**THE CITY COLLEGE OF NEW YORK**
**Department of Electrical Engineering**
**EE425 Computer Engineering Laboratory - Spring 2020**

---

**Exp. 5: Discrete-time Series Averaging Filters**

**Objective:** Implementation of moving average filters with application to discrete-time series.

Consider the periodic, discrete time series, **x[n]**, shown in Figure 1, where the abscissa corresponds to a discrete-time index, while the ordinate values (in decimal number representation) are given by the marker atop each stem. Our goal in this assignment will be to implement some elementary moving average filters using the periodic time series **x[n]** as input. The specific tasks below have been designed to guide you through the implementation of the moving average filters.

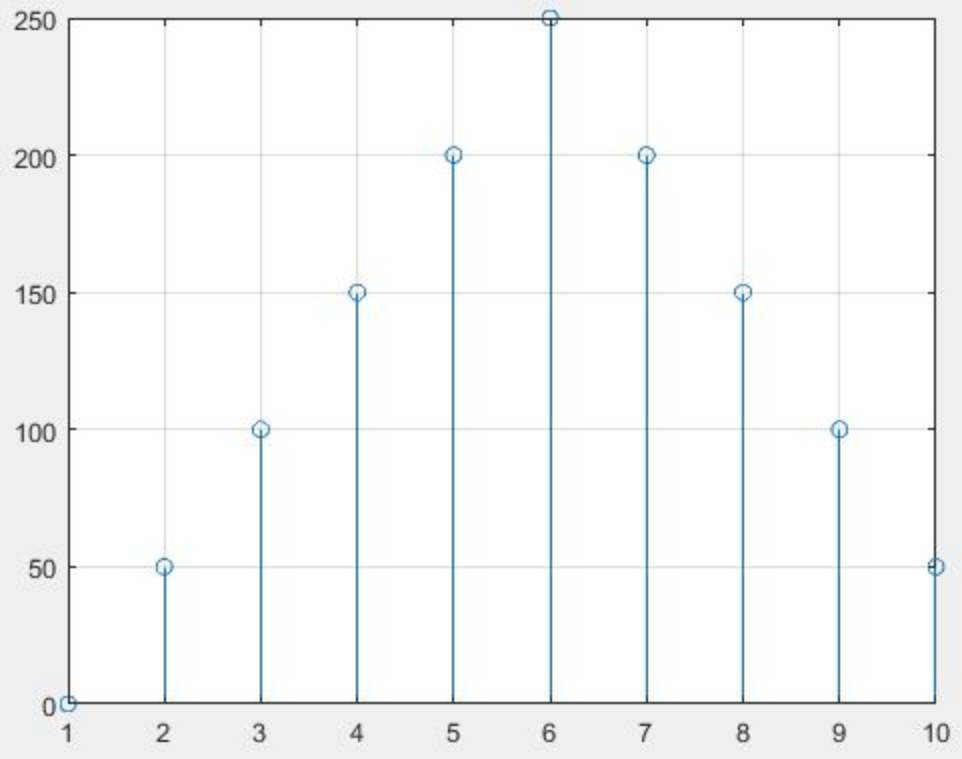


**Figure 1**. One period of the time series **x[n].**

## Task 1

1. The first thing for you to do is to compile and simulate the given *.asm* template called "**template_for_moving_average_by_AC.**" For now, the variable of interest in this template is the register called *value*. Run a simulation of this template and use a *Watch* window to monitor the content of this register. Note that, as the simulation runs through the code, the values of the *value* register correspond exactly to those of the time series **x[n]** shown in Figure 1. In short, **x[n]**= *value*, where the contents of the *value* register will evolve in time as the code is executed in the simulator. Make sure to set the *Animation Step Time* of the simulator to a setting that is appropriate for you to be able to see the evolution of the values of **x[n],** it should be evident to you that **x[n]** is indeed periodic..
2. Study the given template code and note that the contents of *value* are being retrieved, using the TABLAT pointer, from data stored originally in the PIC's program memory. This is something that I did not cover in class, but don't worry, you do not need to understand the PIC's TABLAT mechanism in order to complete this assignment. Thus, by *studying* the code, I mean that you should try to get a sense of what the entire code is doing (through simulations, of course) and make sure that you read all the comments that I wrote there so that you know where to start writing your code later on.
3. Do not proceed any further in this assignment until you have completed this Task. Please take step 2 very seriously, and email me if something isn't clear.

