ECE 4755: Digital Signal Processing Lab

DSP Lab Final Project

Angel Rodriguez and Austin Philips

Utah Valley University

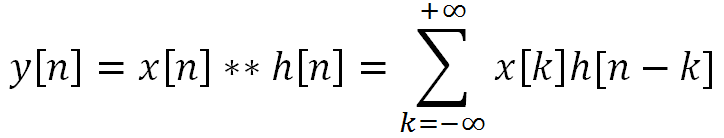
# Introduction

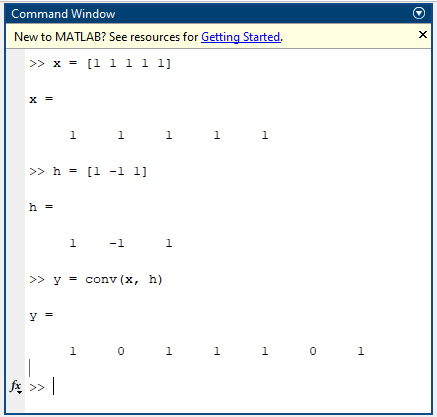
The objective of this lab is to learn how to use Code Composer Studio (CCS) to implement convolution in C. This project is focused on the basics of CCS and how to load the code from the computer into a TI DSP memory. Voice recordings of the input signal x[n] will be convolved with the impulse response y[n] to derive an output y[n] = x[n] \* h[n]. X[n] will include the recordings of our own voice within an area without an echo while h[n] shall be a recording of a clap within an area to create an echoing impulse response. The goal of the convolution in C is to import these two sounds files as arrays of whatever types necessary (i.e., uint16, int32), output y[n] as another array of the same type and convert that array back into the same sound file type x[n] and h[n] originated from. The expected resulting sound is the voice with an echo.

