

real time digital singal processing

LAB 4 REPORT



February 19, 2015

Imperial College London

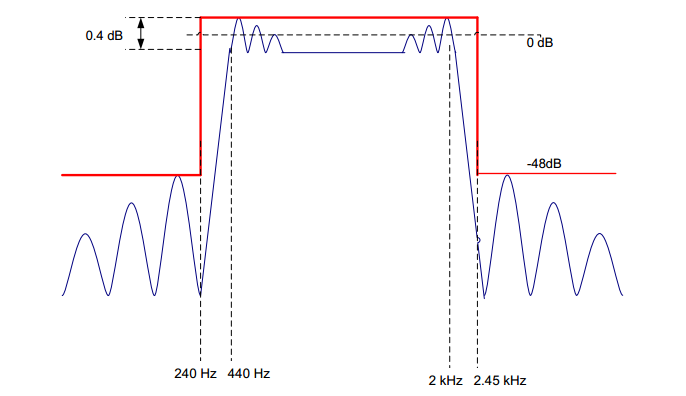
JIABO ZHOU (JZ2611), HAO DING (HD1812)

**CONTENT**

1. MATLAB
   1. Filter design
   2. Proof of the expected frequency response
2. Code Explanation
   1. Operation Principles
   2. Non-circular FIR Filter Design
   3. Code efficiency with complier optimisation
3. Circular FIR filter design
   1. Naive Circular FIR Filter Design
   2. Improvements on Basic Design
   3. Code coefficient and comparison
4. Frequency response analysis

**MATLAB**

* 1. **Filter Design**

****The FIR filter required to build in MATLAB has specifications shown in Figure 1.

Passband

**Figure 1.** FIR specifications

Transition Band

Transition Band

Passband Ripple

Stopband Ripple

Stopband

Stopband

In MATLAB, the Parks-McClelland algorithm is suggested to approximate required filters by using *firpmord* and *firpm* functions. Code shown as the following:

**Figure 2.** MATLAB filter design code

|  |
| --- |
| close all;  clear all;    f1=240;f2=440;f3=2000;f4=2300;% frequency boundary  f=[f1,f2,f3,f4];%make array of frequencies    rp=0.4;% ripple in dB  sa=48;% minimum stop band attenuation in dB  dev1=(10^(rp/20)-1)/(10^(rp/20)+1);%calculate pass band deviation  dev2=10^(-sa/20);%calculate stop band deviation  dev=[dev2,dev1,dev2];%make array of deviations    Fs=8000;%specify sampling rate  a=[0,1,0];%specify amplitude    [N,Fo,Ao,W] = firpmord(f,a,dev,Fs);%function to approximate filter  coefs = firpm(N,Fo,Ao,W);%function to calculate frequency coefficients.    freqz(coefs,1,1024,8000)%Plot frequency and phase response  H = tf(coefs,1);%derive transfer function  figure;  pzmap(H);%plot pole and zero map of filter  grid on;    %The following code store filter coefficients in format readable for c.  fileID = fopen('fir\_coef.txt','w');  fprintf(fileID,'double coefs[]={');  for i = 1:length(coefs)  fprintf(fileID,'%f,',coefs(1,i));  end  fprintf(fileID,'};');  fclose(fileID); |

For *passband*, according to the equations:

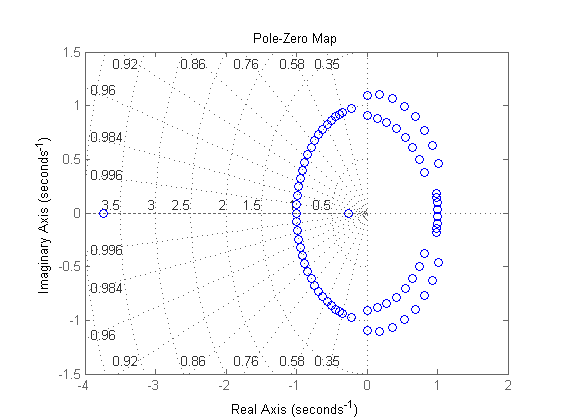
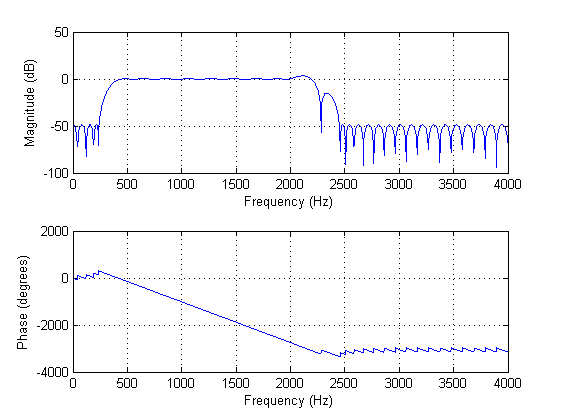
We know that the difference between these is the ripple in dB, *rp.*

