Jurgen Shiqerukaj - CID 00938154 Jason Yuan – CID 00954645 Due 24th February 2017

Lab 4 – RT Implementation of FIR Filters

Contents

Overview

In this lab session, we will learn how to design and implement FIR filters. We will take an input signal from a signal generator, pass it through a band pass filter and retrieve desired response based on given specifications.

The Problem

Our ultimate aim in this lab is to implement a band pass FIR filter that satisfies the following specifications.

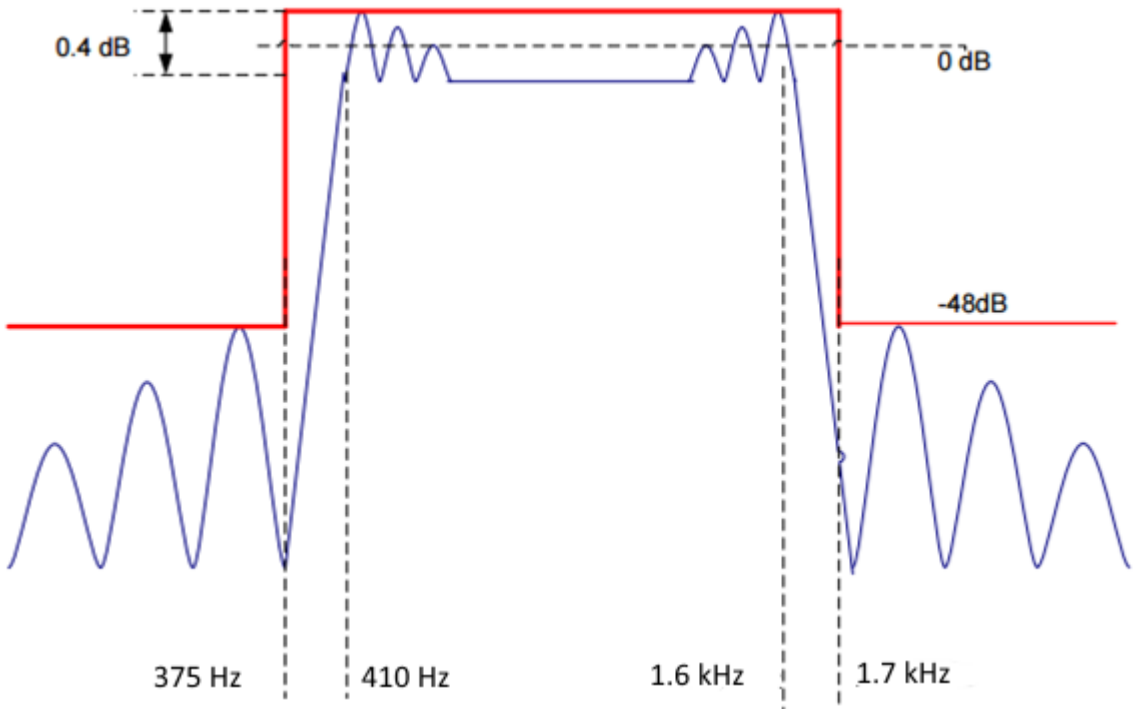


Figure 1: Image shows the required specifications required for this lab. [1]

How a FIR Filter Works.

To implement a FIR filter we have to familiarize ourselves with convolution and how it works.

Mathematically convolution has the following operation:

$$Y[n] = x[n] * y[n] = \sum_{k=-\infty}^{\infty} x[k]\, b[n-k]$$

$$Y[n] = b_0 x[n] + b_1 x[n-1] + \cdots + b_M x[n-M]$$

$$H[z] = b_0 + \frac{b_1}{z} + \cdots + \frac{b_M}{z^M} = \sum_{k=0}^{M} b[k] z^{-k}$$

- *Where k is a dummy variable which causes a shift and flip*
- *B is the system impulse response*
- *Y is the filtered output*

Graphically we can intuitively see how by convolving with a system impulse response we can get a desired filter effect. Here we use an example of how a high pass filter is implemented.

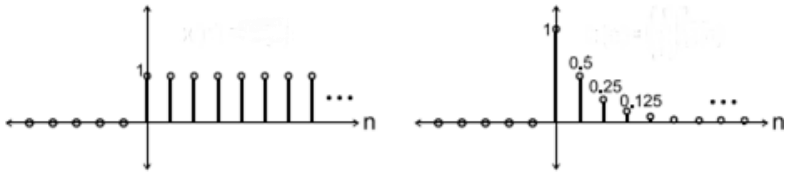


Figure 2: Image shows input X[n] on the left and impulse response B[n] on the right. [1]

When we convolve we have to flip the impulse response along the y axis.

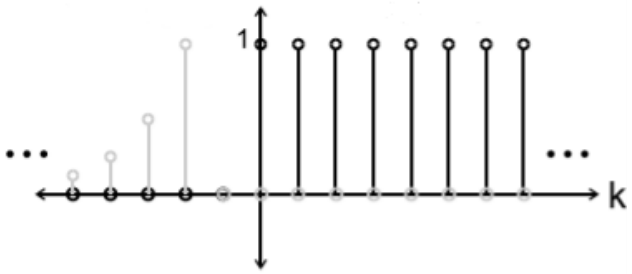


Figure 3: Image shows input X[n] and flipped impulse response B[n] Getting ready for the next stage. [2]

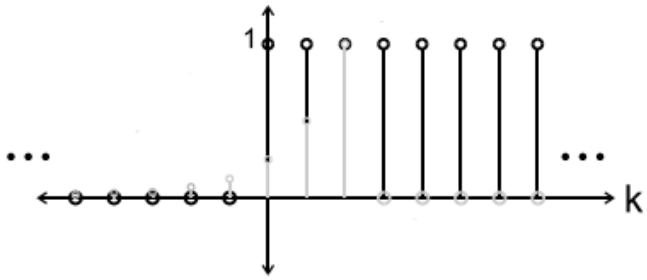


Figure 4: Image shows the flipped impulse response B[n] sliding across input X[n] whilst multiplication occurs. [2]

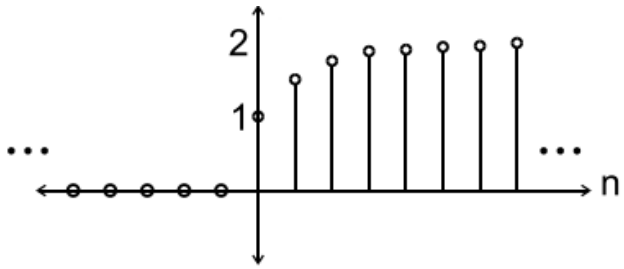


Figure 5: Image shows output Y[n] as the result of convolution, we can see sample values getting attenuated. [2]

Graphically we have seen how convolution is used to manipulate the original signal. We will use this to implement a band pass filter. The key to implementing such a band pass filter we have to get the correct impulse response coefficients.