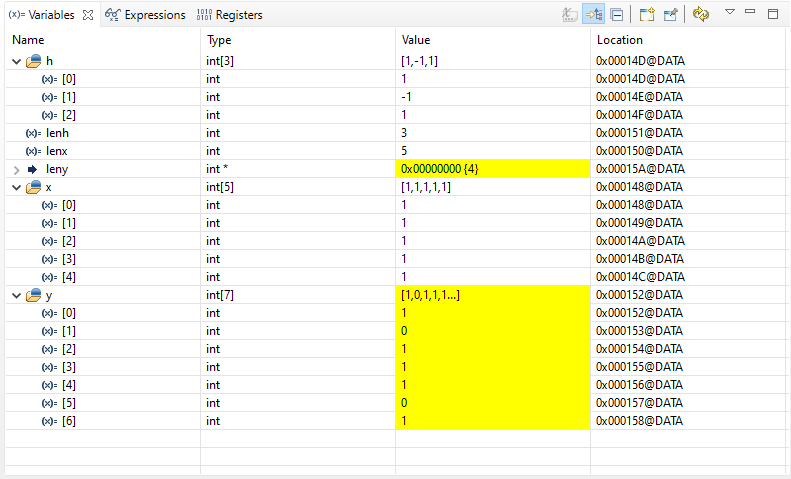
# Process

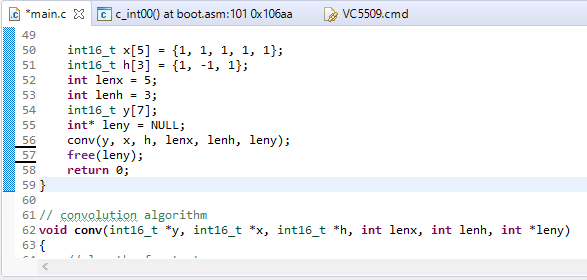
The first step of the process is to implement a convolution function in C and test out the function with two small arrays. To start off, the function will have a for loop to repeat as many times as the length of the longer array between x[n] and h[n] with each loop performing the multiplication operation x[k] \* h[n-k] shown in Figure 1. For testing the small arrays with the convolution function created, the correct output can be determined by using the “conv” function in MATALB. The convolution function is to be tested using x = [1, 1, 1, 1, 1], h = [1, -1, 1], and y = conv(x, h). In MATLAB, the code is run with the result of [1, 0, 1, 1, 1, 0, 1] (See Figure 2), so our convolution function should yield the same result to be considered adequately functioning. Within the CCS program, the same output is achieved with the same x[n] and h[n] used in MATLAB showing that the convolution function is working as expected (Figures 3, 4). The next step is to load the voice recordings which are MA4 files. To get them into the CCS program as arrays, the MA4 files were converted to large arrays of type uint8\_t (unsigned 8-byte integers) and declared “echo” (x[n]) and “impulse” (y[n]). An issue was run into when the program would not build due to not having enough memory to hold the very large arrays which came from just a few seconds of recording. The issue was resolved through using #pragma in C which dedicated memory to just an array (Figure 5). Once the program was able to compile and run, output was received which was converted to .WAV file and played. It did not sound as expected and it was hypothesized that the convolution function changes y[n] to a reversed version of what the correct y[n] should be. This would make sense as the MATLAB test we did has an output [1, 0, 1, 1, 0, 1] which is the same when it is reversed. A new MATLAB test was made which confirmed our hypothesis (Figures 6, 7). The C code in CCS was edited to include a new function in main which reverses the output signal y[n]. Following reversal of the output, the convolution function was this time confirmed to be working adequately and thus, was applied to the input signal x[n] and impulse response h[n]. Results are described and shown below.

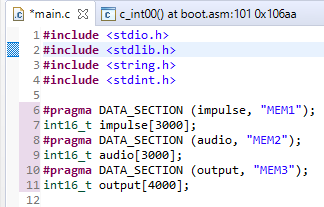
# Figures

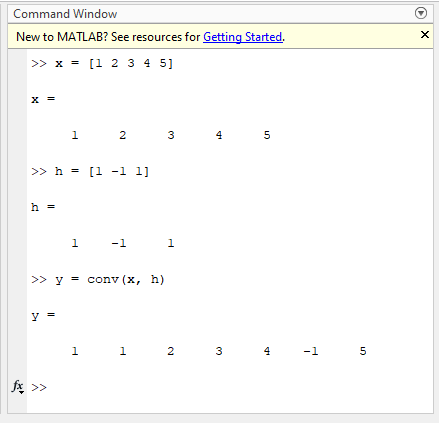
  
Figure 1:  
Convolution summation equation used for C program. The sigma is implemented by a for loop where negative infinity is 0 and positive infinity is the length of the longer array between x[n] and h[n].

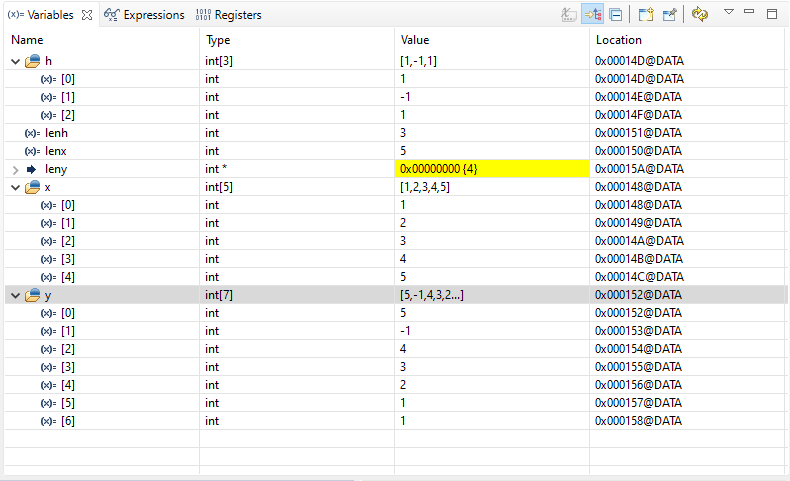
  
Figure 2:  
Command Window of MATLAB used to find the correct output of convolution using small testing arrays to also be used in CCS.

