Lab 3
FIR Filters
EE 445S

Enoc Balderas Daniel Diamont

March 18, 2019

Date Performed:	Febraury 18, 2019
Instructor:	Professor Evans

1 Introduction

For this lab we experimented with the FIR and IIR filter design process. We started off by using MATLAB's fdatool to experiment with different filter designs, and to gain intuition about the rules of thumb for meeting filter specifications. Additionally we plotted the filter magnitude and phase to verify that the specifications have been met. Finally we used the filter coefficients generated from MATLAB to implement IIR and FIR filters in C on our DSP board.

1.1 FIR

We experimented with both sample based processing and frame based processing. Within our sample based implementation we tested the difference in speed between a linear buffer and a circular buffer. We also compared our sample based circular buffer implementation to a frame based processing implementation and compared the tradeoffs between them.

1.2 IIR

We experimented with a direct form II implementation and also a cascade of second order biquads. We compared the implementation complexity of both methods and the number of coefficients required for each. For instance biquads are reusable logic blocks, but there is lots of programming overhead setting up the biquad implementation.

2 Methods

2.1 Linear and Circular buffering

We designed a 30th order FIR filter in MATLAB and recorded its magnitude response for frequencies between 1 - 24 kHz. After confirming the response in MATLAB we exported the filter coefficients to a C file for implementation on DSP board. On the DSP board we tested the clock cycles spent with both linear and circular buffers.

2.2 Frame based processing

Using the same FIR filter coefficients we experimented with a frame based approach and compared the results to the circular buffer sample based implementation. We also experimented with the compiler optomizations available in CCS for a convolution method written in C compared to one in assebly.

2.3 IIR filtering

We designed a 6th order IIR filter in MATLAB and recorded its magnitude response for frequencies between 1 - 24 kHz. After confirming the response in MATLAB we exported the filter coefficients to a C file for implementation on DSP board. We compared the performance of the direct form implementation with a second order biquad cascade.

3 Results

3.1 Linear and Circular buffering

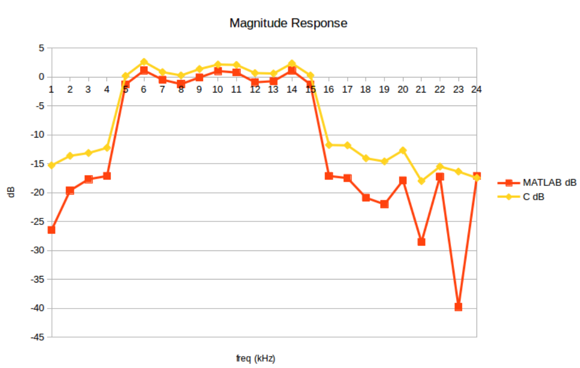

Figure 1: FIR magnitude response.

Table:

freq (Hz)	MATLAB dB	C dB
1	-26.4764686348	-15.289
2	-19.6317280623	-13.639
3	-17.6768240012	-13.152
4	-17.123864155	-12.252
5	-1.3024968679	0.172
6	1.111588031	2.607
7	-0.4856202938	0.828
8	-1.2464254947	0.257
9	-0.0834986716	1.364
10	1.016410665	2.144
11	0.7589322154	2.076
12	-0.9311437073	0.668
13	-0.7304622226	0.588
14	1.0955609737	2.3454
15	-1.3024968679	0.257
16	-17.123864155	-11.768
17	-17.4842248929	-11.835
18	-20.8880803374	-14.067
19	-22.0160480332	-14.609
20	-17.8808698809	-12.69
21	-28.5482145367	-17.993
22	-17.2303232362	-15.494
23	-39.7758165405	-16.363
24	-17.123864155	-17.458

Code:

```
/*********Coefficients************/
float B[N+1] = {
-0.020403335532,/* h[0] */
-0.034752540459,/* h[1] */
0.011675748908,/* h[2] */
-0.081290790410,/* h[3] */
-0.020874660735,/* h[4] */
0.052412925124,/* h[5] */
0.022816502846,/* h[6] */
0.018922432476,/* h[7] */
0.074229254675,/* h[8] */
-0.003637178873,/* h[9] */
-0.052161838503,/* h[10] */
0.022826239218,/* h[11] */
-0.123591009216,/* h[12] */
-0.273422708726,/* h[13] */
0.108309337556,/* h[14] */
0.458629527602,/* h[15] */
0.108309337556,/* h[16] */
-0.273422708726,/* h[17] */
-0.123591009216,/* h[18] */
0.022826239218,/* h[19] */
-0.052161838503,/* h[20] */
-0.003637178873,/* h[21] */
0.074229254675,/* h[22] */
0.018922432476,/* h[23] */
0.022816502846,/* h[24] */
0.052412925124,/* h[25] */
-0.020874660735,/* h[26] */
-0.081290790410,/* h[27] */
0.011675748908,/* h[28] */
-0.034752540459,/* h[29] */
-0.020403335532,/* h[30] */
};

/*********Linear buffer************/
xLeft[0]  = CodecDataIn.Channel[LEFT]; // current LEFT input value
yLeft  = 0; // initialize the LEFT output value

for (i = 0; i <= N; i++)
{
  yLeft  += xLeft[ i]*B[i]; // perform the LEFT dot-product
}
```

```
for (i = N; i > 0; i--)
{
  xLeft[i] = xLeft[i-1];        // setup xLeft[] for the next input
}

CodecDataOut.Channel[LEFT]  = yLeft; // setup the LEFT value

/*********Circular buffer**********/
*pLeft  = CodecDataIn.Channel[LEFT]; // store LEFT input value

output = 0; // set up for LEFT channel
p = pLeft; // save current sample pointer

if(++pLeft > &xLeft[N]) // update pointer, wrap if necessary
  pLeft = xLeft; // and store

for (i = 0; i <= N; i++)
{
  output += *p-- * B[i];   // multiply and accumulate

  if(p < &xLeft[0])        // check for pointer wrap around
    p = &xLeft[N];
}

CodecDataOut.Channel[LEFT]  = output; // store filtered value
```