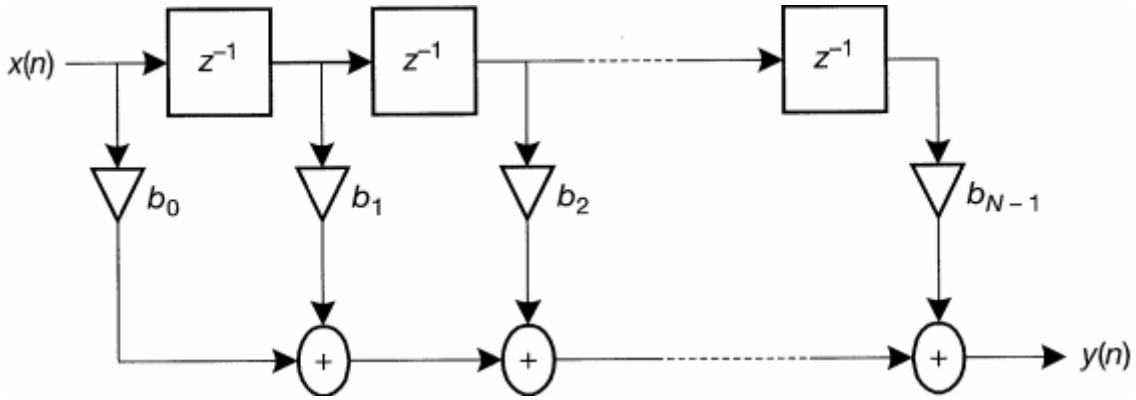


Figure 6: Image shows implement delay line so we can store all samples in an array. [1]

It is also very important we use a delay line by shifting in a new sample at the start of the array, when a new sample comes in, you lift to the left by one with the last element of the array being the oldest sample.

New Sample = 1 Oldest Sample = 8

1	2	1	5	8

New Sample = 9 Oldest Sample = 5

9	1	2	1	5

There are essentially 3 main stages to implementing a FIR filter.

- Calculating the required impulse response and obtain coefficients through MATLAB.
- Multiply coefficients and input samples
- Shift the stored inputs.

Using MATLAB

The first steps to designing the desired filter is to use MATLAB to get the coefficients for our filter. To do this we will be using Parks-McClelland algorithm which is an iterative algorithm used to finding the most optimal FIR filter. It is designed to minimise errors in the pass and stop bands. For this lab we will be using this algorithm as a black box by only feeding it inputs and receiving outputs.

```
rp = 0.4;            % Passband ripple
rs = 46;           % Stopband ripple
fs = 8000;         % Sampling frequency
f = [375 410 1655 1700];    % Cutoff frequencies
a = [0 1 0];         % Desired amplitudes

% Convert the deviations to linear units. Design the filter and visualize
% its magnitude and phase responses.

dev = [10^(-rs/20) (10^(rp/20)-1)/(10^(rp/20)+1)  10^(-rs/20)]; %dev is a
vector of the same size as 'a' that specifies the maximum allowable
deviation or ripples

[n,fo,ao,w] = firpmord(f,a,dev,fs); % finds the approximate order,
                %normalized frequency band edges, frequency band
amplitudes,
                %and weights that meet input specifications f, a, and dev
                %where fs is sampling frequency.

b = firpm(n,fo,ao,w); % returns row vector b containing the n+1
coefficients of the order n FIR filter whose frequency-amplitude
characteristics match those given by vectors f and a.

freqz(b,1,1024,fs) % Finds frequency response of digital filter and plots
graph.

title('Frequency Response') % Set title to graph

ss=['double b[]={']; % Initialise string starting with 'double b[]={'

for i = 1:429  % For loop used to cycle through all co-efficients.

    ss = [ss  num2str(b(i)) ', ' ]  %Add co-efficients to the string
                                    %followed by a comma.
end
ss=[ss '}']; %finish the string off with a curly bracket.
```

'ss' here is a string of characters. We implemented this so we can easily input the co-efficient in a text format for our implementation of the band pass filter in CCS.

Here are the plots we received with above code:

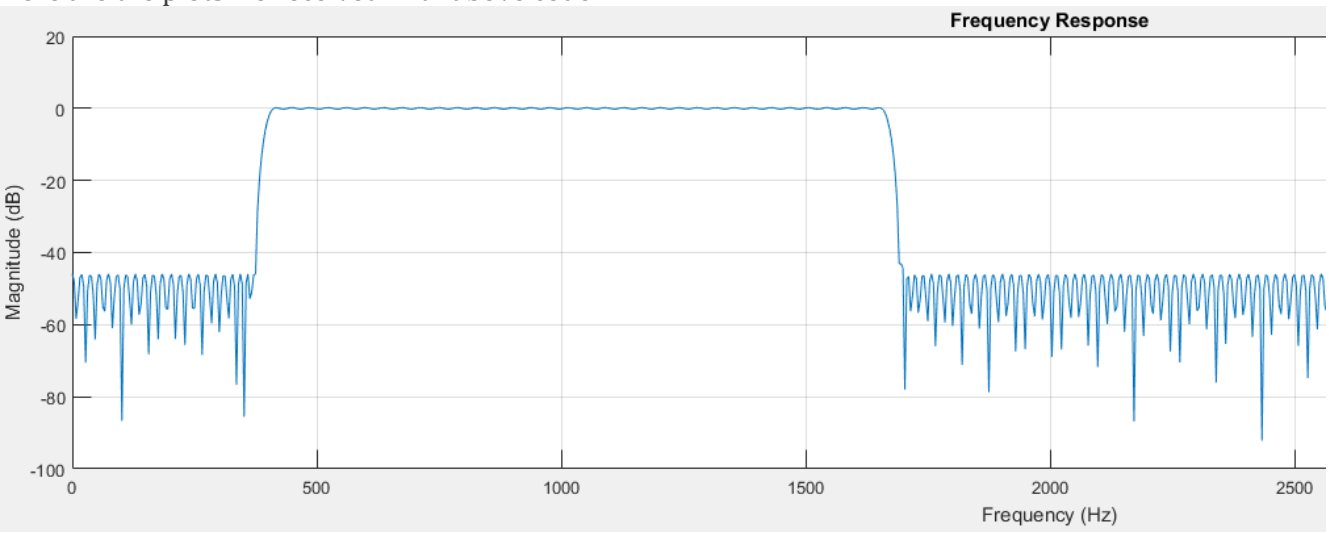


Figure 7: Image shows a snippet of the resultant frequency response, it shows a band pass filter and 375 Hz and 1.7 kHz.

To showcase that it also obeys by the other specifications, close ups have been taken.