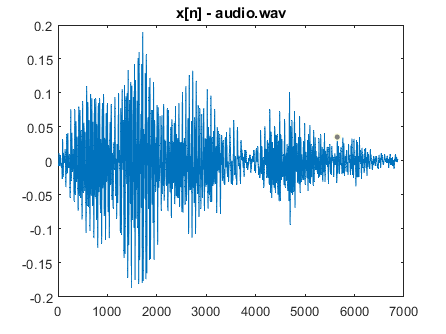
  
Figure 3:  
Expressions for arrays of x, h, and y depicting values which match the results of MATLAB showing the convolution function written in C works adequately.

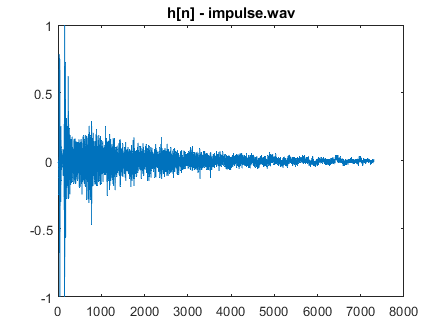
  
Figure 4:  
C code of CCS implementing convolution for small testing arrays.

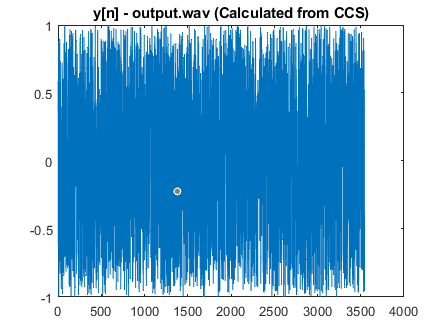
  
Figure 5:  
Dedicating specific amounts of RAM (”MEM1“, ”MEM2“, ”MEM3”) to the large arrays which are used for impulse array h[n], audio array x[n], and output array y[n].

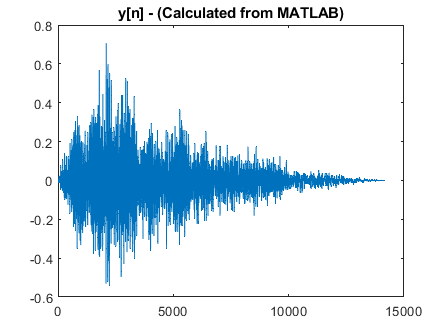
  
Figure 6:  
The second MATLAB test shows results of y[n] which is not equivalent to itself reversed.

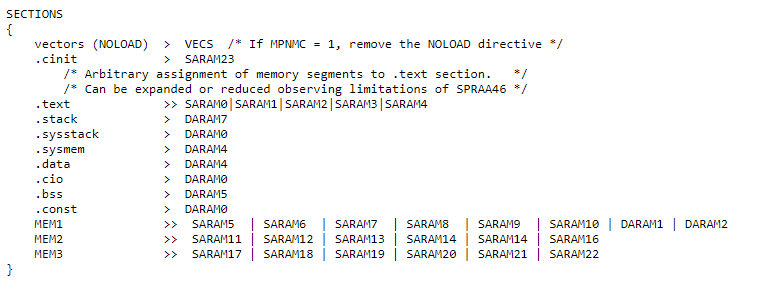
  
Figure 7:  
Expressions for arrays of x, h, and y depicting values which when compared to the second MATLAB test, show that y is reversed of what it should be and, therefore, the convolution function written in C does not work as previously thought.

  
Figure 8:  
MATLAB-generated plot of the input signal x[n] used for the convolution in CCS.

  
Figure 9:  
MATLAB-generated plot of the impulse response h[n] used for the convolution in CCS.

  
Figure 10:  
MATLAB-generated plot of the output signal y[n] which is the result of the convolution in CCS.

  
Figure 11:  
MATLAB-generated plot of the output signal y[n] which is the result of the conv function of MATLAB.

Figure 12:  
Edited command file to allocate RAM required for x[n], h[n], and y[n].