Thus, we can combine them:

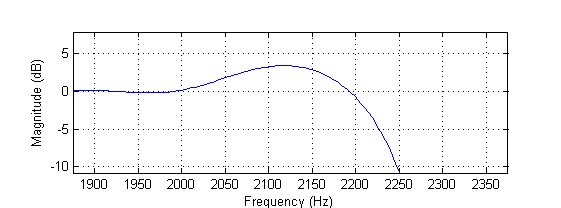
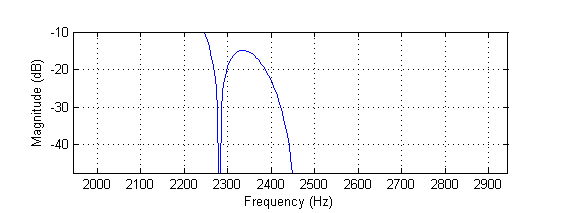
Same procedure for stopband, the function is shown below:

Since it is a FIR filter, when we use function *freqz* we need to set the second parameter to 1 (FIR only has one pole). Meanwhile, we use 1024 (resolution) and 8000 (recovering the frequency) to print the graph nicely.

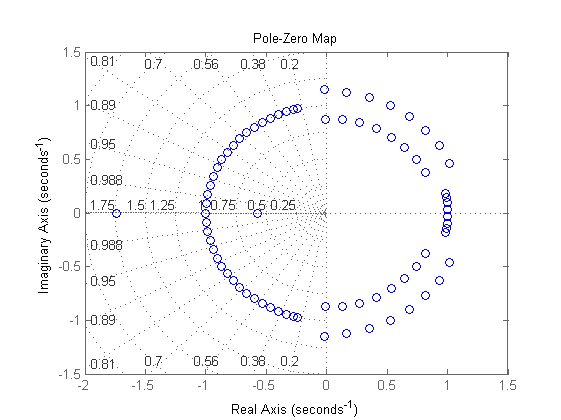
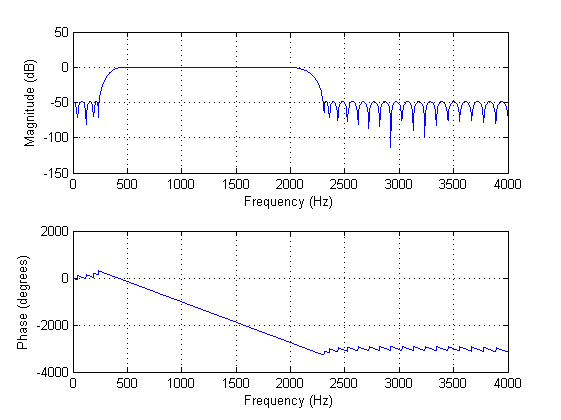
**1.2 Proof of the expected frequency response**

Frequency and phase response as well as pole map are included in Figure 3, from which we can see that specifications described are satisfied well. However, at the end of pass band and middle of transition band exist two bumps, which may bring in potential risk, shown in Figure 4.

**Figure 3.** (Left)Frequency and phase response. (Right) Pole and zero map



**Figure 4.** (Left)Bump at the end of pass band. (Right) Bump in the middle of transition band

Accordingly, we slightly narrow the transition band range by change the boundary from 2.45 kHz to 2.3 kHz. In z-domain, it is equivalent to a small shift of zeroes. Frequency response and pole map of modified filter is shown in Figure 5, with pole shift marked. At position **A**, poles are closer to each other, which avoids a single zero to pull down amplitude too early so that the bump in middle of transition band is removed. At position **B**, a pole pair is moving to opposite direction, which reduces gain at that frequency and removes the bump at the end of pass band.