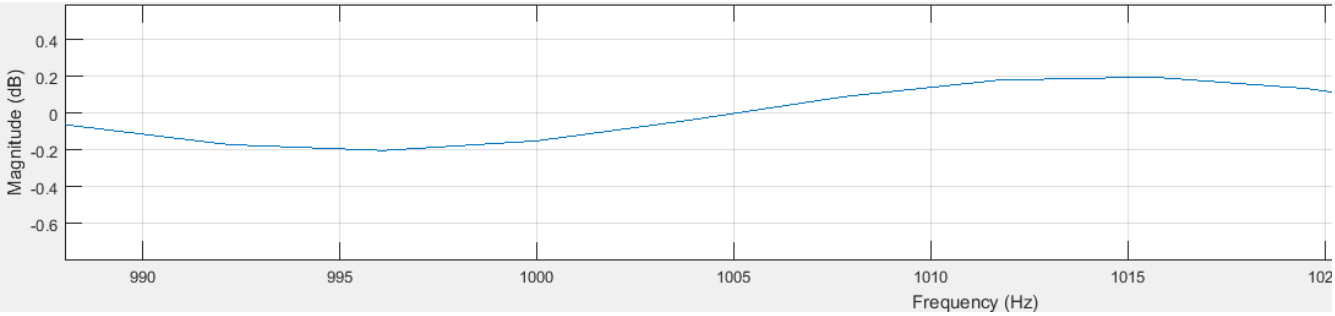


Figure 8: Image shows a zoom in on the pass band filter near 1000 Hz. We can clearly see that it meets the specification of 0.4 dB ripple.

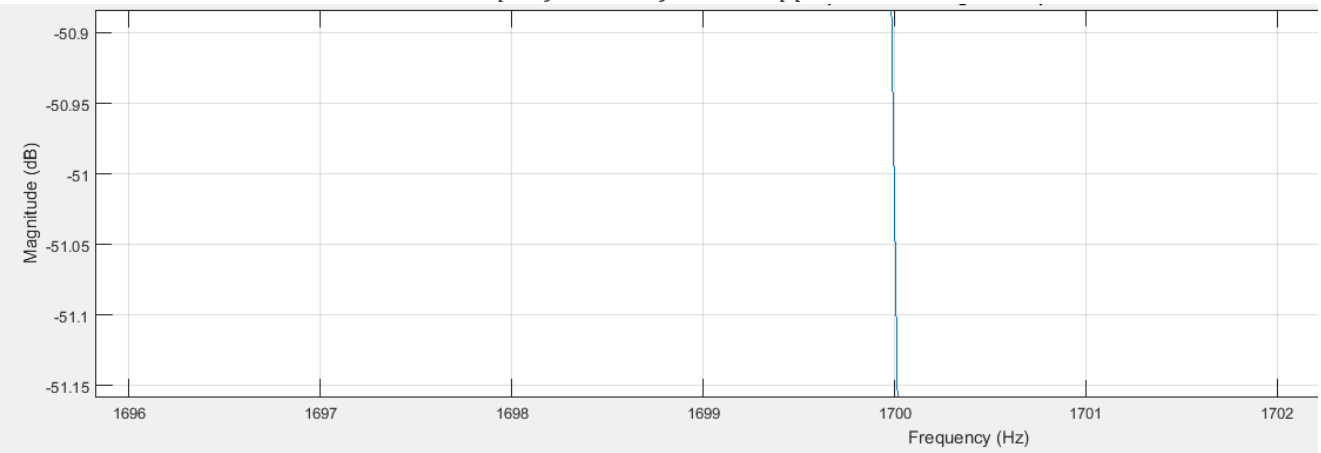


Figure 7: Image shows a zoom in on the pass band filter on 1700 Hz. We can clearly see that the cut off is well below the -46 dB mark.

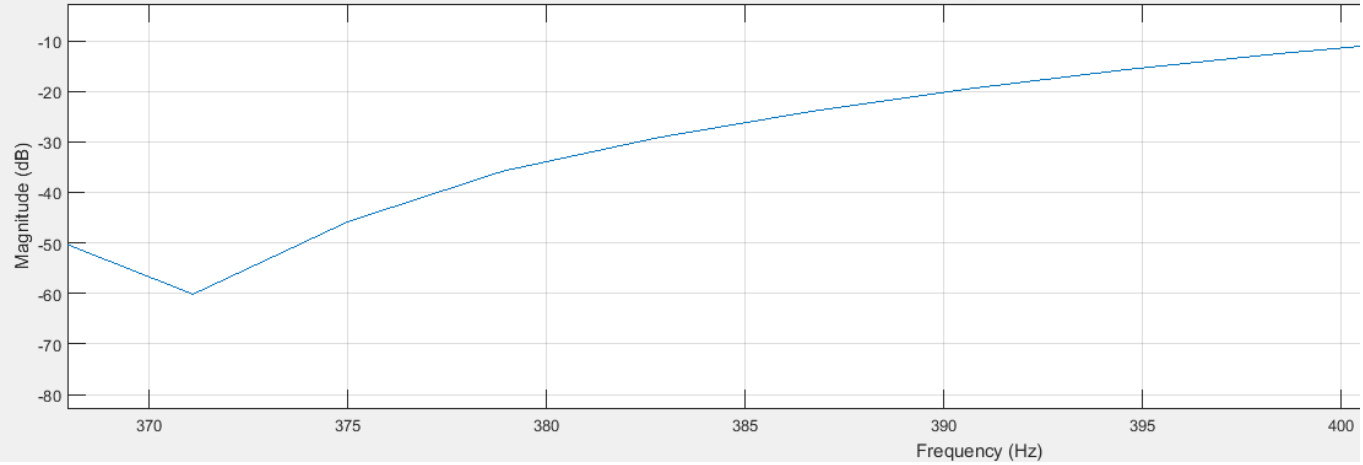


Figure 9: Image shows a zoom in on the pass band filter on 375 Hz. We can clearly see that the cut off is just below the -46 dB mark and within specifications.

