Ayden Dauenhauer and Benjamin Banh

Prof. Wood

ELEN 133L

22 May 2025

Lab 7: Real Time Implementations using Circular Buffers

Step 1: 4th Order FIR Test

When your program compiles without errors, test it and make sure it produces the same result that you observed in Step 6 of Lab 6 when you listen to the scales audio.

We can hear that it acts as a bandstop filter which is the same as step 6 from last lab.

Step 2: 2nd Order IIR Test

When your program compiles without errors, test it and make sure it produces the same result that you observed in Step 5 of Lab 6 when you listen to the scales audio. If your output sounds like it comes from a high pass filter rather than a low pass filter, what would cause this problem to happen?

We can hear that this filter acts as a low pass filter. This is the same as step 5 from last lab.

Step 3: Butterworth Bandpass Filter

Sounds like only middle frequencies are preserved, cutting both lows and highs which is shown in the bandpass filter.

Step 4: Butterworth Bandstop Filter

Sound similar to Step 1, but smoother due to the IIR filter characteristics.

Step 5: Parks McClellan Bandpass Filters

Sounds “focused” on a specific pitch range. It is more precise than Butterworth due to FIR linear phase.

Step 6: Implement a Filter to Create an Echo

Audio takes previous values to now sound echo-y.

```
#include "main.h"
#include "coefficients.h"
#include <stdbool.h>



#define XBUFFERLENGTH 512



#define YBUFFERLENGTH 32


extern const float xcoeff[XBUFFERLENGTH];
extern const float ycoeff[YBUFFERLENGTH];




float getSample(void);
void send2OutBuffer(float);

const int Nx = 1;
const int Ny = 2;
const int X_BUFFER_SIZE = 256;
const int Y_BUFFER_SIZE = 16;

float x_buff[X_BUFFER_SIZE];
float y_buff[Y_BUFFER_SIZE];
int ts;
```

```
float y;
float x;


float outVol;
bool outputEnb;


void leftButtonFunc() {
	outVol = outVol/1.26f;
}


void rightButtonFunc() {
		outVol = outVol * 1.26f;
}

void upButtonFunc() {
	outputEnb = !outputEnb;
}

//void initFilter(void){
//	/*	Place your initialization code for this module here
//			This code runs once prior to any input, output, or filtering actions
//
//			It is a good place to initialize any data structures you create in this
module  */
//}

//void filter(void) {
//	/*	This is where you place all of your filtering code
//			This function is called 8K times per second prior to outputting any
sound samples to the DAC
//
//			Ordinarily, each time this function is called you would:
//				read one data sample as input
//				perform any filtering or other computation to create
an output value
//				output the calculated value
//	*/
//}

void initFilter(void) {

	outVol = 1.0f;
	outputEnb = true;
```

```
        ts = 0;
        y = 0.0f;
        x = 0.0f;
}

void filter(void){
        x = getSample();

        //Step 1
        x_buff[(ts %X_BUFFER_SIZE)] = x;

        //y(n) = sum(i)[b[i]x[n-1] - a[n]y[n-1-i]
        for(int i = 0; i < Nx; i++){
                y = y + xcoeff[i] * x_buff[(ts -i) %X_BUFFER_SIZE];
        }
        for(int i = 1; i < Ny; i++){
                y = y - (ycoeff[i] * y_buff[(ts -i)%Y_BUFFER_SIZE]);
        }
        y_buff[(ts%Y_BUFFER_SIZE)] = y;
        ts ++;


        if (outputEnb){
                send2OutBuffer(outVol * y);
        }
        else
                send2OutBuffer(x);
        y=0.0f;

//send2OutBuffer(getSample());

}
```

Part 2:

```
float getSample(void);
void send2OutBuffer(float);
```

```
const int Nx = 1;
const int Ny = 2;
const int X_BUFFER_SIZE = 256;
const int Y_BUFFER_SIZE = 16;

float x_buff[X_BUFFER_SIZE];
float y_buff[Y_BUFFER_SIZE];
int ts;
float y;
float x;
```

```
#define XCOEFLENGTH 512
#define YCOEFLENGTH 32

const float ycoeff[YCOEFLENGTH] = {
        1.0,
        -0.95,
        0.0001,
        0.0001,
        0.0001,
        0.0001,
        0.0001,
        0.0001
};


        const float xcoeff[XCOEFLENGTH] = {
        0.05,
        0.000,
        …
```