# Conclusion

The lab ends with learning the manuals regards to memory and working with DSPLab unit to implement convolution. Concepts learned include connecting dedicating RAM to arrays, converting .WAV files to int16\_t arrays in C, passing arrays to a function as a pointer, using Python to shorten the length of audio values to put into C and then convert back to audio files, and better understanding of arrays/pointers/memory allocation in CCS. A concept learned is that variables assigned to arrays are pointers which point to the 0th index of the array. That way, when the convolution function created has a parameter which is an int16\_t pointer, the array variable name can be used as the parameter to the function call since that array variable name points to a memory address holding an int16\_t value (the first index of the array). When demanding large amounts of memory for a program array, the .cmd file in CCS must be manually edited to allocate RAM necessary to hold the large array(s). This issue was the greatest and most time-consuming obstacle of the lab and required much assistance and reading from the DSP books. The command file was edited to create “MEM1”, “MEM2”, and “MEM3” to hold the memory required for the data files of the input signal, impulse response signal, and output signal (see Figure 12). Sound files cannot be imported simply to CCS as easily as in MATLAB, so the solution was to convert sound files (.WAV) to binary (using Python) to be used in CCS, and then when CCS gives an output, the output was converted back to .WAV form using a Python program for the sound to be played. Figures 8-11 depict the input signal .WAV file, impulse response .WAV file, output .WAV file produced by convolution in CCS, and the output signal calculated using the convolution function in MATLAB. It appears that the convolution function implemented in C does not work as intended based on the graphs. The input signal is a voice saying, “hello there” and the impulse response is the sounds of an echoing clap. The output signal produced by CCS (Figure 10) sounds noisy like television static, meanwhile, the output signal produced by MATLAB’s convolution sounds like the “hello there” voice with an echo which is what is expected. These results let us know that something with the reading and writing of wav files has gone wrong. The C code does not interpret the values the same way that Python/MATLAB does. Also, integer overflow within the convolution may be another cause of the imperfection.

# Attachments

## CCS Code of main.c: Implements convolution

#include "convolution.h"  
  
// convolution algorithm  
int16\_t conv(int32\_t \*y, int16\_t \*x, int16\_t \*h, int16\_t lenx, int16\_t lenh)  
{  
 // length of output  
 int16\_t conv\_length;  
 // iterator for x, h, and inverted x respectively  
 int16\_t i, j, hmin, hmax;  
  
 // allocated convolution array  
 conv\_length = lenx+lenh-1;  
  
 for(i = 0; i < conv\_length; i++) {  
 y[i] = 0;  
  
 hmin = (i >= lenh - 1) ? i - (lenh - 1) : 0;  
 hmax = (i < lenh - 1) ? i : lenx - 1;  
 for(j = hmin; j < hmax; j++) {  
 y[i] += x[j]\*h[i-j];  
 }  
 }  
  
 //get length of convolution array  
 return conv\_length;  
}

## CCS Code of convolution.h: Header file for convolution.c

## #ifndef CONV\_FUNC #define CONV\_FUNC #include <stdint.h> int16\_t conv(int32\_t \*y, int16\_t \*x, int16\_t \*h, int16\_t lenx, int16\_t lenh); #endif

## CCS Code of fileread\_main.c: Alternate main.c that incorrectly reads binary files

#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <stdint.h>  
  
#pragma DATA\_SECTION (impulse, "MEM1");  
int16\_t impulse[3000];  
#pragma DATA\_SECTION (audio, "MEM2");  
int16\_t audio[3000];  
#pragma DATA\_SECTION (output, "MEM3");  
int16\_t output[4000];  
  
void conv(int16\_t \*y, int16\_t \*x, int16\_t \*h, int lenx, int lenh, int \*leny);  
  
/\*\*  
 \* main.c  
 \*/  
int main(void)  
{  
  
 // create pointers to audio files  
 FILE \* impulse\_file;  
 FILE \* audio\_file;  
  
 // temporary values for the file reading  
 int16\_t tmp\_read;  
 int tmp\_pointer = 0;  
  
 // code to read the impulse file and store it into the impulse array  
 impulse\_file = fopen("impulse.bin", "r");  
 if(impulse\_file == NULL) {  
 perror("Error opening impulse file.");  
 return(-1);  
 }  
 // pointer, size in bytes of each element, number of elements, file stream  
 fread(impulse, 2, 1826, impulse\_file);  
 fclose(impulse\_file);  
  
