**RTDSP LAB 4 REPORT**

**Real-time Implementation of FIR Filters**

**HAO DING, JIABO ZHOU**

**Content**

MATLAB filter design

Code Explanation

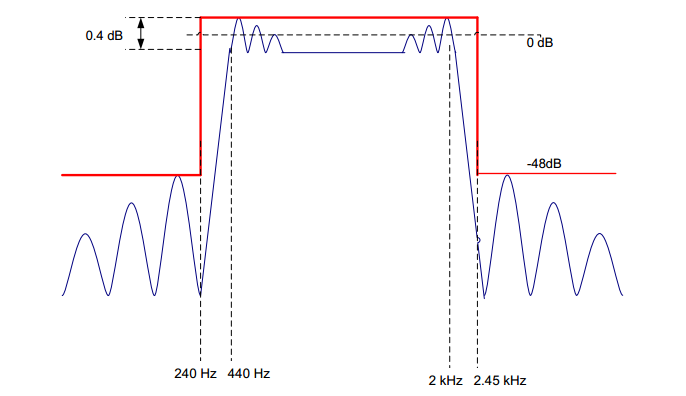
Operation Principles

Non-circular FIR filter design

Circular FIR filter design

Frequency response analysis

**MATLAB filter design**

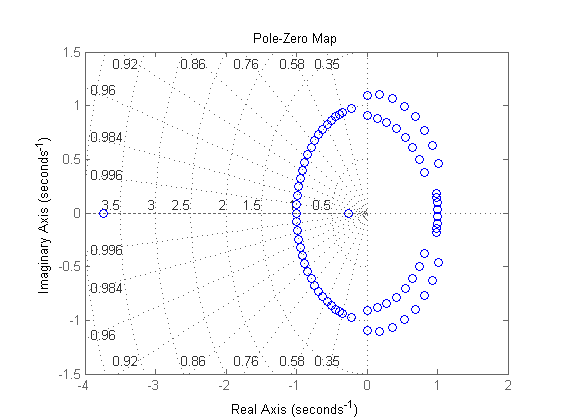
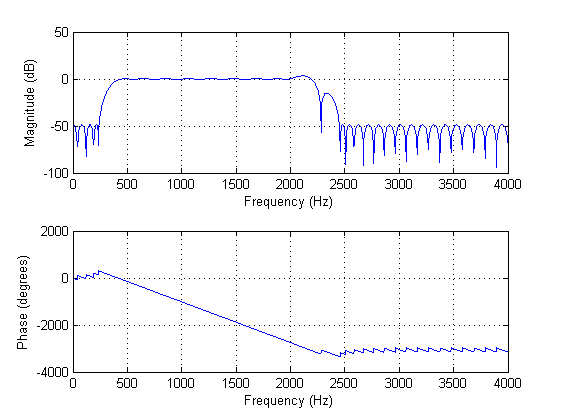
****The FIR filter required to build in MATLAB has specifications shown in Figure 1.

**Figure 1.** FIR specifications

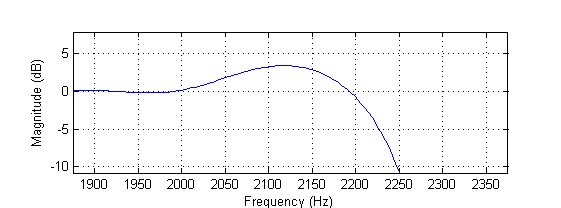
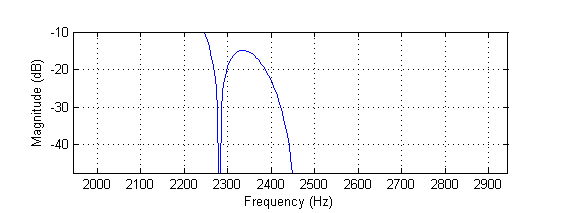
In MATLAB, the Parks-McClelland algorithm is suggested to approximate required filters using *firpmord* and *firpm* functions. Code shown as the following:

**Figure 2.** MATLAB filter design code

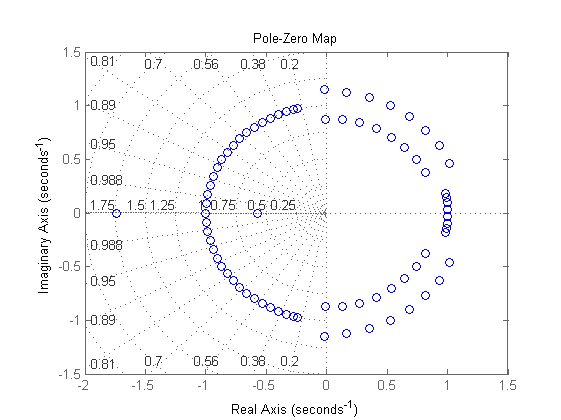
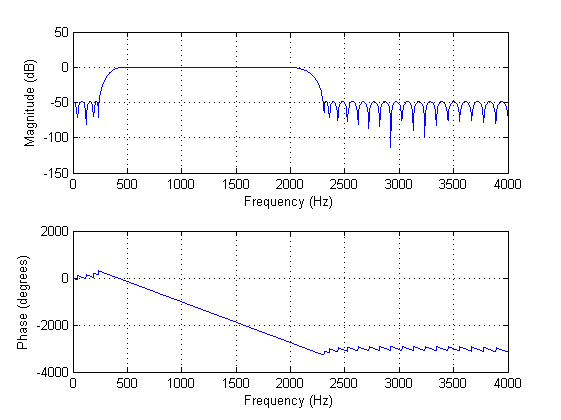
|  |
| --- |
| clear all;    f1=240;f2=440;f3=2000;f4=2450;% frequency boundary  f=[f1,f2,f3,f4];%make array of frequencies    rp=0.4;% ripple in dB  sa=48;% minimum stop band attenuation in dB  dev1=(10^(rp/20)-1)/(10^(rp/20)+1);%calculate pass band deviation  dev2=10^(-sa/20);%calculate stop band deviation  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); |

Frequency and phase response as well as pole map are included in Figure 3, from which we can see that specifications are 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**

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**

**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"*

**Non-circular FIR filter design**

**Code explanation**

|  |
| --- |
| **#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**

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

**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. Note 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.

**Naïve Implementation**

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--;  } |

**Code Efficiency and comparison**

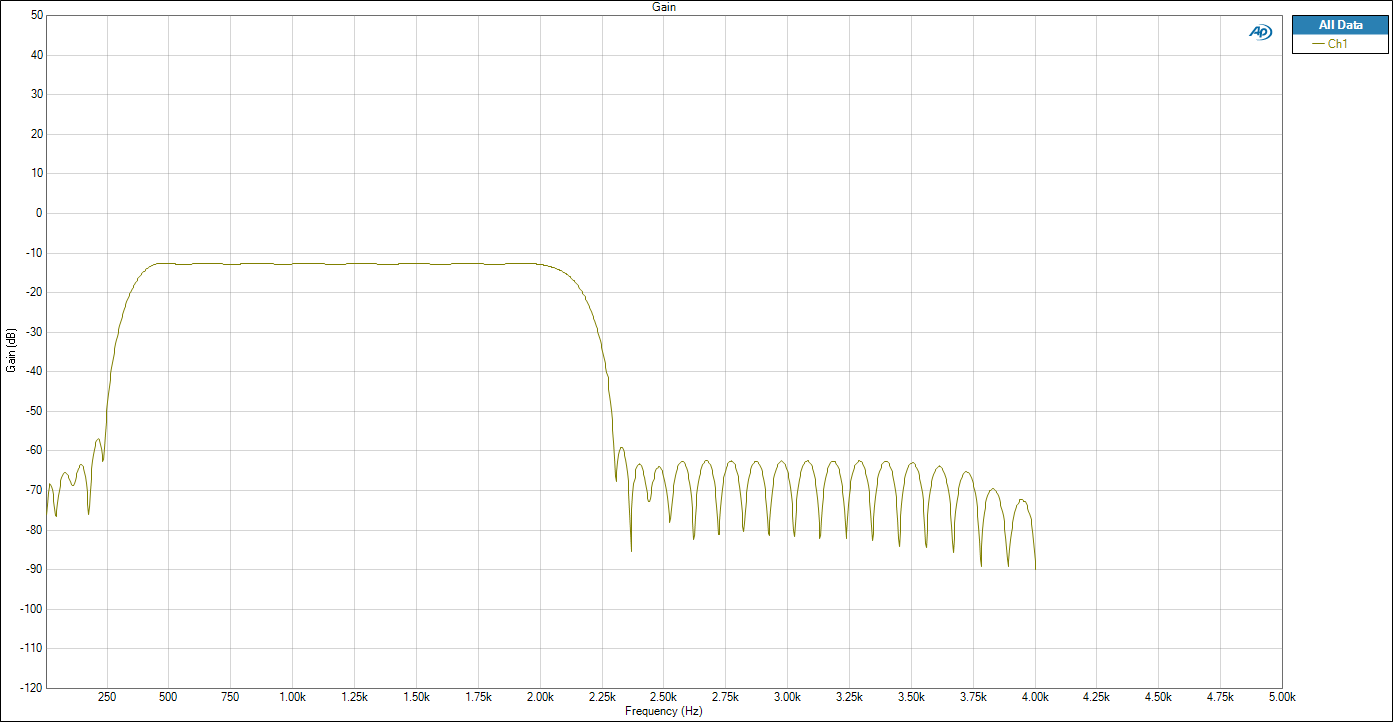
Code performance with and without compiler optimisation is measured as the following:

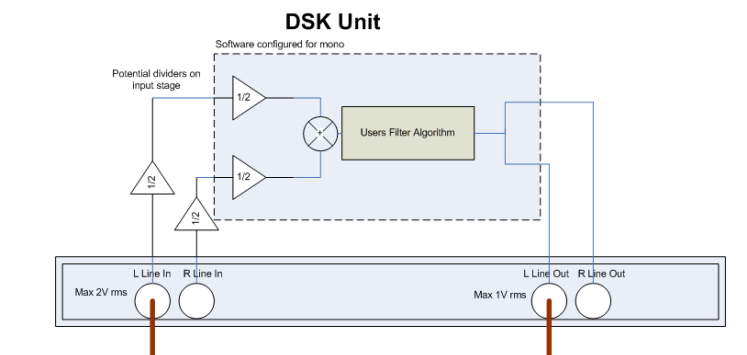
**Table 2.** Optimisation level and clock cycle

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **--opt\_level** | **Non-circular** | **Basic Circular** | **Circular using linear phase** | **Circular with linear phase and double input array** |
| no | 5145 | 5640 | 3737 | 2826 |
| 0 | 4208 | 3756 | 2432 | 1729 |
| 2 | 1024 | 623 | 403 | 313 |

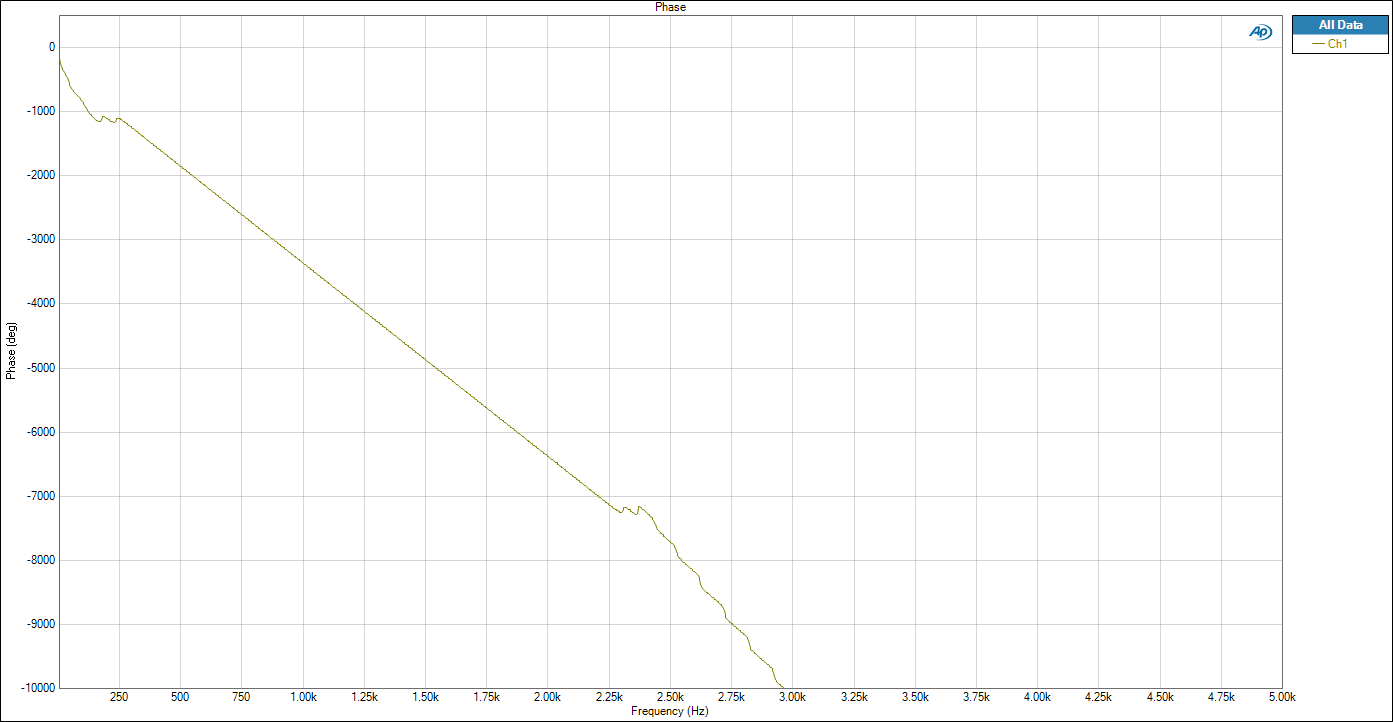
Table 2 shows that the code efficiency is improved significantly as optimisation level increases. Compare to non-circular buffer version, circular buffer without optimisation performs worse than non-circular buffer. This is due to the additional if-statement and pointer increment inside the for-loop. However, this process is further optimised by compiler and after performance optimisation, clock cycle use by circular buffer is half of that of non-circular buffer.

**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.