**Figure 5.** (Left)Frequency and phase response. (Right) Pole and zero map

**A**

**B**

Actually, we can increase the order by 1 or 2 to make sure the filter fits the specs (*firpmord* only generate the approximate order required for filter). But once we use this way, it will make our program complicated (more calculations needed for convolution function). We prefer to minimise the N as small as possible. The order of N (77) is calculated.

Coefficients of filter with modified boundary frequency is then included and used in Code composer. Method of using coefficients is explained in next section.

**Code explanation**

**2.1 Operation principles**

The basic idea of this section is similar to the previous lab. Interrupt ISA\_AIC function is called every time signal is received. Input sample is read from codec, using mono\_read\_16Bit (). Current and some past samples are stored and convoluted with FIR filter. The corresponding time equation of filtering is

For any instant time n, an output is generated based on current and past input samples. Then it is expected that actual output varies as frequency, as described in the frequency response graph in MATLAB.

FIR filter coefficients are saved in a text file in an array format:

*double coefs[]={0.002154,….};*

The text file is then saved in the same folder as c file, and included using command:

*#include "fir\_coef.txt"*

**2.2 Non-circular FIR filter design**

**Code:**

|  |
| --- |
| **#include** "fir\_coef.txt"  **#define** N 78  **short** x[N]={0}; //initialise array to zero  **void** ISR\_AIC()  {  **int** i;  **for**(i=N-1;i>0;i--){//loop through every element from last to first  x[i]=x[i-1];//shift elements to right and delete the last one  }  x[0]=mono\_read\_16Bit(); //store input  non\_circ\_FIR();  }    **void** non\_circ\_FIR()  {  **int** i;  **double** y=0;  **for**(i=0;i<N;i++){ //loop through the array and perform convolution  y+=x[i]\*coefs[i];  }  mono\_write\_16Bit((**short**)y);//write to output port  } |

A naïve implementation of non-circular FIR filter is shown above. Necessary files and filter coefficient number are defined and included in the beginning. Input array is initialised to zero.

Inside the interrupt function is a for-loop which shifts all elements one position to right and delete the last element. After that, input is read and saved in position 0. Figure is shown below.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |

**Input array:**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |

**Updated array:**

**New Element**

Since the order is 77, we need 78 parameters in our convolution function.

The function non\_circ\_FIR () performs convolution by looping through the whole array and doing multiply accumulation on input elements and corresponding filter coefficients.

**2.3 Code efficiency with compiler optimisation**

**Table 1.** Optimisation level and clock cycle

|  |  |
| --- | --- |
| **--opt\_level** | **Clock cycle** |
| no | 5145 |
| 0 | 4208 |
| 2 | 1024 |

Table 1 shows that clock cycle number is reduced significantly with optimisation level. Notice that bread points are placed at the beginning and end of ISR\_AIC function, which measures the total time cycle of interrupt function, including reading and writing samples. Each cycle count is the best case found after running code a few times. In this report, all clock cycle are measured using this method.

The opt\_level is an option for user to optimise their code.

**Table 2.** Optimisation priority

|  |  |
| --- | --- |
| --opt\_level | performance |
| no | Disable optimisation |
| 0 | Optimisation priority is compilation time and debugging ease |
| 2 | Compiler optimise primarily for performance |

**Circular FIR filter design**

In this section, three different implementation of circular FIR filters are constructed and compared, including naïve implementation, using linear phase property and using double array size. The fundamental ideal of circular buffer is to store input data along the array repeatedly, which avoid the trouble to shift input data every time a new input is logged.

**3.1 Naïve Circular FIR Filter Design**

This is a basic realisation of the idea of a circular buffer.

