Systems and Circuits
Lab 1: Signals

1. Objectives

In this first laboratory session, you will learn about

- Basic aspects of Matlab as a high-level programming language
- Represent discrete-time signals in Matlab
- Perform simple operations with discrete-time signals
- Compute the partial mean, power and energy of discrete-time signals

2. Evaluation

- Students must complete the exercises proposed during the lab session.

3. Working with discrete-time signals in Matlab

Matlab is a powerful mathematical software to perform matrix computations. It is a high-level programming language for numerical computation, visualization, and application development. Matlab provides the user a large amount of toolboxes to address different problems such as mathematical functions for linear algebra, statistics, Fourier analysis, filtering, optimization, numerical integration, and solving ordinary differential equations.
Our first goal is to represent discrete-time signals by means of vectors so they can be processed by Matlab. A vector in Matlab is a matrix of dimensions $N \times 1$ or $1 \times N$. In principle, a discrete-time signal will be represent by a vector of infinite size where each element contains the value of the signal at each time point. However, for obvious reasons, Matlab only works with **finite-length discrete-time signals**. Despite the fact that any computer can only represent real numbers up to a given numeric precision, we will consider that the discrete-time signals that we can represent in Matlab are can take any value. For instance, a computer cannot store an infinite number of decimals for the irrational number π, but representing this number with a sufficiently large number of decimals is a valid approximation to us.
Throughout the lab session, we will work with finite-length discrete-time signals: they only take non-zero values in a certain set of time points and these are the values that we store. For any other time point, the signal is equal to zero and this information does not have to be stored. To represent these kind of signals we need two vectors **of the same length**. For instance, assume the signal $x[n]$ is equal to $3n$ for $n = -1, \ldots, 5$ and zero otherwise. First, we need a vector that contains the time points of interest. In this case the vector is $[-1\ 0\ 1\ 2\ 3\ 4\ 5]$. The second vector contains the value of the signal at these time points, i.e., $[-3\ 0\ 3\ 6\ 9\ 12\ 15]$.
The command that we can use to represent a discrete-time signal is `stem`. The following code represents the following signal:

$$x[n] = \begin{cases} \sin(n) & -49 \leq n \leq 50 \\ 0 & \text{resto} \end{cases}$$

```
>> n = -49:50; % Vector of time-points
>> x = sin(n); % Vector with the values of the signal
>> stem(n,x); % Represent
>> xlabel('n'); % Horizontal Axis
>> ylabel('x[n]=sin[n]');  % Vertical Axis
```

If you need help about the usage of any Matlab command, for instance `stem`, type the following:

```
>> help stem
```

4. Transforming the independent variable

4.1. Temporal Inversion

A temporal inversion over the signal $x[n]$ means to construct the signal $y[n] = x[-n]$. The following Matlab code performs this operation and represents the obtained signal:

```
>> % Input signal x[n]
>> x = [0:0.1:1 ones(1,5)]; % Vector with the values of the signal
>> nx = 1:16; % Vector of time points.
>> % We represent the signal
>> figure
>> subplot(2,1,1)
>> stem(nx,x);
>> axis([-25 25 -0.2 1.2]); % We set an adequate zoom in both axis
>> xlabel('n'); % Horizontal axis
>> ylabel('x[n]');  % Vertical axis
>> % Temporal inversion
>> y = x(end:-1:1); % Inversion of the vector of signal values
>> ny = -nx(end:-1:1); % We have to perform the temporal inversion
>> %over the vector of time points too!
>> % We represent the signal y[n]
>> subplot(2,1,2)
>> stem(ny,y);
>>  axis([-25 25 -0.2 1.2]);
>> xlabel('n');
>> ylabel('y[n]=x[-n]');
```

where

- `0:0.1:1` is a command that returns a vector of real numbers between 0 and 1 with 0.1 steps.
- `ones(1,5)` returns a vector of five ones.