Figure 2: Linear buffer clock cycles.

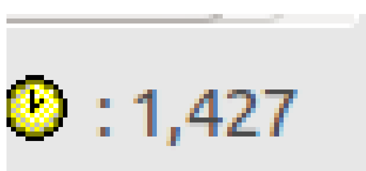

Figure 3: Circular buffer clock cycles.

3.2 Frame based processing

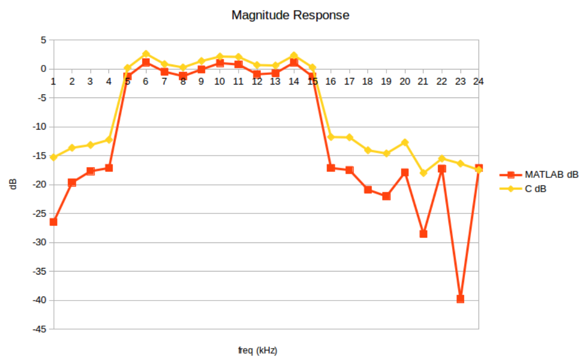

Figure 4: FIR magnitude response.

```
/*********Coefficients*************/
float B[N+1] = {
-0.020403335532,/* h[0] */
-0.034752540459,/* h[1] */
0.011675748908,/* h[2] */
-0.081290790410,/* h[3] */
-0.020874660735,/* h[4] */
0.052412925124,/* h[5] */
0.022816502846,/* h[6] */
0.018922432476,/* h[7] */
0.074229254675,/* h[8] */
-0.003637178873,/* h[9] */
-0.052161838503,/* h[10] */
0.022826239218,/* h[11] */
-0.123591009216,/* h[12] */
-0.273422708726,/* h[13] */
0.108309337556,/* h[14] */
0.458629527602,/* h[15] */
0.108309337556,/* h[16] */
-0.273422708726,/* h[17] */
-0.123591009216,/* h[18] */
0.022826239218,/* h[19] */
-0.052161838503,/* h[20] */
-0.003637178873,/* h[21] */
0.074229254675,/* h[22] */
0.018922432476,/* h[23] */
0.022816502846,/* h[24] */
```

```
0.052412925124,/* h[25] */
-0.020874660735,/* h[26] */
-0.081290790410,/* h[27] */
0.011675748908,/* h[28] */
-0.034752540459,/* h[29] */
-0.020403335532,/* h[30] */
};
```

Figure 5: Clock cycles to process one frame.

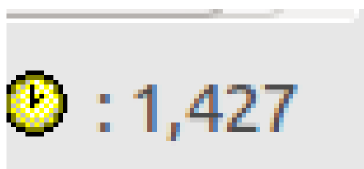

Figure 6: Clock cycles to process one sample.

Compiler Optimizations

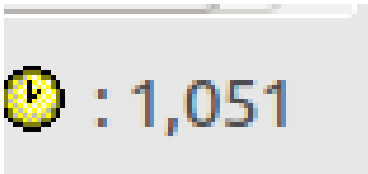

Figure 7: C code optimization level 0.

Figure 8: C code optimization level 1.

Figure 9: C code optimization level 2.

Figure 10: C code optimization level 3.

Figure 11: ASM code optimization level 0.

Figure 12: ASM code optimization level 1.

Figure 13: ASM code optimization level 2.

Figure 14: ASM code optimization level 3.

3.3 IIR filtering

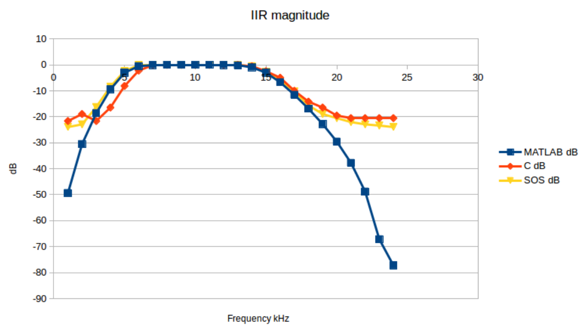

Figure 15: IIR magnitude response.

