```
A=[0 1; 4 7];
```

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

```
ans =
```

```
    0
    4
```

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

```
ans =
```

```
    0    1
```

It is also possible to modify the elements of a whole row or a whole column in one line by using this. However, be aware of dimensions! You can't overwrite for example a row of size n with a column vector or with a scalar. It needs to be a row vector of size n ! In the above example using `A(:,1)=[3 5]` or `A(:,1)=3` will not work, but `A(:,1)=[3;5]`, `A(:,1)=[3 5]'` or `A(:,1)=3*ones(2,1)` will work flawlessly.

One last thing: Be aware of the kind of variable you are working with! Trying to access the (i,j) element of a vector will yield a harmless error, but trying to access a (j) element of a matrix won't if it exists, so for example, `A(3)` will yield 1 instead of giving an error. In case of doubt, check the workspace or print your variable.

3 Basic commands

3.1 Mathematics

MATLAB come with a large variety of mathematics embedded in it. Some examples are (all of them can be checked by typing `help elfun`):

<code>sin(x)</code> (sine)	<code>cos(x)</code> (cosine)	<code>tan(x)</code> (tangent)
<code>asin(x)</code> (arc-sine)	<code>acos(x)</code> (arc-cosine)	<code>atan(x)</code> (arc-tangent)
<code>exp(x)</code> (e^x)	<code>log(x)</code> , <code>ln(x)</code> (logarithm)	<code>eig(A)</code> (eigenvalues of A)
<code>abs(x)</code> ($-x-$)	<code>sqrt(x)</code> (\sqrt{x})	<code>norm(x)</code> ($\ x\ _2$)

Note that all of these functions (except the norm) are taken element-wise, so for example `exp(A)` won't yield e^A , but the exponentials of its elements, unlike for example the power operand `^`. Also, the trigonometric functions **always** work with radians, not with degrees!

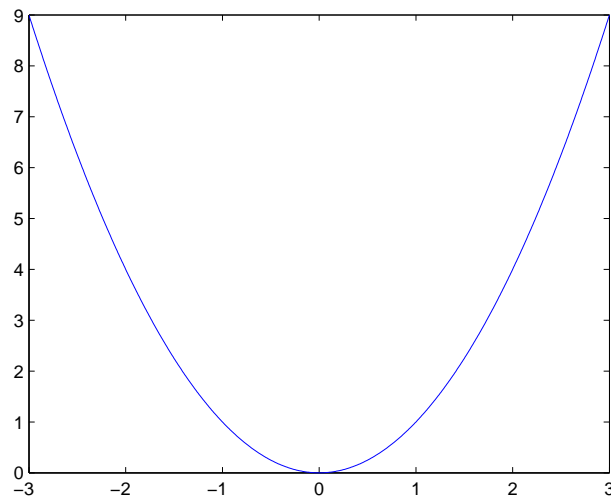
3.2 Plotting

There are many kind of plots that can be done in MATLAB. Here, only the 2D scatter/line plots will be explained because of its simplicity, but there are many other kinds.

The basic instruction for plotting is: `plot(xdata,ydata)`. `xdata` and `ydata` should be vectors of **the same size**. For example, if we want to plot the function $f(x) = x^2$ in the interval $(-3,3)$, it could be done in the following way:

```
x=-3:0.01:3;  
fx=x.^2;  
plot(x,fx)
```

obtaining the following plot:

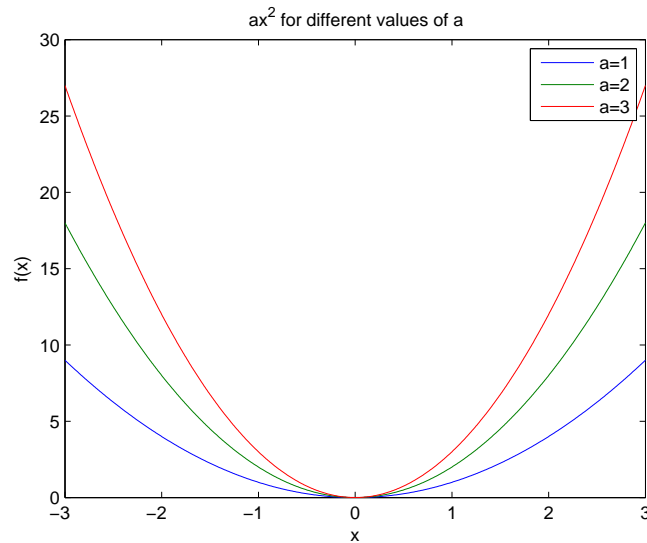


Note two things in the code: first, we created `x` using what is called a range. Its syntax is `xmin:increment:xmax` where `xmin` and `xmax` are the minimum and maximum values of your range, and `increment` is the value that is summed to the previous one, so for example above it would yield $x = (-3, -2.99, -2.98, \dots, 2.98, 2.99, 3)$. Second, we used the element-wise power which was mentioned earlier.

But plotting just one thing is dull, right? Furthermore, to present a nice figure we should have a title and labels on the axis, and maybe a legend.

Let's plot now the function $f(x) = ax^2$, with the parameter a being 1, 2 and 3.

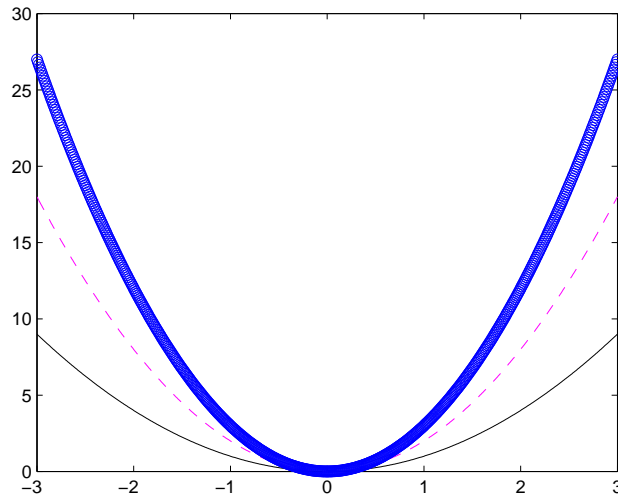
```
x=-3:0.01:3;
plot(x,x.^2,x,2*x.^2,x,3*x.^2)
xlabel('x')
ylabel('f(x)')
title('ax^2 for different values of a')
legend('a=1','a=2','a=3')
```



Now it looks nicer, right? Note that it's not necessary to define what to plot in a variable, although it is encouraged since it makes the code easier to read. The instructions to do the labels and the title are self-explanatory. The legend must be introduced in the same order as in the plot command.

Perhaps we don't like the colors or we want to make one of them in a dashed line or in a scatter plot. It can be done by exchanging the plot line above for:

```
plot(x,x.^2,'k',x,2*x.^2,'m--',x,3*x.^2,'o')
```



Notice the new arguments between apostrophes. The `k` and `m` will put the lines black and magenta, respectively. The `--` part will put a dashed line and the `o` will make the plot to be a scattered plot. More of these options are available in the MATLAB help (type `help plot` to access it). Furthermore, you can go to **Edit**→**Figure Properties** to edit these properties and many more in the graphical interface instead of using commands.

Probably you've noticed that every time you use a plot command, it will delete the existing plot. There are two ways to circumvent this: one is to create a new figure before the plot command with the command `figure` or `figure(i)`, where `i` is the figure number you wish. The other way is to use `hold on` after your first plot command. That will make MATLAB paint **all** the following plot commands in the same figure. You can deactivate it with `hold off`.

Finally, once your plot is good enough, you might want to export it. To do so, go to **File**→**Export Setup** and click **Export...** . Then you have to choose a place and a format for your file. For the format, I recommend using either PNG or EPS for your images.

3.3 Logic: if-else

An important part of any programming language is the ability to execute different code chunks for different conditions. In MATLAB (and in general, in most languages) this is handled by the `if`, `else` and `elseif` clauses.