4.2. Time-scale operations (I)

Given a signal $x[n]$, we are now interested in constructing the signal $y[n] = x[3n]$. Recall this operation is not reversible, $y[n]$ is a compressed version of $x[n]$ and we have discarded samples of $x[n]$. To implement this operation in Matlab, we can use the following code:

```
>> % x[n]
>> x1 = [0:0.1:1 ones(1,5)];
>> x = [x1 x1 x1] % This command replicates the vector x1 three times
>> % Vector of time points
>> nx = 1:48;
>> % We represent the signal
>> figure
>> subplot(2,1,1)
>> stem(nx,x);
>> axis([-1 50 -0.2 1.2]);
>> xlabel('n'); % Horizontal axis
>> ylabel('x[n]');  % Vertical axis
>> % We compute the length of y[n]. Floor is a command
>> % to compute the lowest integer of the argument
>> Ny = floor(length(x)/3);
>> for k=1:1:Ny %For loop.  y(k) = x(k*3) is repeated for k=1, 2, ..., Ny
>>    y(k) = x(k*3);
>> end
>> % Vector of time points for y[n]
>> ny = 1:Ny
>>  subplot(2,1,2)
```

```
>> stem(ny,y);
>> axis([-1 50 -0.2 1.2]);
>> xlabel('n');
>> ylabel('y[n]=x[3n]');
```

4.3. Time scale operations (II)

Given $x[n]$, the operation $y[n] = x[n/2]$ is a reversible transformation since we do not discard any samples of $y[n]$. For those time points n such that $n/2$ is not an integer, we simply assume $y[n] = 0$. For any even n, then $y[n] = x[n/2]$. Therefore, $y[n]$ has double length that $x[n]$. The following code performs this operation

```
>> x1 = [0:0.1:1 ones(1,5)];
>> x = [x1 x1]
>> nx = 1:32;
>> figure
>> subplot(2,1,1)
>> stem(nx,x);
>> axis([-1 66 -0.2 1.2]);
>> xlabel('n');
>> ylabel('x[n]');
>> % Length of the signal y[n]
>> Ny = length(x)*2;
>> % We include the zeros
>> for k=1:1:Ny
>>    if rem(k,2) == 0 %If remainder of k/2 is equal to zero
>>       y(k) = x(k/2);
>>    else
>>       y(k) = 0;
>>    end
>> end
>> % Vector of time points for y[n]
>> ny = 1:Ny
>>  subplot(2,1,2)
>> stem(ny,y);
>> axis([-1 66 -0.2 1.2]);
>> xlabel('n');
>> ylabel('y[n]=x[n/2]');
```

4.4. Temporal shift

This operation is simple to perform since only affects to the vector of time points. For instance, to compute the signal $y[n] = x[n + 10]$, we only have to add 10 to each time point.

```
>> x = [0:0.1:1 ones(1,5)];
>> nx = 1:16;
>> figure
>> subplot(2,1,1)
>> stem(nx,x);
>> axis([-1 30 -0.2 1.2]);
>> % We shift the signal 10 time units to the right
>> ny = nx + 10;
>> y = x; % The values of y[n] are the same!
>>  subplot(2,1,2)
>> stem(ny,y);
>> axis([-1 30 -0.2 1.2]);
```

5. Computing energy and Power

5.1. Energy

The following Matlab code computes the energy of a signal $x[n]$ with values stored in the vector `x`.

```
>> x = [0:0.1:1 ones(1,5)];
>> nx = 1:16;
>> figure
>> stem(nx,x);
>> axis([-1 20 -0.2 1.2]);
>> Energia = sum(abs(x).^2)
```

5.2. Power

In general, for finite-length discrete-time signals we are not interested in computing the total average power, since we know it must be equal zero. For finite-length signals we can define the average power in the range of time-points of interest, which can be easily computed in Matlab as follows:

```
>> Power = mean(abs(x).^2)
```

In the case of periodic signals, we can compute the total average power since it suffices to store in Matlab a finite-length signal equal to an integer number of periods of the signal. For instance, define the signal $x[n] = \cos(2\pi n/13)$, which is periodic with fundamental period $N = 13$. The following code computes the power of $x[n]$ in one and four periods:

```
>> nx = 0:1:48;
>> x = cos(nx * 2 * pi/13);
>> figure
>> subplot(2,1,1)
>> stem(nx(1:13), x(1:13)); % We  represent 13 samples (one period)
>> axis([-1 50 -1.2 1.2]);
>> xlabel('n');
>> ylabel('1 periodo');
>> subplot(2,1,2)
>> stem(nx(1:end), x(1:end)); % We represent four periods of x
>> axis([-1 50 -1.2 1.2]);
>> xlabel('n');
>> ylabel('4 periodos');
>> Power_one_period = mean(abs(x(1:13)).^2)
>> Power_three_periods = mean(abs(x(8:43)).^2)
```

6. Exercises using audio signals

Using the above examples as reference, you have to implement in Matlab some simple operations using audio signals. The required audio files (`.wav`) can be found in Aula Global. The audio files correspond to either recorded voice or music.