|  |
| --- |
| **#include** "fir\_coef.txt" //include filter coefficients  **#define** N 78 //filter coefficient number  **short** x[N]={0}; //initialise array to zero  **short** \* ptr=&x[0]; //initilise the pointer to first element  **void** circ();  **void** ISR\_AIC()  {  circ();  }  **void** circ(){  **int** i;  **double** y=0;  **short**\* ptr\_loop; //pointer to loop through input array  **double**\* ptr\_coef=&coefs[0];//pointer to filter coefficient array    \*ptr=mono\_read\_16Bit(); //read input  ptr\_loop=ptr; //initilise loop pointer to latest reading    **for**(i=0;i<N;i++){  y+=\*ptr\_loop\*(\*ptr\_coef);//perform convolution  **if**(ptr\_loop==&x[0]){  ptr\_loop=&x[N-1]; //loop back to beginning if loop pointer reaches end  ptr\_loop++; //adjustment to make sure pointer at 0 position at  } //the end of for loop  ptr\_coef++; //increment filter pointer  ptr\_loop--; //decrement loop pointer  }    **if**(ptr==&x[N-1]){ //pointer to store input loop back to beginning  ptr=&x[0]; //after reaching the end of the array  ptr--;  }  ptr++; //increment pointer    mono\_write\_16Bit((**short**)y); |

In the naïve implementation, we built an array with size N and used two pointers *ptr\_coef* and *ptr\_loop*, which controls filter coefficient array and input array respectively.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |

**Input:**

**Both shift to right**

**ptr**

**ptr\_loop**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |

**Coefs:**

**Shift to left**

**ptr\_coef**

*Ptr* is a global input pointer which is initialised to the first position of input array and incremented each time new data is logged. So *ptr* always points to the latest element and the oldest element is always at *ptr+1*. Therefore, *ptr* should multiply with *coefs*[0], *ptr+1* should multiply with *coefs*[1],etc. In our program, we perform convolution from the oldest sample to latest, so loop pointer shift to right while coefficient pointer shift from right to left.

**Notice that there are some special cases when pointers are required to be manually set to the correct position to show circularity.**

The first case is inside the for-loop. Every time loop pointer reaches the right most element after multiply accumulation, it is reset to position 0. Loop pointer is also incremented to cancel out the effect of decrement at the end of for-loop.

|  |
| --- |
| **if**(ptr\_loop==&x[0]){  ptr\_loop=&x[N-1]; //loop back to beginning if loop pointer reaches end  ptr\_loop++; //adjustment to make sure pointer at 0 position at  } |

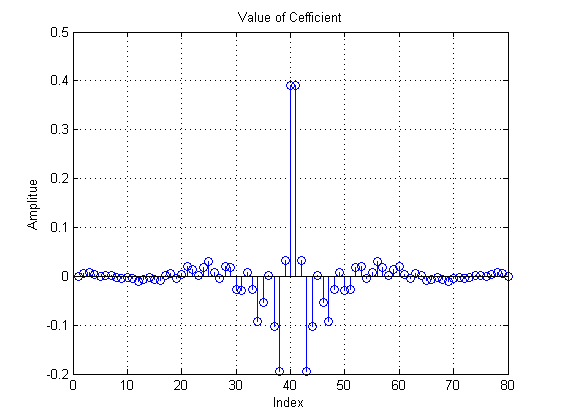
Another case is about input pointer, which is incremented every time interrupt function is called. Similarly, it should loop back to position 0 once reaches end of array.

|  |
| --- |
| **if**(ptr==&x[N-1]){ //pointer to store input loop back to beginning  ptr=&x[0]; //after reaching the end of the array  ptr--;  } |

**3.2 Improvement on Basic Design**

Linear phase

Since the function we used in MATLAB is *firpmord* which follows the Parks-MaClellan algorithm, in which case we will get a linear phased FIR filter. The trick of linear phase filter (also referred to as constant group delay) is the coefficients within it are real and symmetric. The details are shown below:



Because the newest and the oldest have the same weights, it will reduce nearly half cycles if we combine them together. We use two pointers to represent the newest and the oldest value stored in buffer (named p\_f and p\_l respectively), and manipulating them simultaneously by the function:

|  |
| --- |
| if(ptr==&x[N-1]){  p\_l=&x[0];  ptr=&x[0];  ptr--;  }  //we use a for loop to perform convolution function.  for(i=0;i<N/2;i++){  y+=\*p\_fir\*(\*p\_f+\*p\_l);  //avoding overflow, we need to reset p\_f once it reaches  //boudary.  if(p\_f==&x[0]){  p\_f=&x[N-1];  p\_f++;  }  //avoding overflow, we need to reset p\_l once it reaches  //boudary.  if(p\_l==&x[N-1]){  p\_l=&x[0];  p\_l--;  }  //changing coefficient to meet our function  p\_fir++;  p\_f--;  p\_l++;  }  //increase pointer by 1. get ready for next interrupt  ptr++;  //write down the result to output port.  //since it is 16bits, we need to change the type from double to  //short.  mono\_write\_16Bit((short)y);  } |

For each interrupt, we will add ptr by 1 to deflect losing our old data (shift the position of our pointer by 1).