 // code to read the audio file and store it into the audio array  
 tmp\_pointer = 0;  
 audio\_file = fopen("audio.bin", "r");  
 if(audio\_file == NULL) {  
 perror("Error opening audio file.");  
 return(-1);  
 }  
 // pointer, size in bytes of each element, number of elements, file stream  
 fread(audio, 2, 1718, audio\_file);  
 fclose(audio\_file);  
  
 int\* leny = NULL;  
  
  
  
 free(leny);  
  
 return 0;  
}  
  
// convolution algorithm  
void conv(int16\_t \*y, int16\_t \*x, int16\_t \*h, int lenx, int lenh, int \*leny)  
{  
 // length of output  
 int conv\_length;  
 // iterator for x, h, and inverted x respectively  
 int i, j, i1;  
 // temporary result of each output  
 int16\_t tmp;  
  
 // allocated convolution array  
 conv\_length = lenx+lenh-1;  
  
 // convolution process  
 // outside loop loops through output  
 for (i=0; i < conv\_length; i++)  
 {  
 // this would point to the end of x  
 i1 = lenx - i - 1;  
 // temporary storage value  
 tmp = 0;  
 // inner loop loops through h  
 for (j=0; j<lenh; j++)  
 {  
 // this if loop checks to make sure the parts (x) are valid  
 if(i1>=0 && i1<lenx)  
 // this is the summation of each multiplication for each x[-n]\*h[m]  
 tmp = tmp + (x[i1]\*h[j]);  
  
 // increment so that we pass through each value of x where an h exists  
 i1 = i1+1;  
 // assign our working value to the output signal  
 y[i] = tmp;  
 }  
 }  
  
 //get length of convolution array  
 (\*leny) = conv\_length;  
}

## CCS Code of main.c: Main C program file

#include <stdio.h>  
#include <stdlib.h>  
#include <stdint.h>  
  
#include "convolution.h"  
  
int32\_t y[9] = {0};  
int16\_t x[5] = {1, 2, 3, 4, 5};  
int16\_t h[5] = {1, 2, 3, 4, 5};  
  
int main(void) {  
  
 int16\_t leny = conv(y, x, h, 5, 5);  
  
 int k = 0;  
 for (k = 0; k < leny; k++) {  
 printf("%d\n", y[k]);  
 }  
  
 return 0;  
}

## CCS Code of VC5509.cmd: Command file edited for RAM allocation

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
/\* VC5509.cmd \*/  
/\* Copyright (c) 2012 Texas Instruments Incorporated \*/  
/\* Author: Rafael de Souza \*/  
/\* \*/  
/\* Description: This file is a sample linker command file that can be \*/  
/\* used for linking programs built with the C compiler and \*/  
/\* running the resulting .out file on a VC5509 or VC5509A. \*/  
/\* Use it as a guideline. You will want to \*/  
/\* change the memory layout to match your specific \*/  
/\* target system. You may want to change the allocation \*/  
/\* scheme according to the size of your program. \*/  
/\* \*/  
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
  
MEMORY  
{  
 MMR: o = 0x000000 l = 0x0000c0 /\* 192B Memory Mapped Registers \*/  
 DARAM0: o = 0x0000C0 l = 0x001F40 /\* 8kB Dual Access RAM 0 \*/  
 DARAM1: o = 0x002000 l = 0x002000 /\* 8kB Dual Access RAM 1 \*/  
 DARAM2: o = 0x004000 l = 0x002000 /\* 8kB Dual Access RAM 2 \*/  
 DARAM3: o = 0x006000 l = 0x002000 /\* 8kB Dual Access RAM 3 \*/   
 DARAM4: o = 0x008000 l = 0x002000 /\* 8kB Dual Access RAM 4 \*/  
 DARAM5: o = 0x00A000 l = 0x002000 /\* 8kB Dual Access RAM 5 \*/  
 DARAM6: o = 0x00C000 l = 0x002000 /\* 8kB Dual Access RAM 6 \*/  
 DARAM7: o = 0x00E000 l = 0x002000 /\* 8kB Dual Access RAM 7 \*/  
   