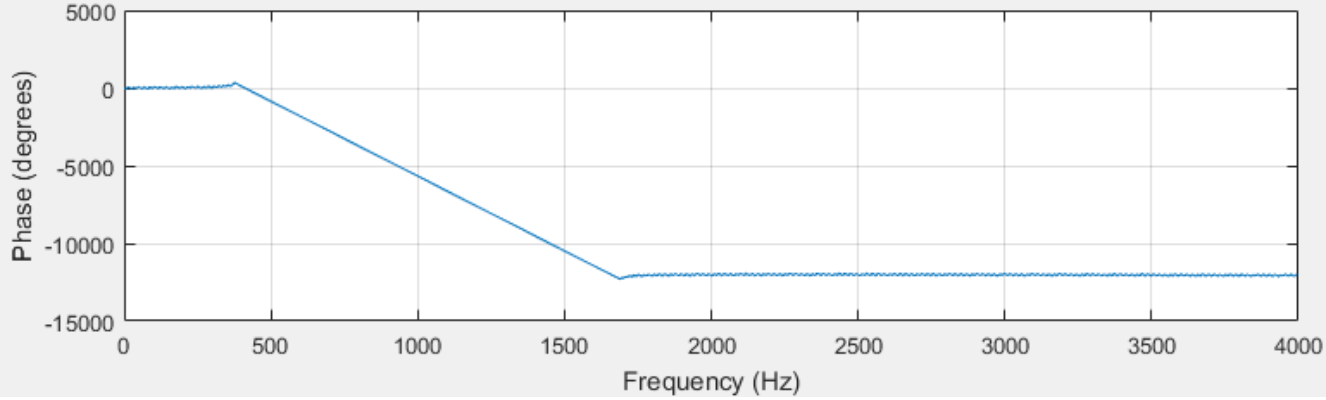


Figure 10: Image shows the phase response of the Band Pass filter, it clearly shows it is linear phase and hence why the coefficients are symmetric.

The Non-circular FIR Filter

The non-circular FIR filter works in a very clunky way, but it is very intuitive to understand. It involves shifting data such that the first sample in the array is always the newest sample and the last sample in the array is always the oldest sample. This results in a simple and intuitive process of multiplying with the array of coefficients. However there are several disadvantages to this method, one reason is that a lot of instructions are used to copy data which could possibly slow the FIR filter for large order filters.

Non-Circular Non-Optimised FIR Filter

This method revolves around multiplying 9 samples at a time with the coefficients. The code operates by first reading a sample and storing it into a short variable. We then go into a for loop. This for loop cycles through N/9 as we are jumping by 9 samples at a time, and within the For loop, we do the multiplication for the first 9 sample, then the next set of 9 and so on. It is important to note that not every N is divisible by 9 hence we have to find out how many elements are left to convolve. This is implemented by an If statement that checks if N is divisible by 9, if it isn't then we find the remainder which directly correlates

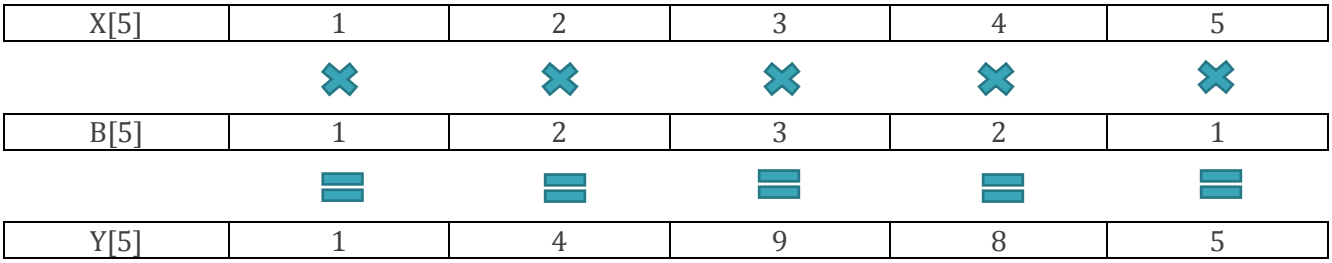
to the number of elements near the end of the array where we still have to convolve. We then perform the iterative push to the right of an input sample.

Through doing this exercise we experimented with doing a different number of elements at the same time. We found that after doing 16 elements at the same time, the FIR filter will not work. This is because we exceed the amount of memory the program can handle and there was very little improvement from how many cycles it save. Hence we stuck with a near midpoint value of 9.

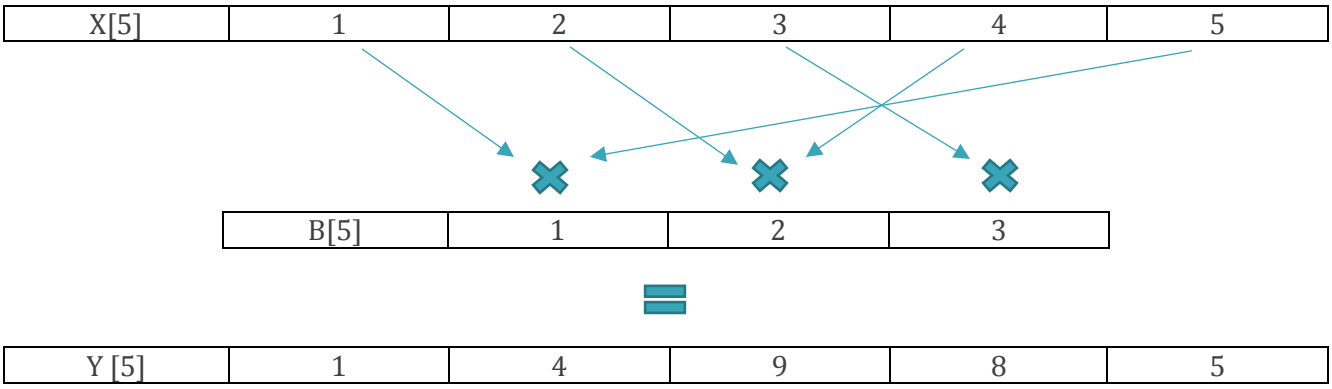
```
void non_circ_FIR_NON_OPTIMISED (void) // Multiple of 9
{
    short read_sample = mono_read_16Bit();  /*Takes a sample reading and places it in
                                             a variable of type float for later
manipulation.*/

    int j;                                  //Counter
    int indexjumper;                        //Counter
    output = 0;                             //Set output of type double to 0 initially.

    for (i = 0; i < N / 9; i++)
    {
        indexjumper = 9 * i;
output += x[indexjumper] * b[indexjumper] + x[indexjumper + 1] * b[indexjumper + 1] +
x[indexjumper + 2] * b[indexjumper + 2] + x[indexjumper + 3] * b[indexjumper + 3] +
x[indexjumper + 4] * b[indexjumper + 4] + x[indexjumper + 5] * b[indexjumper + 5] +
x[indexjumper + 6] * b[indexjumper + 6] + x[indexjumper + 7] * b[indexjumper + 7] +
x[indexjumper + 8] * b[indexjumper + 8];

        /* performs the Convolution, this method performs the multiplication for 9
samples at a time.*/

    }

    if (N % 9 != 0)
    {
        for (j = 0; j < N % 9; j++)
        {
            output += b[N - 1 - j] * x[N - 1 - j];   /* Not everything is divisible by
9, hence we find the remainder so do the last set of elements in the array   */
        }
    }

    for (i = N - 1; i > 0; i--) --)                /* For loop cycles through the
                                                      entire coefficient range*/

    {
        x[i] = x[i - 1];                           /*Performs shift of data, pushing
data through from the left.*/
    }
    x[0] = read_sample;                            /* Add data to first element of
array that will later be push along array.*/


    mono_write_16Bit(output);                      /* Output sample with respect to
pass band filter attenuation.*/

}
```

Non-Circular Optimised FIR Filter

To get a more optimised non-circular FIR filter we used a property about the coefficients. The coefficients retrieved are real and symmetric indicating we have a linear phase filter. Initially, we have been performing the convolution as follows.