Note that Task 1 does not have anything to do with filters, *yet*.

## Task 2

The idea behind a moving average filter may be easily understood via an example. Consider again the time series **x[n]** from before, which is now given in Table 1 below, where **n** is the discrete-time index. Note that the ellipses (...) in the table mean time continuation, and *not* missing data.

| **n** | **...** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | **...** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **x[n]** | **...** | 50 | 0 | 50 | 100 | 150 | 200 | 250 | 200 | 150 | 100 | 50 | 0 | **...** |

**Table 1**. Periodic discrete-time series **x[n].**

We want to convolve this time series with the trivial moving average filter given by the following difference equation.

$$y[n] = \frac{1}{2}x[n] + \frac{1}{2}x[n-1]$$

Note how this simple equation just says that the output, **y[n]**, is simply calculated, for each **n**, as the average of two values of **x[n]** at consecutive time indices, namely, **n** and **n-1**. In other words, you take

**x[n]** (the current data value), add it to **x[n-1]** (the previous data value), and finally divide the resulting sum by two. All of this is shown in Table 2 below.

| **n** | -1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | **...** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **x[n]** | 100 | 50 | 0 | 50 | 100 | 150 | 200 | 250 | 200 | 150 | 100 | 50 | 0 | **...** |
| **y[n]** | **...** | 75 | 25 | 25 | 75 | 125 | 175 | 225 | 225 | 175 | 125 | 75 | 25 | **...** |

**Table 2**. Periodic time series **x[n]** and **y[n].**

Let us illustrate the idea above with another simple example corresponding to the following difference equation.

$$z[n] = \frac{1}{2}x[n] + \frac{1}{2}x[n-2]$$

Now try to convince yourself that the output **z[n]** corresponding to this new moving average filter should be as in Table 3.

| **n** | -1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | **...** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **x[n]** | 100 | 50 | 0 | 50 | 100 | 150 | 200 | 250 | 200 | 150 | 100 | 50 | 0 | **...** |
| **z[n]** | **...** | 100 | 50 | 50 | 50 | 100 | 150 | 200 | 200 | 200 | 150 | 100 | 50 | **...** |

**Table 3.** Periodic time series **x[n]** and **z[n].**

As an exercise, and to prepare you for what is to come later, please generate the output table for each of the following difference equations. You may need to extend the tables to allow for higher time indices in order to show the output correctly. Please include each table in your report. Each table should look similar to either Table 2 or 3.

$$a[n] = \frac{1}{2}x[n] + \frac{1}{2}x[n-3]$$

$$b[n] = \frac{1}{2}x[n] + \frac{1}{2}x[n-4]$$

$$c[n] = \frac{1}{2}x[n] + \frac{1}{2}x[n-5]$$

## Task 3

Now we finally arrive at the part of this assignment where you get to implement the moving average filters on the PIC using assembly language.

First, consider again the filter **y[n]** above, and note that in order to calculate the output we need to know **x[n]** at time indices **t=n** and **t=n-1,** and because of this we call **y[n]** a first-order filter. This should be clear from the equation for **y[n]**. Similarly, for the other filter, **z[n]**, in order to compute the output for **z[n]** we need to know **x[n]** at time indices **t=n** and **t=n-2**. Thus **z[n]** is a second-order filter. It is important that you understand what is really going on behind the scenes with the filter **z[n]**, so we discuss it in more detail in the following paragraph.

We have seen that in order to compute the output for **z[n],** at each time index, we need to know **x[n]** at time indices **t=n** and **t=n-2**. This means that the following sequential steps are executed as the time index increases..

1. To compute **z[0]**, we need **x[0]** and **x[-2]**.
2. To compute **z[1]**, we need **x[1]** and **x[-1]**.
3. To compute **z[2]**, we need **x[2]** and **x[0]**.
4. To compute **z[3]**, we need **x[3]** and **x[1]**.
5. ...
6. This algorithm simply continues ad infinitum.