Shift 1 buffer each time

ptr

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| X[0] | X[1] | X[2] | X[3] | … | X[73] | X[74] | X[75] | X[76] | X[77] |

There is a special case, once our pointer reaches boundary, we have to pull it back to the initial buffer (avoiding overflow).

ptr

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| X[0] | X[1] | X[2] | X[3] | … | X[73] | X[74] | X[75] | X[76] | X[77] |

ptr

We add an offset to ptr to maintain the whole procedure.

|  |
| --- |
| if(ptr==&x[N-1]){  p\_l=&x[0];  ptr=&x[0];  ptr--;  } |

Since the ptr is moving from left to right, the oldest value will next to the newest value.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| X[0] | X[1] | X[2] | X[3] | … | X[73] | X[74] | X[75] | X[76] | X[77] |

p\_l

p\_f

As before, we need to change the position of them when either of them reaches boundary.

|  |
| --- |
| //avoding overflow, we need to reset p\_f once it reaches  //boudary.  if(p\_f==&x[0]){  p\_f=&x[N-1];  p\_f++;  }  //avoding overflow, we need to reset p\_l once it reaches  //boudary.  if(p\_l==&x[N-1]){  p\_l=&x[0];  p\_l--;  } |

Offset is used here as well.

Double Memory

Using the property of linear phase, we do need to execute several branches depending on conditions are satisfied or not. Since the judgements are included in for loop which complex our code and wasting time. We can simplify this by double memory, in which case judgement is eliminated.

In order to double our original data set, we add one more pointer (ptr2) whose value is synchronise with ptr1, but with a different place. The starting point of *ptr\_f* is correspond to ptr2 (input will always be the newest value), the gap between *p\_f* and *p\_l* is fixed which is 76, so by processing the for-loop, *p\_f* and *p\_f* will never overlap.

|  |
| --- |
| **void** Double memory(){  //declare parameters initially, including type and name.  **int** i;  // we set y to double which avoid overflow.  **double** y=0;  //p\_f pointed to newest number, p\_l pointed to oldest number.  **short**\* p\_f=ptr2;  **short**\* p\_l=ptr1;  **double**\* p\_fir=&coefs[0];  //get readings from input port.  \*ptr1=mono\_read\_16Bit();  //copying data to fulfil the buffer.  \*ptr2=\*ptr1;  //increasing p\_l by 1 to make sure it points to the oldest number.  p\_l++;  //Although the buffer size is doubled, we still need to reset the pointer index  //when it reaches boundary.  **if**(ptr1==&x[N-1]){  ptr1=&x[0];  ptr1--;  }  **if**(ptr2==&x[2\*N-1]){  ptr2=&x[N];  ptr2++;  }  //using the property of linear phase, we match the inputs with  //the same coefficient together.  **for**(i=0;i<N/2;i++){  y+=\*p\_fir\*(\*p\_f+\*p\_l);  p\_fir++;  p\_f--;  p\_l++;  }  //increment pointer avoiding over-written.  ptr1++;  ptr2++;  mono\_write\_16Bit((**short**)y);  } |

Section 2

Section 1

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| X[0] | X[1] | … | X[76] | X[77] | X[78] | X[79] | … | X[154] | X[155] |

ptr2

ptr\_f

ptr1

ptr\_l

Section 2

Section 1

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| X[0] | X[1] | … | X[76] | X[77] | X[78] | X[79] | … | X[154] | X[155] |

ptr2

ptr\_f

ptr\_l

ptr1

When ptr 2 and ptr 1 reach their boundary, since x[78] was the oldest value, it won’t influence the result if ptr\_l jump to section ‘2’.

**3.3 Code Efficiency and comparison**

Code performance with and without compiler optimisation is measured as the following:

**Table 2.** Optimisation level and clock cycle

|  |  |  |  |
| --- | --- | --- | --- |
| **--opt\_level** | **Basic Circular** | **Circular using linear phase** | **Circular with linear phase and double input array** |
| no | 5640 | 3737 | 2826 |
| 0 | 3756 | 2432 | 1729 |
| 2 | 623 | 403 | 313 |

Table 2 shows that the code efficiency is improved significantly as optimisation level increases.