Table:

freq (Hz)	MATLAB dB	C dB	SOS dB
1	-49.336911533	-21.6164638623	-23.8559390512
2	-30.4835221116	-18.9224923843	-22.8898454597
3	-18.5780755991	-21.6164638623	-16.2092403443
4	-9.4231800523	-16.4237176522	-8.3784900031
5	-3.0102999566	-8.1242506928	-2.3578900898
6	-0.4907931843	-2.1875730714	0
7	-0.0431819157	0	0
8	-0.0013881395	0.0008193849	0
9	-0.000001107	0.0016386925	0
10	-2.45915822097029E-06	0.0024579228	0
11	-0.0012585765	0.0032770759	0
12	-0.0255740552	0.0040961518	0
13	-0.1988119832	0.0049151504	0
14	-0.9408492775	-0.5061173053	-0.5061173053
15	-3.0102999566	-2.4443175655	-2.7088828801
16	-6.7158691567	-4.9430922976	-6.1860504326
17	-11.500586955	-9.9793317575	-10.1886404311
18	-16.8767672084	-14.144850606	-15.595863949
19	-22.8006774261	-16.4237176522	-18.9224923843
20	-29.5500422777	-19.5217568519	-20.5061173053
21	-37.7484943246	-20.5061173053	-22.0205315841
22	-48.8004344666	-20.5061173053	-22.8898454597
23	-67.1478923263	-20.5061173053	-23.3594673767
24	-77.232927344	-20.5061173053	-23.8559390512

Code:

```
/*********Coefficients*************/
float B[N + 1] = {
    0.108499,/* B[0] */
           0,/* B[1] */
   -0.325498,/* B[2] */
           0,/* B[3] */
    0.325498,/* B[4] */
           0,/* B[5] */
   -0.108499,/* B[6] */
};

float A[N + 1] = {
           1,/* A[0] */
    -1.13553,/* A[1] */
    0.944316,/* A[2] */
   -0.637821,/* A[3] */
    0.544417,/* A[4] */
   -0.173657,/* A[5] */
   0.0459008,/* A[6] */
};
```

```
/*********df_one******************/
float sum = 0;
int i;

//dot product for numerator
for(i = 0; i < N + 1; i++)
{
sum += B[i]*x[i];
}

//dot product for denominator
for(i = 1; i < N + 1; i++)
{
sum -= A[i]*y[i];
}

y[0] = A[0]*sum;

//setup for next iteration
for(i = N; 0 < i; i--)
{
x[i] = x[i - 1];
y[i] = y[i - 1];
}
```

```
/***********SOS*****************/
float biquad(int index, float value)
{
x[index][0] = value;
float ret_val = SOS[index][0]*x[index][0] + SOS[index][1]*x[index][1] + SOS[index][2]*x[inde
SOS[index][3]*y[index][1] + SOS[index][4]*y[index][2];

y[index][0] = ret_val;

x[index][2] = x[index][1];
x[index][1] = x[index][0];

y[index][2] = y[index][1];
y[index][1] = y[index][0];

return ret_val;
}
```

4 Discussion

4.1 Linear and Circular buffering

We learned that the MATLAB fda tool is very good for quickly designing filters. After examining the magnitude plot in MATLAB we got a good idea about what to expect from the real time implementation.

4.2 Frame based processing

One big take away was that the compiler is a way better optimizer than we are. Also we realized that some compiler optimizations lead to very hard to understand assembly code.

4.3 IIR filtering

It was a surprise that the IIR filter achieved such great performance with far less filter coefficients. Also it was interesting to see the effect of using incorrect gains in the cascuade of biquads.

5 Answers to questions

1. Number of taps before you run out of time for FIR filtering.

   Answer: For an arbitrary processor using a circular buffer to implement an FIR filter, where m is the time it takes for a multiply to complete and a is the time it takes for an addition to complete. We would want our processing to take less than our sampling period T_s. So for an FIR filter with N coefficients we would have N multiply and $N-1$ add opperations.

   $$T_s > mN + a(N-1) \tag{1}$$

   Solving for N we arrive at this constraint for the maximum number of taps.

   $$\frac{T_s + a}{m + a} > N \tag{2}$$

2. Why use circular buffers?

   Answer: Using a circular buffer eliminates the time needed to shift values using a linear buffer implementation, which would scale shifts linearly.

3. What happens if you do not multiply the scale factor in an IIR filter implementation using a cascade of second-order sections (SOS)?

   Answer: If we do not multiply by a scaling factor our system may be unstable. When we design a transfer function we use a set of specifications to guide our choices of poles and zeros. From this set of poles and zeros we arrive at a single section implementation. To create a more practical implementation we instead break our single section into many second order sections. As a result of factoring our large polynomial into many second order polynomials we get a set of scaling factors between each section. By ignoring these scaling factors, we would essentially change the location of the poles and zeros, and by extent the response of the entire filter.

4. Advantages and disadvantages of FIR vs. IIR filters.

   Answer:

   FIR

   Pros:

   - Linear phase
   - BIBO stability

   Cons:

   - Large number of coefficients

IIR

Pros:

- Low number of Coefficients

Cons:

- May become unstable
- non linear phase