The syntax for the three of them are quite similar:

```
if expression
    commands
elseif expression
    commands
else
    commands
end
```

Note the **end** at the last line here. It must be put at the end of a conditional, so if we only have an **if** clause, it has to be after the commands that were put below the **if**. The expressions that can be used are usually boolean, that is, true-false conditions. The most basic ones would deal with equalities and inequalities. For example:

	a	Result
a==5	5	true
a>4	5	true
a<5	5	false
a<=5	5	true
a~=5	5	false

Most of them are self-explanatory. Note the double equality in the first one, this is to distinguish it from the non-equality ($a \neq 5$) of the last one.

One can also concatenate different conditions in one single expression, to do that the symbols **&&** and **||** can be used between different conditions. The former means AND and the latter means OR.

Note also that when you make an if clause with an else and/or elseif, if the condition under the if is fulfilled, it will go through the if but not the other two, even if the condition for the elseif is satisfied.

Example

Let's assume that we want to check if an integer **a** is even or odd, and we want to display the result. To do so, one can proceed as follows:

```
if floor(a)~=a
    disp('a is not an integer')
elseif mod(a,2)>0
    disp('a is an odd integer')
else
    disp('a is an even integer')
end
```

The first conditional will check whether the integer is really an integer or if it has a decimal part. To do so, we compare it with its rounded down number by using the command

floor. If both are equal, then it is an integer, otherwise it is not, exiting the conditional at that very moment.

If it is an integer, it goes to the second conditional, which will see if it's odd. This can be easily done with the `mod(x,y)` command, which yields the rest (or modulus) of the division x/y , so for example `mod(5,2)` yields 1. If the rest is a non-zero number (which would be 1 in this case) then the number must be odd. If the rest is zero, it must be an even integer, which is printed with the final conditional.

3.4 Loops: for-while

Many times we want to automatize a large number of repetitive tasks. For example we might want to treat the values of a vector and then move them to another one. To do so, one can use a **for** loop:

```
for i = imin:increment:imax
    commands
end
```

The variable `i` will be the one which coordinates the loop and it's given as a range (see **Plotting**). The loop will execute as long as there are remaining values of `i` in the range, so for example a `for i = 1:30` will execute the commands inside the loop 30 times. Note that the variable `i` will change its value everytime it goes through the commands, so for example if we put `A(i)=i` inside the loop, it will assign the value 1 to the first entry of `A`, 2 to the second and so on. This could be done with the following code:

```
for i = 1:30
    A(i)=i;
end
```

Some useful commands that go very well with a **for** loop are the ones related to the size of a vector or matrix, which are `length(A)` and `size(A)`. The former will yield the value of the length of the **first** dimension of `A`. In vectors it would be the number of entries, but in matrices it will be the number of rows. The latter will yield the values of **all** the dimensions of `A` as a vector. So if you have a 20×10 matrix, `size` will yield (20,10) while `length` will only yield 20.

For example, let's assume that we have a vector `A` whose length is not known beforehand and we want to compute the sine of each element and store it in another vector `B`. One way of doing it by using a **for** loop would be:

```
for i = 1:length(A)
    B(i)=sin(A(i));
end
```

But at times we don't know how many times we have to perform the loop. For example, if we implement a numerical method whose finishing condition is a certain threshold, we have no way of telling how many times it will have to perform it. To solve this issue we have the `while` loop:

```
while expression
    commands
end
```

where the expression to be used is a logical expression like the in the conditionals mentioned above. For example, let's say that we want to compute the sum of $\frac{1}{n}$ where n is an integer until the terms are less than 10^{-6} . It could be done by:

```
thres=inf;
sum=0;
n=1;
while thres>1e-6
    thres=1/n;
    sum=sum+thres;
    n=n+1;
end
```

Note a couple of things about the above code: first, some of the values are initialized **before** the loop since otherwise the program will not work. You can even assign $\pm\infty$ to variables if you want. Then, unlike in a `for` loop, if you want to change a counter (in this case, it would be increasing n) you have to do it inside the loop.

Be aware that the commands inside the while will be executed as long as the conditional expression is true. It can happen that the expression is always true, for example the following code:

```
a=1;
while a==1
    disp('virus')
end
```

will display the word virus in the prompt for all eternity. This is known as an infinite loop and must be interrupted. **To interrupt the execution of any code, press Ctrl+C while in the MATLAB prompt**

4 Control applications

Now, we will focus on how to use MATLAB for control engineering. Those are basic functions that will be used throughout the course, but there are many more.