Part 3:

```
const int Nx = 12;
const int Ny = 12;
```

```
B = [0.0012826,0,-0.0064129,0,0.012826,0,-0.012826,0,0.0064129,0,-0.0012826]
A =
[1,-6.7843,21.5777,-42.3386,56.7291,-54.2081,37.3992,-18.3975,6.18088,-1.28302,0.125
431]
```

Part 4:

```
xcoef
0.59994,-4.2951,15.2995,-34.7918,55.5072,-64.604,55.5072,-34.7918,15.2995,-4.2951,0.
59994
```

```
Y coef
1,-6.43222,20.5771,-42.0775,60.4562,-63.4748,49.2785,-27.9547,11.1415,-2.83846,0.359
928
```

Part 5:

```
const int Nx = 128;
const int Ny = 1;


Y
1

X

-0.0091068,0.0025091,0.0016753,0.00032888,-0.001101,-0.0015272,-3.8241e-06,0.0032389
,0.0065279,0.0077176,0.0058,0.0018718,-0.0015026,-0.0021823,-0.00019481,0.0021713,0.
0021588,-0.001138,-0.0058231,-0.0085442,-0.0071736,-0.002651,0.0016394,0.0026155,0.0
0015386,-0.0026362,-0.001935,0.0033659,0.010309,0.013909,0.011168,0.003787,-0.002892
1,-0.0041481,5.6626e-05,0.0046508,0.0036067,-0.0049216,-0.016229,-0.022283,-0.018111
,-0.0062514,0.0047206,0.0069469,8.0322e-05,-0.0077066,-0.0060124,0.0087345,0.028926,
0.040448,0.033612,0.011885,-0.0093031,-0.014111,-3.6302e-05,0.017368,0.014296,-0.022
434,-0.080967,-0.12665,-0.1221,-0.053019,0.057736,0.15915,0.20005,0.15915,0.057736,-
0.053019,-0.1221,-0.12665,-0.080967,-0.022434,0.014296,0.017368,-3.6302e-05,-0.01411
1,-0.0093031,0.011885,0.033612,0.040448,0.028926,0.0087345,-0.0060124,-0.0077066,8.0
322e-05,0.0069469,0.0047206,-0.0062514,-0.018111,-0.022283,-0.016229,-0.0049216,0.00
36067,0.0046508,5.6626e-05,-0.0041481,-0.0028921,0.003787,0.011168,0.013909,0.010309
,0.0033659,-0.001935,-0.0026362,0.00015386,0.0026155,0.0016394,-0.002651,-0.0071736,
-0.0085442,-0.0058231,-0.001138,0.0021588,0.0021713,-0.00019481,-0.0021823,-0.001502
6,0.0018718,0.0058,0.0077176,0.0065279,0.0032389,-3.8241e-06,-0.0015272,-0.001101,0.
00032888,0.0016753,0.0025091,-0.0091068
```

Step 6:

```
#include "main.h"
#include "coefficients.h"
#include <stdbool.h>

#define XBUFFERLENGTH 4096
```

```
float xbuffer[XBUFFERLENGTH];
int xindex;

float outVol;
bool outputEnb;

int delayLen = 4000;       // Echo delay (in samples)
float reflect = 0.3f;      // Echo reflection gain

void initFilter(void) {
   for (int i = 0; i < XBUFFERLENGTH; i++) {
      xbuffer[i] = 0.0f;
   }
   xindex = 0;
   outVol = 1.0f;
   outputEnb = true;
}

void filter(void) {
   float xnew = getSample();
   xbuffer[xindex] = xnew;

   // Compute delayed index using circular buffer wrap-around
   int delayedIndex = (xindex - delayLen + XBUFFERLENGTH) %
XBUFFERLENGTH;

   // Echo equation
   float ynew = (1.0f - reflect) * xnew + reflect * xbuffer[delayedIndex];

   if (outputEnb) {
      send2OutBuffer(outVol * ynew);
   } else {
      send2OutBuffer(xnew);
   }

   xindex = (xindex + 1) % XBUFFERLENGTH;
}
```