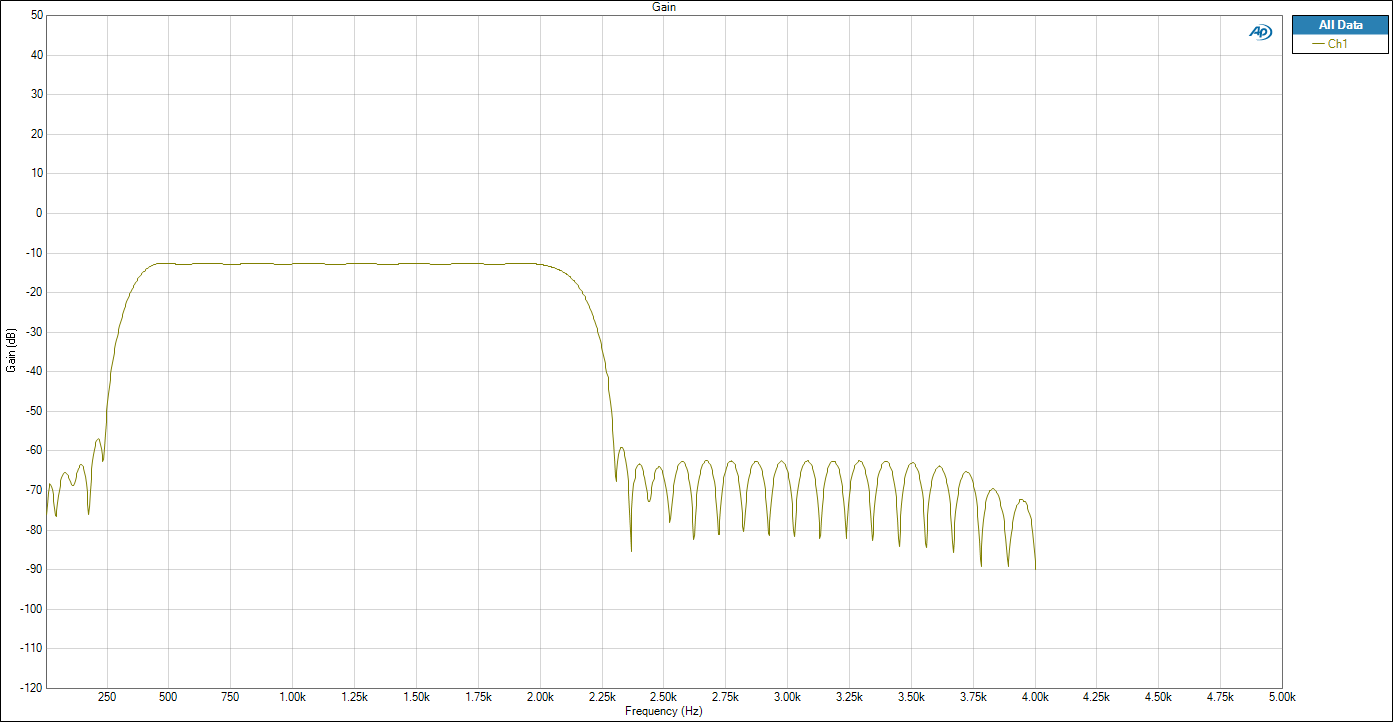
It can be seen from the figure that compiler optimisation significantly improves code performance, especially at optimisation level 2, where pipelining, loop unrolling and optimisation happen. It ensures that processor units are maximally used, pipeline stalls are maximally avoided. While optimisation level 0 only remove used code. It also focuses on improving debugging and compiling ease, which may result in worse code speed.

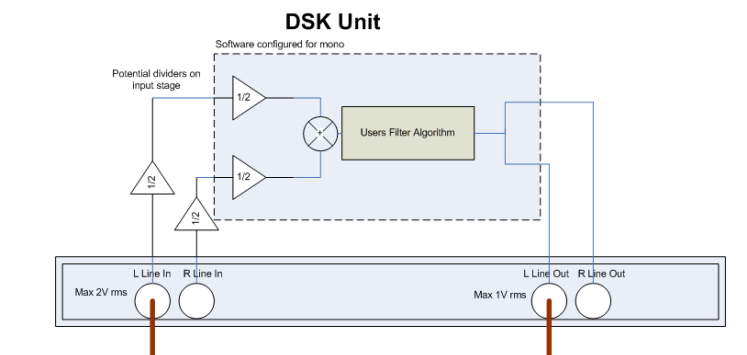
Algorithms with linear phase have the advantage that less multiply accumulation is required. In the for-loop, it only requires to loop through half of the input array, which significantly reduce the amount of calculation for processor.