4.1 Defining a system

To define a system there are two possibilities: using a transfer function or using state-space.

- **Transfer function**, which usually has the following shape:

$$G(s) = \frac{A(s)}{B(s)}$$

where $A(s)$ and $B(s)$ are polynomials. The syntax to define a system like above would be using `tf(a,b)` where `a` and `b` are vectors which have the numerator and denominator coefficients of the polynomials starting with the highest degree one, so for example if we want to define a system for $\frac{s+3}{s^2-5s+3}$:

```
a=[1 1];
b=[1 5 3];
sys=tf(a,b);
sys =
      s + 1
-----
s^2 + 5 s + 3
Continuous-time transfer function
```

The variable we called `sys` is a special kind of variable which is only used for transfer functions.

- **State-space model**, which has to be specified with the 4 usual matrices A, B, C, D . To do so, we use the command `ss(A,B,C,D)`. Like when we define a transfer function, a state-space model will also be a special kind of variable.

Converting between transfer functions and state-space is quite easy: you have to use the functions `[A,B,C,D]=ssdata(sys_tf)` to obtain the matrices (which can be transformed into a state-space model with `ss`) and `sys_tf=tf(sys_ss)` to convert the state-space model to a transfer function.

Another issue is that if you use the commands above, it'll always yield a continuous-time system. To define a discrete-time system, you have to give an extra argument which would be the sampling time and MATLAB will do it for you, so for example, with `a` and `b` as defined above:

```
sys_d=tf(a,b,1)
sys_d =
      z + 1
-----
z^2 + 5 z + 3
Sample time: 1 seconds
Discrete-time transfer function
```

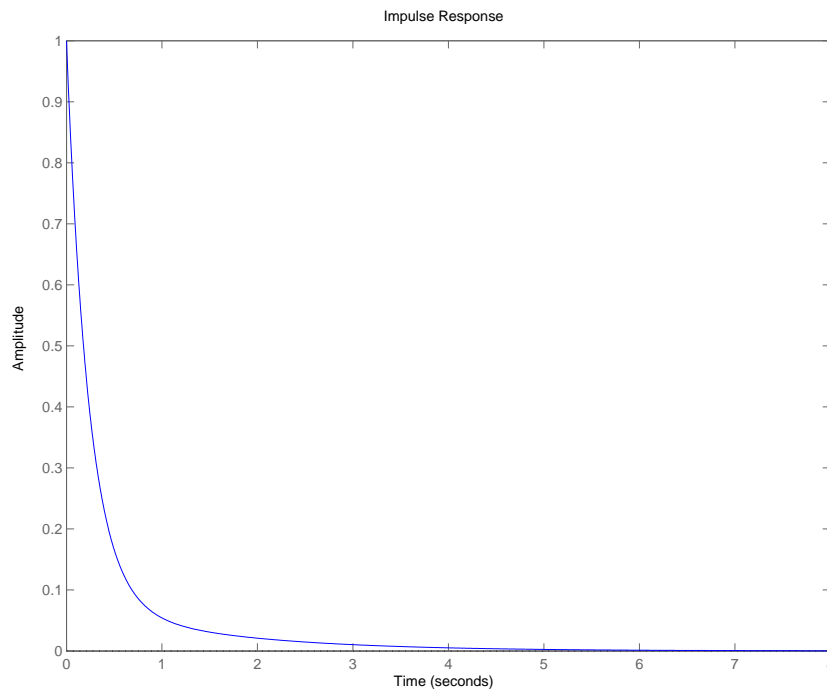
and the same with `ss(A,B,C,D,Ts)`.

To discretize a system which is given in continuous time, you have to use `c2d(sys,Ts)`, where `sys` is your system (either an `ss` or a `tf`) and `Ts` is your sampling time.