Step 6: Echo

```
#include "main.h"
```

```
#include "coefficients.h"
#include <stdbool.h>



#define x_buffLENGTH 512
#define YBUFFERLENGTH 32


extern const float xcoeff[x_buffLENGTH];
extern const float ycoeff[YBUFFERLENGTH];




float getSample(void);
void send2OutBuffer(float);

const int Nx = 128;
const int Ny = 1;
const int X_BUFFER_SIZE = 4096;
const int Y_BUFFER_SIZE = 16;

float x_buff[X_BUFFER_SIZE];
float y_buff[Y_BUFFER_SIZE];
int ts;

int delayLen;
float reflect;


float outVol;
bool outputEnb;


void leftButtonFunc() {
	outVol = outVol/1.26f;
}


void rightButtonFunc() {
		outVol = outVol * 1.26f;
}

void upButtonFunc() {
	outputEnb = !outputEnb;
```

```
}

//void initFilter(void){
//        /*      Place your initialization code for this module here
//                        This code runs once prior to any input, output, or filtering actions
//
//                        It is a good place to initialize any data structures you create in this
module  */
//}

//void filter(void) {
//        /*      This is where you place all of your filtering code
//                        This function is called 8K times per second prior to outputting any
sound samples to the DAC
//
//                        Ordinarily, each time this function is called you would:
//                                        read one data sample as input
//                                        perform any filtering or other computation to create
an output value
//                                        output the calculated value
//        */
//}

void initFilter(void) {
                delayLen = 4000;
                reflect = 0.3f;

   ts = 0;
   outVol = 1.0f;
   outputEnb = true;


}

void filter(void) {
   float xnew = getSample();
   x_buff[ts% X_BUFFER_SIZE] = xnew;

   // Compute delayed index using circular buffer wrap-around
   int delayedIndex = (ts - delayLen + X_BUFFER_SIZE) % X_BUFFER_SIZE;


   // Echo
   float ynew = (1.0f - reflect) * xnew + reflect * x_buff[delayedIndex];

   if (outputEnb) {
```

```
        send2OutBuffer(outVol * ynew);
    } else {
        send2OutBuffer(xnew);
    }

    ts = ts + 1;
}
```

Extra Credit:

```
#include "main.h"
#include "coefficients.h"
#include <stdbool.h>
#include "math.h"



#define XBUFFERLENGTH 512
#define YBUFFERLENGTH 32
#define SAMPLE_RATE 8000.0f


extern const float xcoeff[XBUFFERLENGTH];
extern const float ycoeff[YBUFFERLENGTH];




float getSample(void);
void send2OutBuffer(float);

const int Nx = 128;
const int Ny = 1;
const int X_BUFFER_SIZE = 4096;
const int Y_BUFFER_SIZE = 16;
const int COSMOD_LENGTH = 2048;

float x_buff[X_BUFFER_SIZE];
```

```
float y_buff[Y_BUFFER_SIZE];
float cosMod[COSMOD_LENGTH];
int ts;

float modFreq;
int modIndex;
int modStep;

float outVol;
bool outputEnb;


void leftButtonFunc() {
	outVol = outVol/1.26f;
}


void rightButtonFunc() {
		outVol = outVol * 1.26f;
}

void upButtonFunc() {
	outputEnb = !outputEnb;
}

//void initFilter(void){
//	/*	Place your initialization code for this module here
//			This code runs once prior to any input, output, or filtering actions
//
//			It is a good place to initialize any data structures you create in this
module  */
//}

//void filter(void) {
//	/*	This is where you place all of your filtering code
//			This function is called 8K times per second prior to outputting any
sound samples to the DAC
//
//			Ordinarily, each time this function is called you would:
//				read one data sample as input
//				perform any filtering or other computation to create
an output value
//				output the calculated value
//	*/
//}
```

```
void initFilter(void) {
   // Initialize echo buffer
   for (int i = 0; i < X_BUFFER_SIZE; i++) {
      x_buff[i] = 0.0f;
   }

   ts = 0;
   modIndex = 0;
                  modFreq = 500.0f;
   outVol = 1.0f;
   outputEnb = true;

   // Fill cosine modulation buffer with one period of cos(2*pi*n/N)
   for (int i = 0; i < COSMOD_LENGTH; i++) {
      cosMod[i] = cosf(2.0f * 3.14f * i / COSMOD_LENGTH);
   }

   // Compute modulation step size
   modStep = (int)((modFreq * COSMOD_LENGTH) / SAMPLE_RATE + 0.5f);
}



void filter(void) {
   float xnew = getSample();
   x_buff[ts% X_BUFFER_SIZE] = xnew;

   float ynew = xnew * cosMod[modIndex % COSMOD_LENGTH];


   if (outputEnb) {
      send2OutBuffer(outVol * ynew);
   } else {
      send2OutBuffer(xnew);
   }

   ts = ts + 1;
                  modIndex = modIndex + modStep;
}
```