However, we noticed that as the coefficients are symmetric we don't need to look up the second half of the coefficient array. It also means we can do the first and last term in the input array simultaneously.



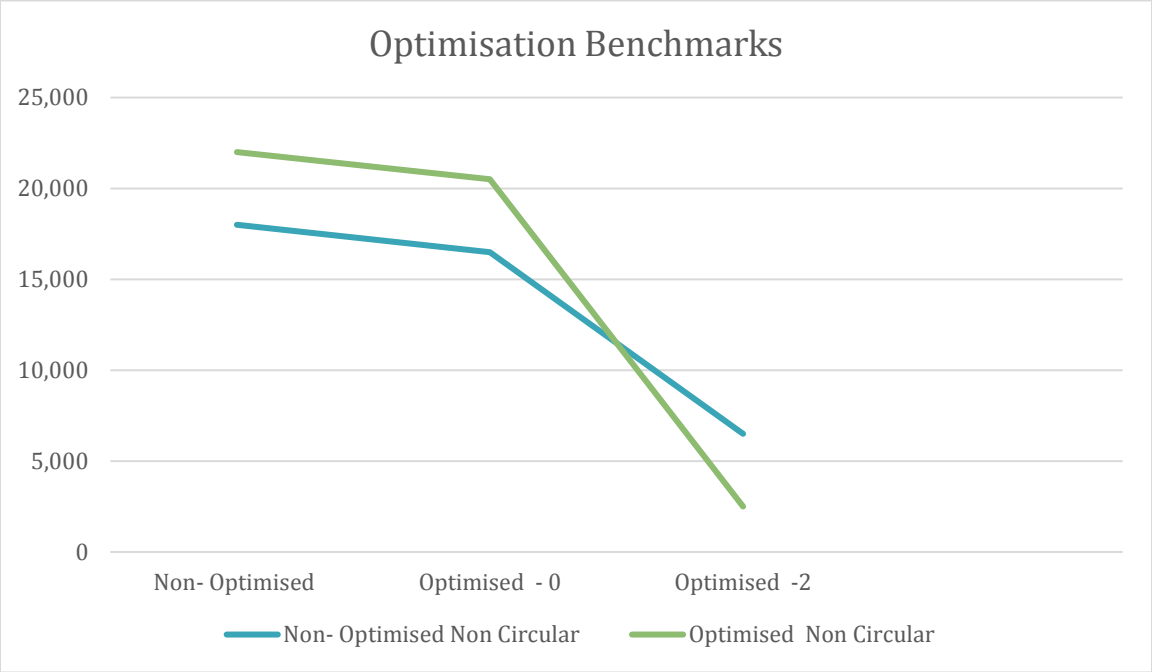
This same concept can be implemented in Code Composer. We first read a sample and store it in a variable of type short. As we are using the property that the coefficient array is symmetric we only need to cycle through half of the array. Hence we can make our for loop half the size of N. Within the for loop we perform convolution, we take the first coefficient and multiply it to both the first and last element of X[N], to the second coefficient is then taken and is multiplied to the second element and second to last element of X[N] and so forth. Issues arise when N is odd, in which case the for loop will not do the convolution of the mid-point. To compensate for this an If condition is used, `if ((N - 1) % 2 == 0)` , This checks if N is odd or even, if it is odd it will go through and perform the midpoint calculation otherwise it has already been performed within the For loop. We then perform the iterative push to the right of an input sample.

```
void non_circ_FIR_V3(void)
{
    short read_sample = mono_read_16Bit(); /*Takes a sample reading and places it in
                                            a variable of type float for later
manipulation.*/
    output = 0;                          //Set output of type double to 0 initially.

    for (i = 0; i < (N - 1) / 2; i++)   // For loop starts using only half of N
    {
        output += b[i] * (x[i] + x[N - 1 - i]); /*Performs convolution , abuses
                                                 That fact that co-efficients are
                                                 Symmetric about the middle and
                                                 allows use to convolve the first
                                                 and last term of X[] at the same
                                                 time. This saves allot of
                                                 cycles and speed.*/
    }
    if ((N - 1) % 2 == 0)               //If statement checks if N is odd or even.
    {                                   /*If it is indeed odd then we need ensure
                                        that the mid point is indeed convolved.*/

        output += x[(N - 1) / 2] * b[(N - 1) / 2]; /* Performs the convolution of the
                                                      mid point.*/
    }

    for (i = N - 1; i > 0; i--)        /* For loop cycles through the
                                          entire coefficient range*/
    {
        x[i] = x[i - 1];               /*Performs shift of data, pushing data
                                         through from the left.*/
    }
    x[0] = read_sample;                /* Add data to first element of array that
                                          will later be push along array.*/

    mono_write_16Bit(output);         /* Output sample with respect to pass band
                                         filter attenuation.*/
}
```

Summary and Trace Scopes

After performing some of the benchmarks, we notice that our non-optimised con-circular FIR filter uses less cycles for no Optimisation level and Optimisation level, but we sacrifice memory as discussed previously. Whilst the Optimised Non-Circular FIR filter has significantly less cycles at Optimisation – 2.

No. Cycles	No Optimisation	Optimisation - 0	Optimisation - 2
Non- Optimised Non Circular FIR filter	**18,000**	**16,500**	**6,500**
Optimised Non Circular FIR filter	**22,000**	**20,599**	**2,500**



The reason for this is because the Optimised Non-Circular FIR filter uses far less instruction cycles due to using the symmetric property of the coefficients array. Whilst the Non-Optimised Non-Circular FIR filter has to look up each and every coefficient thus uses more instructions.

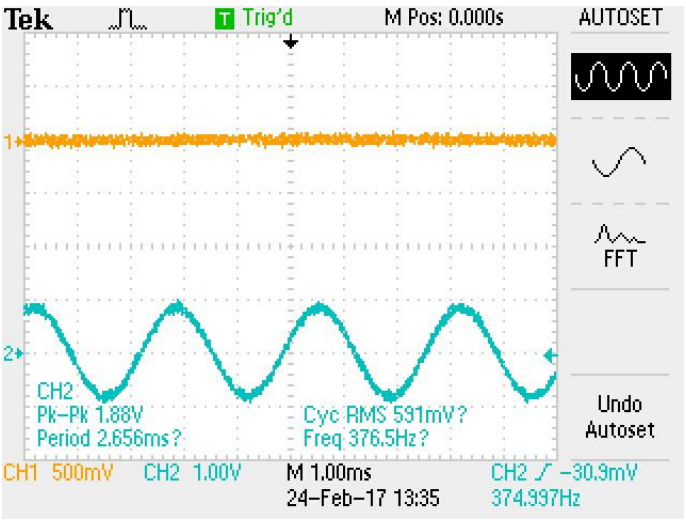


Figure 11: Image shows Non-Circular FIR filter with 375 Hz input. This is in the stop band of the filter and should result with a zero output as shown.