 SARAM0: o = 0x010000 l = 0x002000 /\* 8kB Single Access RAM 0 \*/  
 SARAM1: o = 0x012000 l = 0x002000 /\* 8kB Single Access RAM 1 \*/  
 SARAM2: o = 0x014000 l = 0x002000 /\* 8kB Single Access RAM 2 \*/  
 SARAM3: o = 0x016000 l = 0x002000 /\* 8kB Single Access RAM 3 \*/   
 SARAM4: o = 0x018000 l = 0x002000 /\* 8kB Single Access RAM 4 \*/  
 SARAM5: o = 0x01A000 l = 0x002000 /\* 8kB Single Access RAM 5 \*/  
 SARAM6: o = 0x01C000 l = 0x002000 /\* 8kB Single Access RAM 6 \*/  
 SARAM7: o = 0x01E000 l = 0x002000 /\* 8kB Single Access RAM 7 \*/  
 SARAM8: o = 0x020000 l = 0x002000 /\* 8kB Single Access RAM 8 \*/  
 SARAM9: o = 0x022000 l = 0x002000 /\* 8kB Single Access RAM 9 \*/  
 SARAM10: o = 0x024000 l = 0x002000 /\* 8kB Single Access RAM 10 \*/  
 SARAM11: o = 0x026000 l = 0x002000 /\* 8kB Single Access RAM 11 \*/   
 SARAM12: o = 0x028000 l = 0x002000 /\* 8kB Single Access RAM 12 \*/  
 SARAM13: o = 0x02A000 l = 0x002000 /\* 8kB Single Access RAM 13 \*/  
 SARAM14: o = 0x02C000 l = 0x002000 /\* 8kB Single Access RAM 14 \*/  
 SARAM15: o = 0x02E000 l = 0x002000 /\* 8kB Single Access RAM 15 \*/  
 SARAM16: o = 0x030000 l = 0x002000 /\* 8kB Single Access RAM 16 \*/  
 SARAM17: o = 0x032000 l = 0x002000 /\* 8kB Single Access RAM 17 \*/  
 SARAM18: o = 0x034000 l = 0x002000 /\* 8kB Single Access RAM 18 \*/  
 SARAM19: o = 0x036000 l = 0x002000 /\* 8kB Single Access RAM 19 \*/  
 SARAM20: o = 0x038000 l = 0x002000 /\* 8kB Single Access RAM 20 \*/  
 SARAM21: o = 0x03A000 l = 0x002000 /\* 8kB Single Access RAM 21 \*/  
 SARAM22: o = 0x03C000 l = 0x002000 /\* 8kB Single Access RAM 22 \*/  
 SARAM23: o = 0x03E000 l = 0x002000 /\* 8kB Single Access RAM 23 \*/  
   
 CE0: o = 0x040000 l = 0x3C0000 /\* 4MB CE0 external memory space \*/   
 CE1: o = 0x400000 l = 0x400000 /\* 4MB CE1 external memory space \*/  
 CE2: o = 0x800000 l = 0x400000 /\* 4MB CE2 external memory space \*/  
 CE3: o = 0xC00000 l = 0x3F0000 /\* 4MB CE3 external memory space \*/  
 ROM: o = 0xFF0000 l = 0x00FF00 /\* 64kB ROM (MPNMC=0) or CE3 (MPNMC=1) \*/  
 VECS: o = 0xFFFF00 l = 0x000100 /\* reset vector \*/  
}   
   
SECTIONS   
{   
 vectors (NOLOAD) > VECS /\* If MPNMC = 1, remove the NOLOAD directive \*/  
 .cinit > SARAM23  
 /\* Arbitrary assignment of memory segments to .text section. \*/  
 /\* Can be expanded or reduced observing limitations of SPRAA46 \*/   
 .text >> SARAM0|SARAM1|SARAM2|SARAM3|SARAM4   
 .stack > DARAM7  
 .sysstack > DARAM0  
 .sysmem > DARAM4  
 .data > DARAM4  
 .cio > DARAM0  
 .bss > DARAM5  
 .const > DARAM0  
 MEM1 >> SARAM5 | SARAM6 | SARAM7 | SARAM8 | SARAM9 | SARAM10 | DARAM1 | DARAM2  
 MEM2 >> SARAM11 | SARAM12 | SARAM13 | SARAM14 | SARAM14 | SARAM16  
 MEM3 >> SARAM17 | SARAM18 | SARAM19 | SARAM20 | SARAM21 | SARAM22  
}