Double array algorithm is efficient in the sense that it reduces removes some if-statement inside for-loop. Hence, cycle count in each loop is reduced, resulting in less total time necessary for execution.

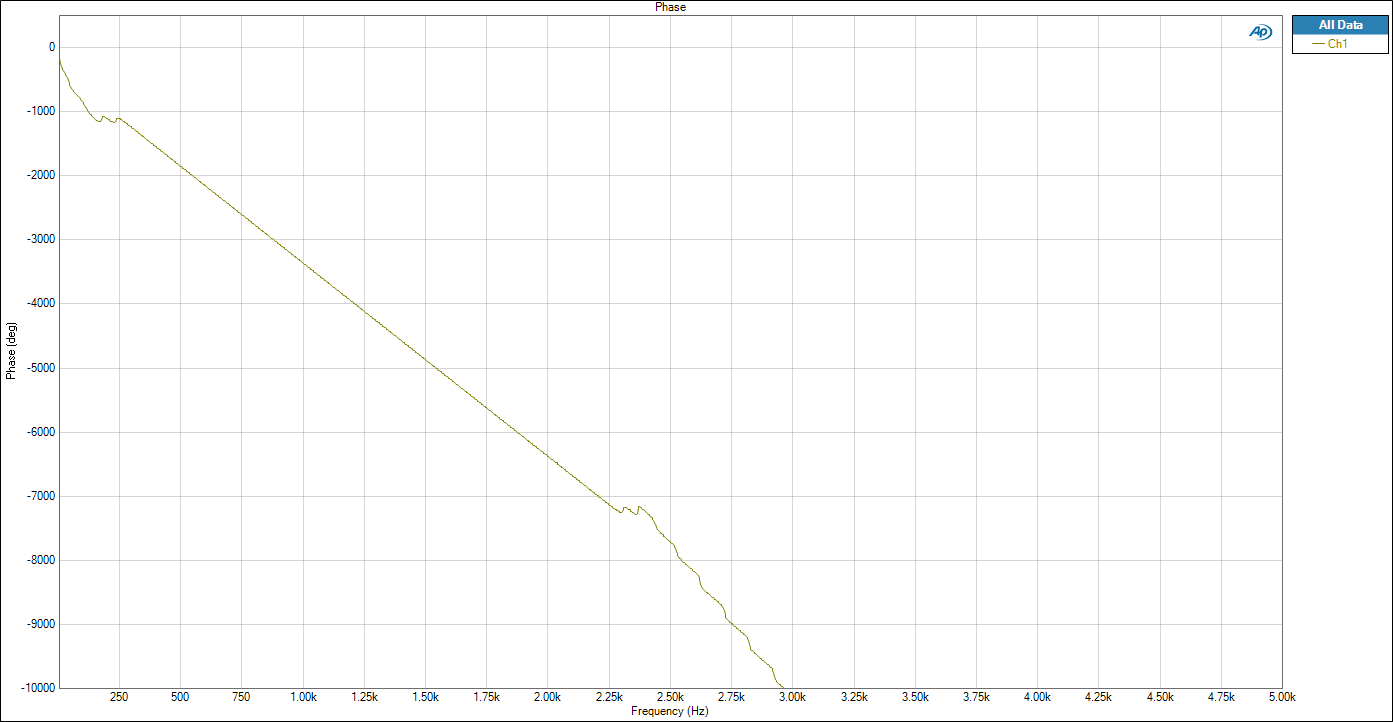
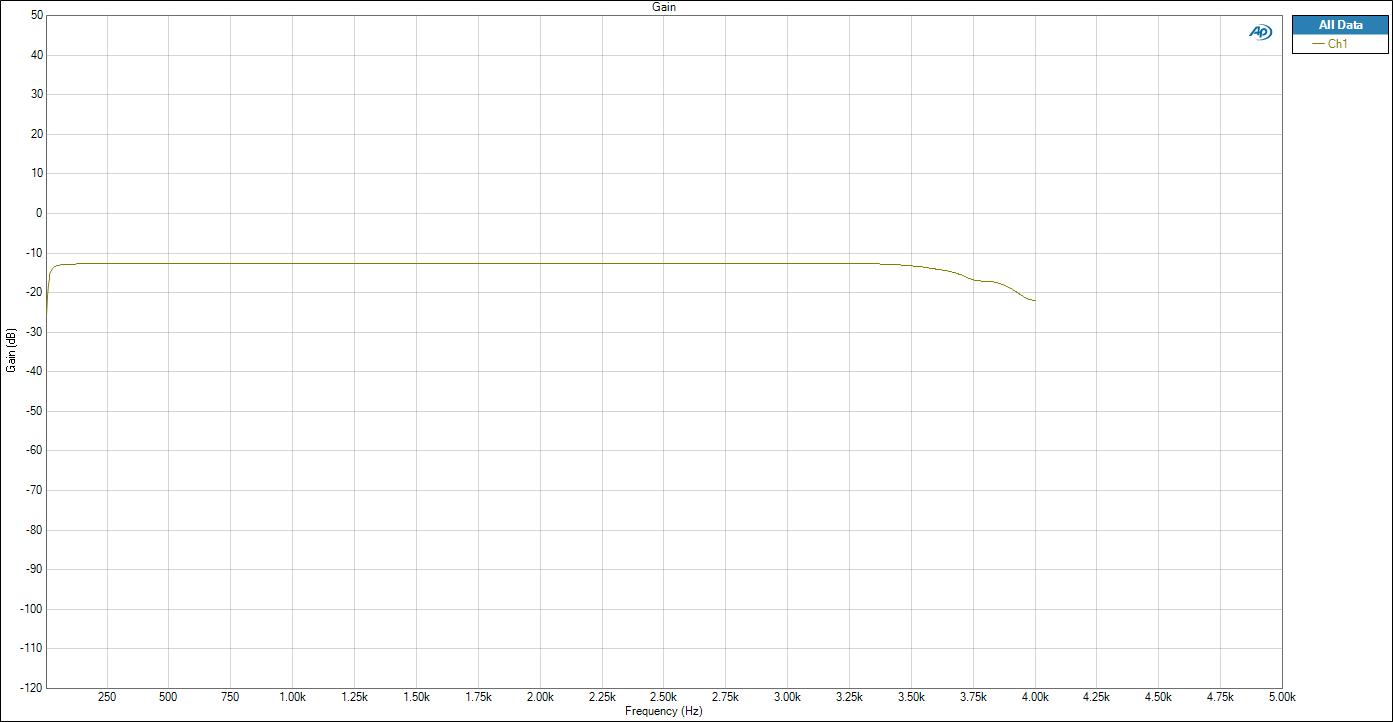
Notice that TI processor is highly optimised for MAC process. However, if the MAC part of user code is not written in a format recognised by compiler. Code turned out to be very slow since optimisation no longer performs. We have experience that a ‘smart’ algorithm turned out to be much slower than a normal algorithm.

**Filter analysis**

Our fastest implementation of FIR filter is used in this section to investigate frequency and phase response. Frequency response is shown below:

From the frequency response, most specifications of required filter are satisfied, including boundary frequency, stop-band attenuation, and pass-band ripple. However, the pass-band gain is expected to be 0 dB. This is due to the structure of audio analyser, shown below.

There are two amplifier with gain one half. In decibel, 20log-0.25= -12, which agrees with our observation. So that the actual pass-band gain of our filter is around 1.

Phase response is plotted. In the frequency range between 240 Hz and 2.25 Hz, a straight downward sloping line is observed. This is the evidence of linear phase in pass-band.

|  |
| --- |
| **void** ISR\_AIC()  {  mono\_write\_16Bit(mono\_read\_16Bit());  } |

At the highest and lowest frequency, observed filter response is different with our expectation. Gain is lower than the theoretical value produced in MATLAB. Therefore, we generate a frequency response graph when sample is directly read to output without filtering. It turned out that the frequency response is not a horizontal line at high and low frequencies. It explains our observation, and our filter behaviour is close to theory.