6.1. Project 1: mixing music with *fading*

The goal of this project is to reproduce two different music pieces alternatively along the time using smooth transitions. We will use the main theme of *The Sting*[1] and the main theme of *Star Wars*.

1. **Load the two audio files.** Using the function `wavread.m` we store in a vector the values of the audio signal sampled at equally-spaced time points.

   ```
   >> TheSting = audioread('thesting.wav');
   ```

[1] `http://www.imdb.com/title/tt0070735/`

Proceed similarly to load the *Star Wars* clip in a vector called `StarWars`.

2. **Align the two signals** With the command `length` you can compute the length (number of samples) of each of the two signals. Since we are interested in working with signals of equal length, you have to cut the longest signal. Hint: the command `x=x(1:L)` cuts the vector `x` to its L first samples.

   Create a vector of time points `n` such that `n(1)=0` and the last component equal to the length of the two audio clips minus 1.

3. **Design the fading for the first signal.** We will create the fading effect (smooth transition between the two audio signals) by multiplying each signal by a fading sinusoidal signal of the form $0.5+0.5*cos(2\pi n/M)$. This signal oscillates between 0 and 1, when it takes values close to 1 the audio signal is almost not attenuated and it can be listened. When sinusoidal signal takes values close to zero, the audio signal will be severely attenuated and it can be hardly listened. At this point, we will reproduce the other audio clip with almost no attenuation.

   To design the fading signal, we have to first determine the number of transitions we wish between the two audio files. The length of the fading signal must be equal to the length of the audio signals and the period of the sinusoidal signal determines the number of transitions between the two clips: there will be as many transitions as periods of the fading signals in the length of the audio clip. First, determine M such that there are four transitions.

   With this value M, construct the signal $s1[n] = 0.5 + 0.5 * \cos(2\pi n/M)$.

4. **Attenuate the signal that is going to be played in the first place**. For instance,

   ```
   >> x1 = StarWars .* s1;
   ```

   Where the operator `.*` is the element-wise product.

5. **Design the fading for the second signal.** The fading signal that multiplies the second audio clip is complementary with the first fading signal. When the first fading signal is close to zero, the second one is close to one. This can be easily achieved by introducing a phase of π radians to the first sinusoidal fading signal $s2[n] = 0.5 + 0.5 * cos(2\pi n/M + \pi)$.

   Represent both $s1[n]$ and $s2[n]$ in the same plot to verify that they are complementary fading signals. Hint: to represent two signals in the same plot, use the commands `hold on` and `hold off`. Any plot command between these two lines will use the same figure for representation.

6. **Attenuate the signal that is going to be played in the second place**. For instance,

   ```
   >> x2 = TheSting.* s2;
   ```

7. **Construct the combination of the two audio clips**. We sum the two audio signals

   ```
   >> x = x1 + x2;
   >> playaudio(x);
   ```

   and we use the function `playaudio.m` to play the audio clip.

6.2. Project 2: Add background music

In this second project we will add background music to a voice audio clip. For instance, we will reproduce the music of the film *The Third Man* as background to the final monologue of the Ridley Scott film *Blade Runner*. We design the solution so that, in the beginning we hear the music loud, and once the voice starts it is attenuated. After the voice finishes, the music level is raised again.

1. **Load the audio files.** Assume we create two vectors called `bladerunner` y `thirdman`.

2. **Setting the vector lengths**. In this case the music clip lasts less than the voice clip. Construct a new vector where we replicate three times the `thirdman` vector. Denote this new vector by `tm`.

3. **Create an attenuation signal for the music.** According to the length of the voice clip, we have to generate a vector to attenuate the music while the voice is active. This vector has length `tm` and all its positions are equal to 1 (no attenuation) except one interval in the central part, where the value of these positions is 0.25. The length of this interval is equal to the number of samples of `bladerunner`.

4. **Aligning voice and music**. We have to create a new vector (of length `tm`). This vector, called `voice`, is equal to zero at all positions except those where `tm` is attenuated, where `voice` is equal to `bladerunner`.

5. **Create the audio clip** Finally, we create the desired audio clip by summing `voice` and the one obtained by multiplying point-wise `tm` with the attenuation signal. Use the function `playaudio.m` to play the audio clip.