**ECE 1111: Engineering Computation I**

**Homework No. 7: File I/O**

**Goal:** Reading from and writing to files is an important part of programming. This allows your program to interact with external data, which is very important in engineering. In this homework assignment, you will learn how to read and write both formatted and unformatted data to a file. You will also learn how to interface data you generate to programs such as Audacity.

**Description:** There are two tasks in this homework assignment:

1. Generate a binary file that contains *64,000* samples of a *100* Hz sinewave. Write short integer values to the file. Use a sample frequency of *8,000* Hz and have the amplitudes range from [-*10,000*, *10,000*]. This means the file will contain *8* seconds of a signal consisting of a sinewave where each sample of the sinewave represents the value of the sinewave at an increment of *1/8000* secs. The calculation you want to implement is:

where *i* ranges from *0* to *64,000*. The file should have a size of *64,000* samples \* *2* bytes/sample = *128,000* bytes.

1. Load this file into an audio tool such as Audacity and listen to it to verify that it sounds correct (you will have to do some research on how to do this). Make sure you set the proper sample frequency and sample size so that the file sounds like a sinewave. If you get these settings wrong, the file will sound like very loud noise.
2. Write a program that reads this file in small chunks. The interface must be as follows:

**myprog.exe myfile.dat 2 6**

Argv[1] represents the filename; argv[2], which we will refer to as *N*, represents the number of samples in a frame; and argv[3], which we will refer to as *M*, represents the number of samples in a window. Your program should work for any combination of *N* and *M*. *N* and *M* are typically *80* and *240* respectively for audio data.

For this assignment, you can assume *M* > *N*. However, if you write your code correctly, that shouldn’t be an issue.

There are two ways to do this: (1) loop over the file, position the file pointer to the sample corresponding to the beginning of the window, and read *M* samples; (2) use a buffer shifting approach described below. every *N* samples. Read *M* samples into and array and print the samples contained in your array to stdout.

Let me demonstrate with an example. Suppose you have the following signal:

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21

Suppose *N = 2* and *M = 6*. Your first frame corresponds to the samples *[0,1]*. The ‘window’ corresponding to this frame will be the samples *[0,0,1,2,3,4]*, and these will be the samples you print to stdout. Zero-stuff the signal, which means you use a value of 0 for values at the beginning or end of the file (e.g., the first two samples are unknown and set to zero for the first frame). You will need to zero-stuff the signal at the “edges” of the file – the first few and last few frames depending on the values of *N* and *M*.

The second frame corresponds to *[3,4]*. The window of samples that corresponds to this frame are *[1,2,3,4,5,6]*.

The third frame corresponds to *[5,6]*. The window of samples that corresponds to this frame are *[3,4,5,6,7,8]*.

Note that in the above example when you hit the end of file you only have half a frame. Our general rule is if there is more than half a frame of data, you output the frame. Otherwise you truncate it (do not output it).

Your program should work for any combinations of *N* and *M*, and it should handle the end of file or beginning of file by assuming signal values of *0*.

To implement this, create a buffer *M* samples long. Using the above example, cycle the data through the buffer this way:

**Initialization: [0, 0, 0, 0, 0, 0]**

**Pre-fetch the first frame: [0, 0, 1, 2, 3, 4]**

**Shift the buffer by N samples: [1, 2, 3, 4, \*, \*]**

**Read the next N samples: [1, 2, 3, 4, 5, 6]**

**Shift the buffer by N samples: [3, 4, 5, 6, \*, \*]**

**Read the next N samples: [3, 4, 5, 6, 7, 8]**

... iterate until there is no more data to read and you have processed the last window ...

Shifts can be done using a for loop or using the memmove() function.

Submit your deliverables into the directory:

/**data/courses/ece\_1111/2019\_fall/homework/hw\_07/lastname\_firstname**

No write up is needed for this assignment. We will compile and run your code, and, of course, read your comments. Use Make files.

The technique of reading data from a file in this way is referred to as framing and windowing. In real code, the window can be center-aligned, left-aligned or right-aligned. In this example, the window is center-aligned. This type of I/O is quite common in signal processing and machine learning systems.