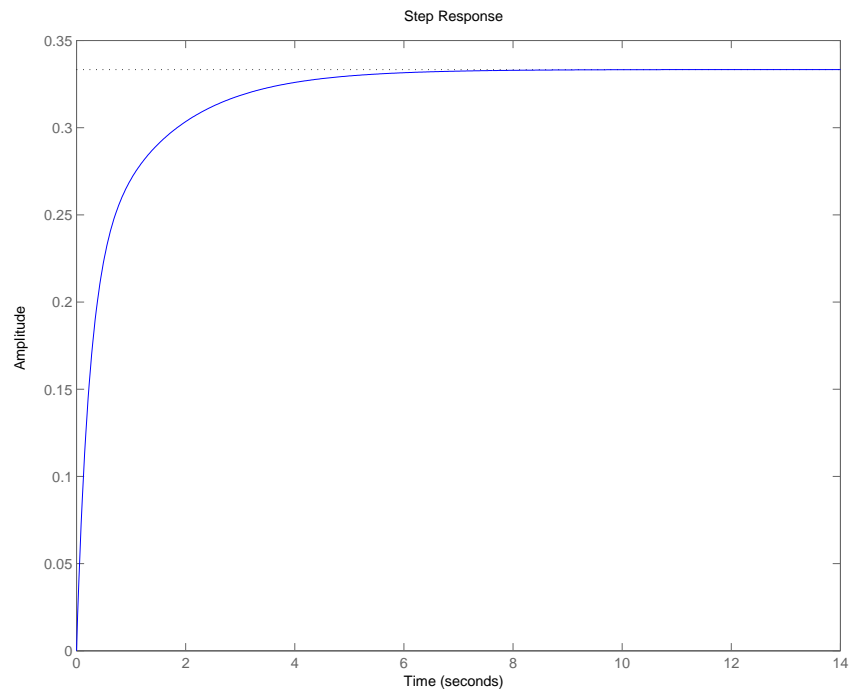
4.2 Responses to different signals

A good thing about defining the systems in the ways above is that there are some routines defined to make things easier for an engineer. One of such things is to be able to simulate the response to different signals:

- **Impulse response:** This is done by using the `impulse(sys)` command. It will plot the response as well. You can also obtain the time steps using for the simulation by using `[y,t]=impulse(sys)` where `y` would be the response and `t` would be the time steps. For the example above in continuous time, it would yield:



- **Step response:** This is done by using the `step(sys)` command. It works exactly the same as the impulse response. For example, for the above system it would yield:

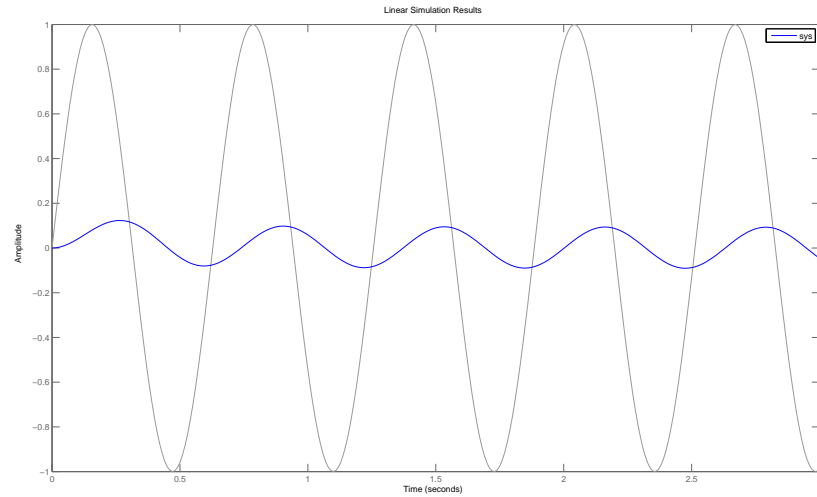


Note that in this case, the static gain is not equal to one!

- **Other responses:** This is done by using the `lsim(sys,u,t)`, where `u` would be the input (such as a parabola or a sine) and `t` would be a vector containing the time. For example for the system defined above:

```
t=0:0.01:3;  
u=sin(10*t);  
lsim(sys,u,t)
```

yields:



where here the blue line is the response of the system.

Last remarks

The MATLAB language is very big. After having used it for many years I still learn new ways of doing things that make my life easier every day. Furthermore, there are many user-made toolboxes (collections of user-made functions that are not shipped with MATLAB) which greatly expand its functionality.

I hope you enjoy the labs and that this guide was helpful to you.