Python Code of [modifydata.py:](https://github.com/austinphill6/ECE4755/blob/main/modifydata.py) Script to edit waveform  
from scipy.io import wavfile  
import numpy as np  
  
impulse = wavfile.read('/home/snekmaster/Documents/impulse\_response2.wav')  
  
impulse\_response = np.array([])  
size = int(impulse[1].shape[0]/4)  
  
with open('impulse\_response.c', 'w') as file:  
 file.write(f'uint8\_t impulse\_response [{size}] = ' + '{')  
 for i in impulse[1]:  
 if i % 4 == 0:  
 file.write(f'{i},\n')  
 impulse\_response = np.append(impulse\_response, i)  
 file.write('};')  
  
fs = int(impulse[0]/4)  
print(fs)  
print(impulse\_response.shape)  
wavfile.write("new\_impulse\_response.wav", fs, impulse\_response.astype(np.uint8))  
  
echo = wavfile.read('/home/snekmaster/Documents/echo.wav')  
  
echo2 = np.array([])  
size = int(echo[1].shape[0]/4)  
with open('echo.c', 'w') as file:  
 file.write(f'uint8\_t echo [{size}] = ' + '{')  
 for i in echo[1]:  
 if i % 4 == 0:  
 file.write(f'{i},\n')  
 echo2 = np.append(echo2, i)  
 file.write('};')  
  
fs = int(echo[0]/4)  
print(fs)  
print(echo2.shape)  
wavfile.write("new\_echo.wav", fs, echo2.astype(np.uint8))

Python Code of [wavtobinary.py](https://github.com/austinphill6/ECE4755/blob/main/wavtobinary.py)[:](https://github.com/austinphill6/ECE4755/blob/main/modifydata.py) Script to convert .WAV file to binary values  
import wave  
  
divider = 4  
  
counter = 0  
with wave.open('impulse.wav', 'r') as impulse:  
 with open('data\_files/impulse.bin', 'bw') as f:  
 while True:  
 frame = impulse.readframes(1)  
 if frame == b'':  
 break  
 elif counter % divider == 0:  
 f.write(frame)  
 counter = counter + 1  
  
counter = 0  
with wave.open('audio.wav', 'r') as audio:  
 with open('data\_files/audio.bin', 'bw') as f:  
 while True:  
 frame = audio.readframes(1)  
 if frame == b'':  
 break  
 elif counter % divider == 0:  
 f.write(frame)  
 counter = counter + 1

## Python Code of [binarytowav.py](https://github.com/austinphill6/ECE4755/blob/main/binarytowav.py)[:](https://github.com/austinphill6/ECE4755/blob/main/modifydata.py) Script to convert binary back to .WAV file

import wave  
  
divider = 4  
  
counter = 0  
with wave.open('output.wav', 'w') as output:  
 with open('data\_files/output.bin', 'br') as f:  
 output.setnchannels(1)  
 output.setsampwidth(2)  
 output.setframerate(2756)  
 output.setnframes(3543)  
 output.writeframes(f.read(3543\*2))  
 output.close()

## MATLAB Code of Project\_6.mlx[:](https://github.com/austinphill6/ECE4755/blob/main/modifydata.py) To visually display sound signals

clc, clear; close all;

x = audioread('audio.wav');  
plot(x); title('x[n] - audio.wav')  
soundsc(x);  
pause(3);

h = audioread('impulse.wav');  
plot(h); title('h[n] - impulse.wav')  
soundsc(h);  
pause(3);

y1 = audioread('output.wav');  
plot(y1); title('y[n] - output.wav (Calculated from CCS)')  
soundsc(y1);  
pause(3);

y2 = conv(x,h);  
plot(y2); title('y[n] - (Calculated from MATLAB)')  
soundsc(y2);  
pause(3);