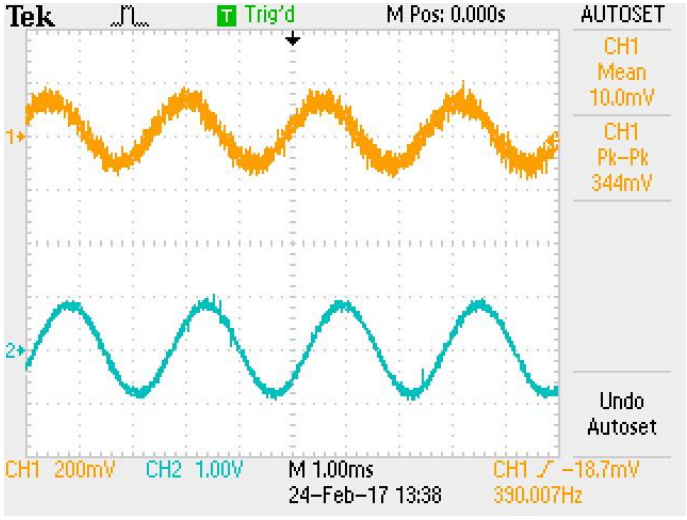


Figure 12: Image shows Non-Circular FIR filter with 390 Hz input. At this frequency we are approaching the cutoff and hence we should expect some attenuation at the output which we see here.

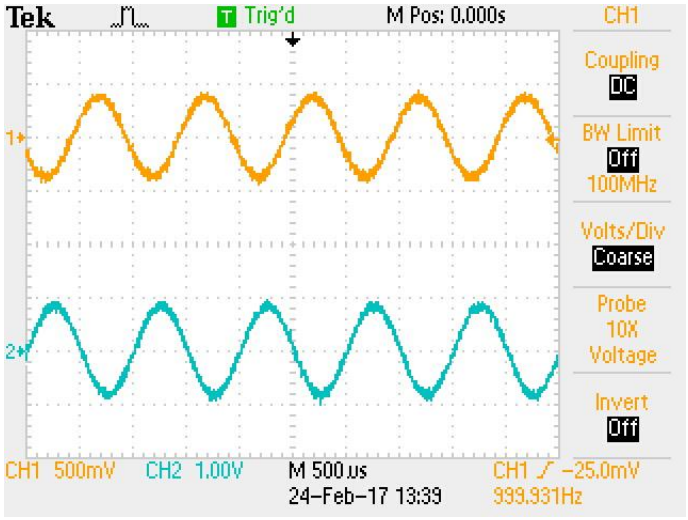


Figure 13: Image shows Non-Circular FIR filter with 1000 Hz input. At this frequency we are in the middle of the band pass filter and shouldn't expect attenuation. The reason why output is half is because of the potential divider at the input of the audio chip, halving the input. [Appendix]

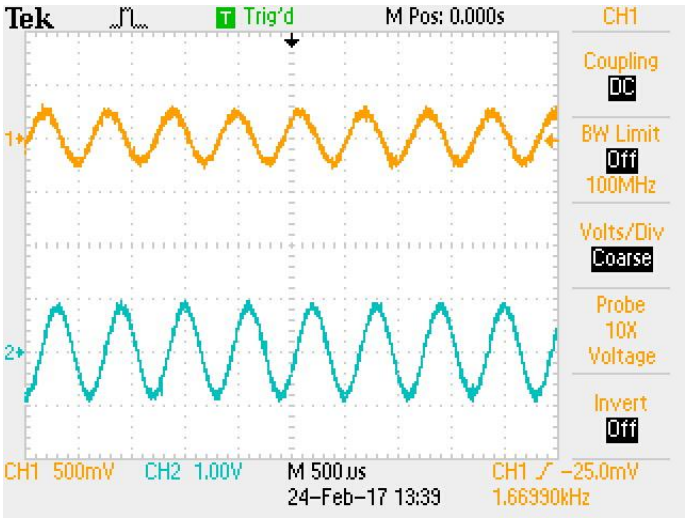


Figure 14: Image shows Non-Circular FIR filter with 1670 Hz input. At this frequency we are approaching the cutoff and hence we should expect some attenuation at the output which we see here.