Note how, in step 1 above, we needed **x[0]** and **x[-2]** to compute **z[0]**. This is clear from the equation for **z[n]**. But, in doing so, what happened with the data with time index **t = -1** which lies between **x[0]** and **x[-2]**, namely **x[-1]**? Did we discard it or throw it away? Well, we cannot simply forget about it or throw it away because, if you look at step 2 above, we will need **x[-1]** in order to compute **z[1]**. (Can you see that?) If we continue this process, as step 6 indicates, then you will notice that at every time index during which the filter **z[n]** is acting upon the data **x[n]**, we need to have saved, somewhere in our computer memory, the data corresponding to **three consecutive time indices**. That is, we need to have saved **x[n]** at indices **t=n, n-1**, and **n-2**. Note, however, that at each time index, the output **z[n]** only requires **two out of the three** consecutive values that we have saved in our memory. Let us call this allocated section in our computer memory, made up of only three data variables, a ***memory buffer***. As the time series evolves in time, our memory buffer must be updated with the (three) appropriate consecutive values so that our filter produces the correct average output at each time index. It may help you to revisit Tables 2 and 3 above with this new understanding that you have to create a memory buffer.

Now, consider the more general formula for an averaging filter.

$$d[n] = \tfrac{1}{2}x[n] + \tfrac{1}{2}x[n-k]$$

Armed with our understanding of **z[n]**, we can see that in order to be able to implement this more general filter, **d[n],** properly, we need to create a memory buffer which contains **k** variables. Now, there are two

different memory buffers that we can use: a linear buffer and a circular buffer. Since we will be concerned with moving average filters of low order, it will be sufficient to focus on implementing a linear buffer. In order to know what the differences between these two buffers are, please take a look at the following website:
https://www.allaboutcircuits.com/technical-articles/circular-buffer-a-critical-element-of-digital-signal-processors/
You should pay particularly close attention to the discussion surrounding Figure 3 from that reference.

Your task now is to read up on the linear buffer details in the reference provided above and, finally, to implement the filter **y[n]** shown above. Below are some guidelines to help you do this. Also. please read the comments in the *.asm* provided to you in order to see where the code for this assignment should be written. The two main parts that you have to figure out for this are as follows:

1. Implementation of the linear buffer.
    a. This should be done in a separate subroutine of its own.
    b. You will need to create several variables for you to save data into. Hint: for **y[n]** your buffer needs to be two variables long.
    c. The discrete-time series data, **x[n]**, to be saved in the buffer will be retrieved from the *value* register discussed in step 1 of Task 1 above. In other words, the input to your filter, y[n], will be the data **x[n]**=*value*, where the contents of the *value* register will evolve in time as the code is executed in the simulator.
2. Arithmetical computations.
    a. Note that the data from the *value* register is eight bits long.
    b. **y[n]** requires the sum of two variables and then a division by two. In this setting of fixed-point arithmetic, it can be shown that these two operations are not commutative. Therefore, in order to achieve the full dynamic range of the filter, you need to add the data values first, and then divide the sum by two at the end.
    c. Now, the division by 2 does not have to be a direct "division" using a "division" instruction. Instead consider the following: note that when you add two 8-bit numbers, in some cases a carry bit will result. This carry bit is automatically saved in the C bit inside the STATUS register. See datasheet, book, or lecture 0 notes for details. An understanding of the carry bit inside the STATUS register will be crucial in this part. Once you have the carry bit, then you should recall what happens when you “shift” the contents of a register towards the right by one bit. Hint: see the instruction *rrcf*
3. **You must simulate your code and make sure that it produces the data corresponding to y[n] shown in Table 2 above.**
4. The following instructions from the instruction set may be useful when implementing this moving average filter.
    a. *movff*
    b. *movf*
    c. *addwf*
    d. *rrcf*
    e. *andlw*

## Task 4

Having completed Task 3, please write the .asm code to implement the other moving average filters specified above, namely, **z[n], a[b], b[n]**, and **c[n]**.

1. Please generate separate *.asm* files for each filter. (Yes, I expect five (5) different .asm files.)
2. Note that once you figure out Task 3, then Task 4 becomes a simple extension of your work. If it doesn't seem to you like that is the case, then you should go back to Task 3 and make sure that you have a thorough understanding of that Task before you attempt this Task 4.