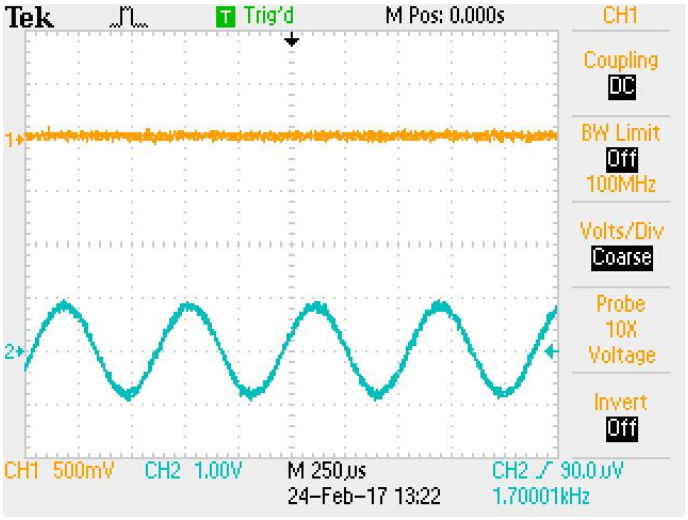


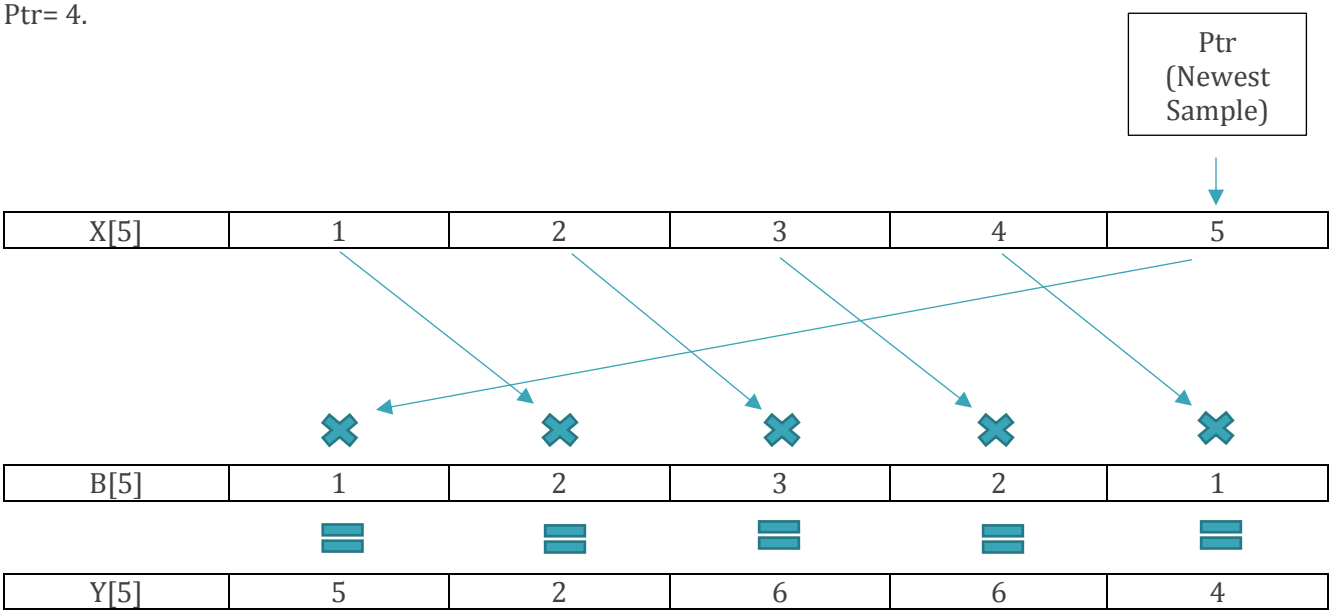
Figure 15: Image shows Non-Circular FIR filter with 1700 Hz input. This is in the stop band of the filter and should result with a zero output as shown.

The Circular FIR Filter

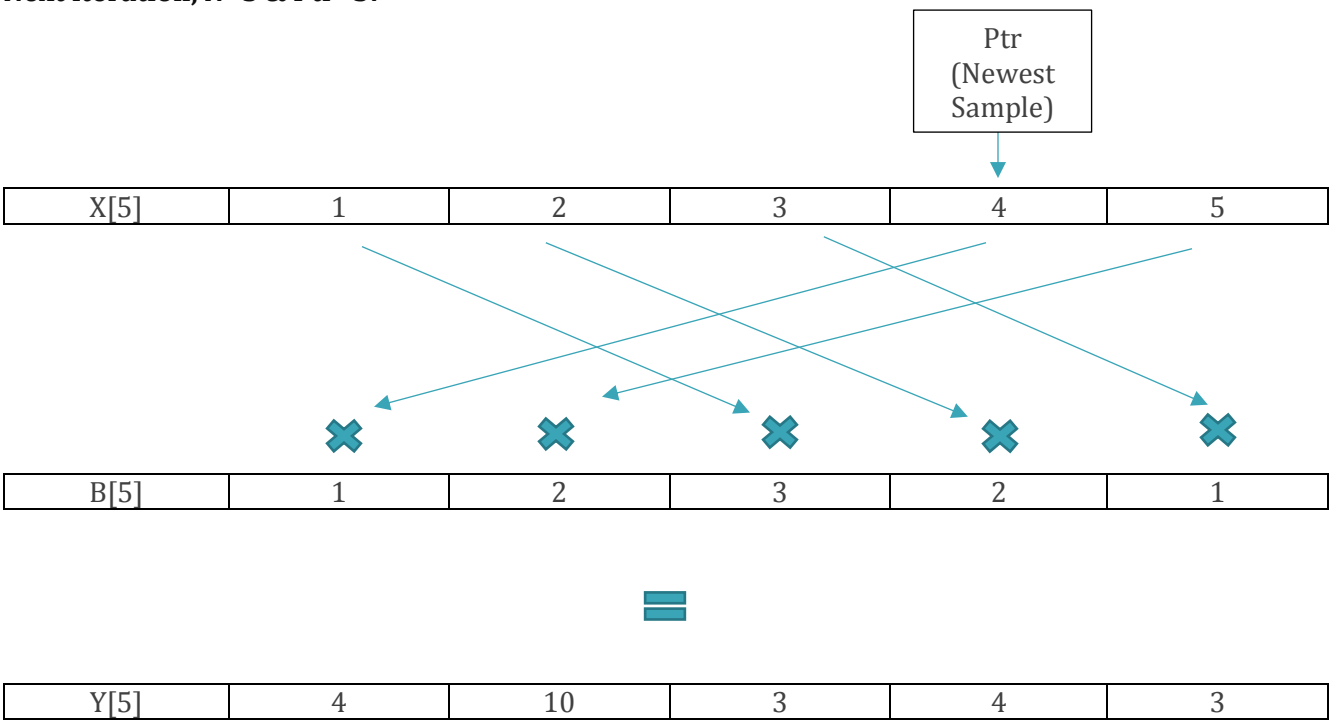
We noticed that the Non-Circular FIR Filter is clunky with its main disadvantages being that a lot of instructions are used to copy data which can slow down the FIR filter as more cycles are required. A Circular Buffer tries to eliminate this issue by having a pointer that moves along the array.

Circular FIR Filter Attempt 1

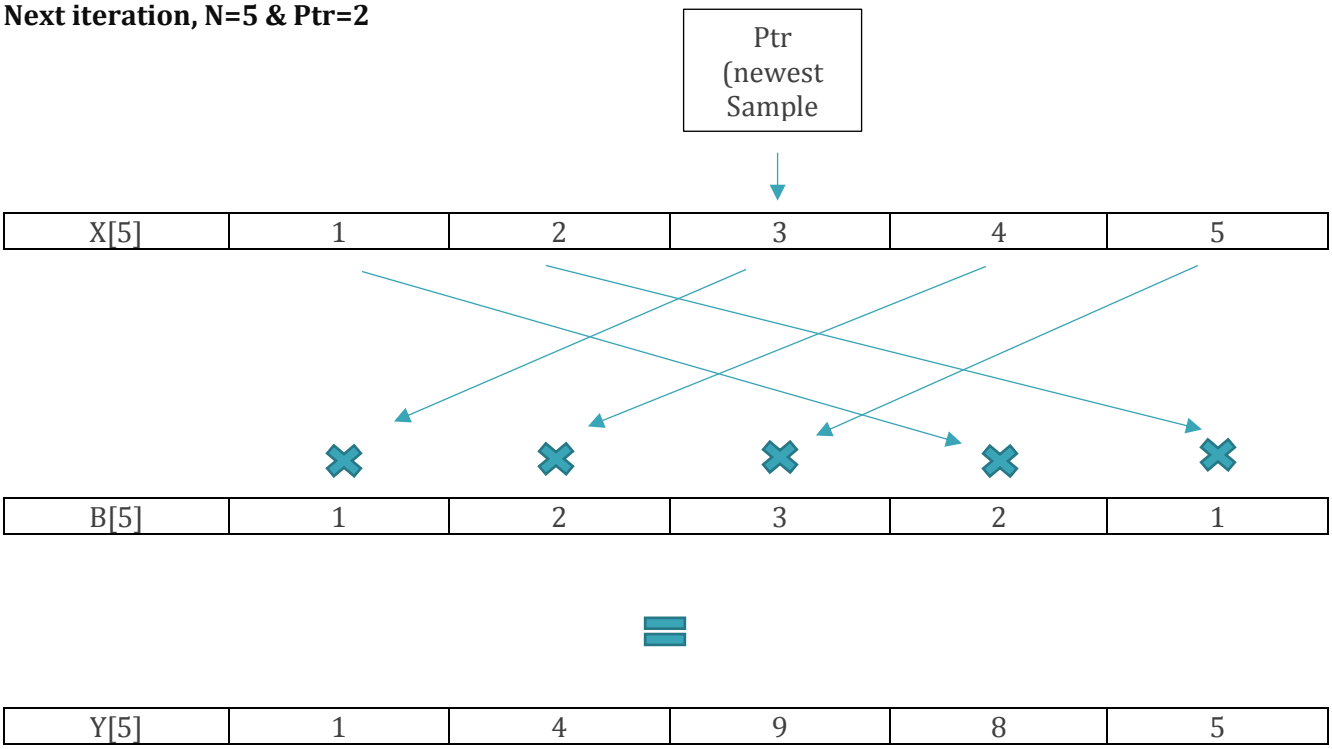
This is our first and one our least optimised attempt as will be seen later on in the summary. With a Circular buffer we are not copying data but instead moving a pointer to indicate the newest sample. To understand how this implements the circular buffer, we need to consider a smaller case with N =5 and Ptr= 4.



Next iteration, N=5 & Ptr=3:



Next iteration, N=5 & Ptr=2



This visual representation of a circular buffer demonstrates how it works. The pointer points to the newest sample with the sample to the left of it being the oldest sample and this shifts accordingly with changing new samples and old samples. The convolution is then calculated.

We implemented this by first taking a sample reading and storing it in `short read_sample.` We then enter the first for loop `for (i = ptr; i < N; i++)` which cycles from the pointer `ptr` and end of the buffer. This can be visually seen from above. Within this `for` loop we do the convolution with the first coefficients.

We then enter the next for loop `for (i = 0; i < ptr; i++)` which wraps a cycles through the start of buffer to the pointer which represents the older samples. Within the `for loop` we perform the convolution by multiplying these samples with later coefficients. It's important to note we haven't used the fact that the coefficients are symmetric in this attempt.

To avoid the `ptr` overflowing we have to include an if condition that checks if pointer = 0 and the wraps it back round to the end of the buffer to start the process again otherwise the pointer decrements.

We then store sample at pointer location and write out the output.

```
void circ_FIR(void)
{
    short read_sample = mono_read_16Bit(); // read_sample holds the input sample.
    output = 0; // reset output to 0 after everytime the interupt function is called.

    for (i = ptr; i < N; i++) /* Performing convolution upwards from ptr
        to the end of the buffer.*/
    {
        output += x[i] * b[i - ptr];
    }

    for (i = 0; i < ptr; i++) // Perform convolution upwards from 0 to ptr
    {
        output += x[i] * b[N - ptr + i];
    }


    if (ptr == 0) // Check if ptr overflows
    {
        ptr = N - 1;
    }
    else
    {
        ptr--; /* Decrement ptr to allow the current sample to be
                stored correctly.*/
    }
    x[ptr] = read_sample;

    mono_write_16Bit(output); // Write output to the audio port of DSP.
}
```

Circular FIR Filter Attempt 2

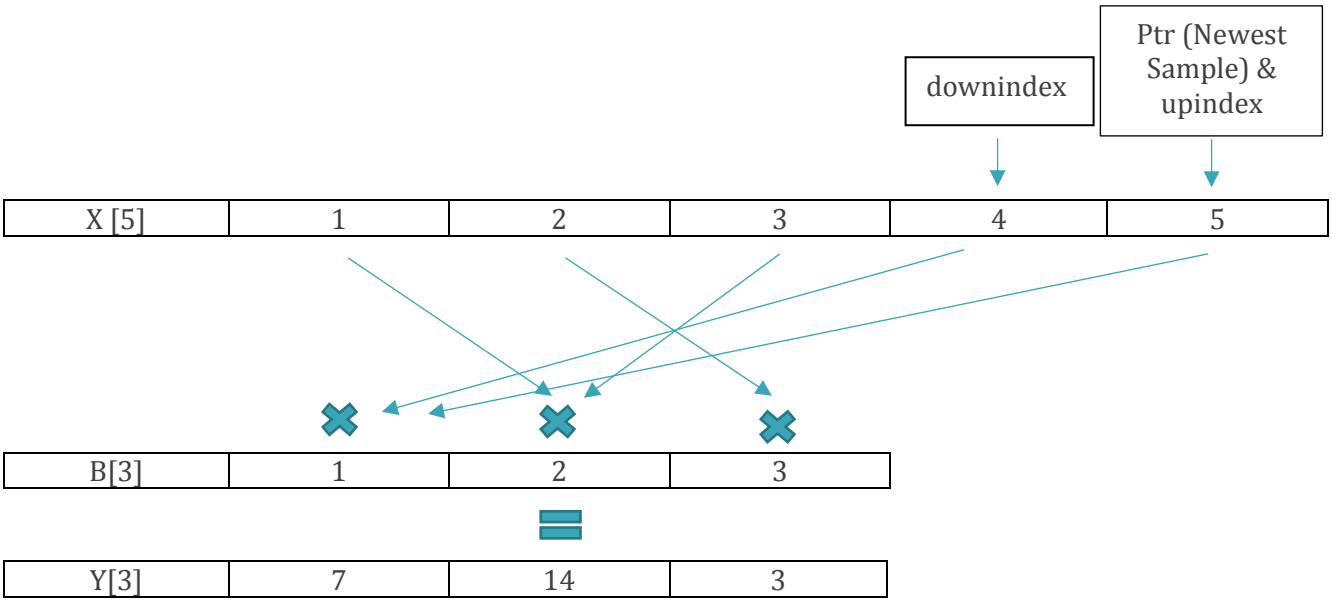
With the previous attempt on circular buffer we did not use the symmetric property of coefficients, in this attempt we will use the symmetric property. To do so we added a few extra variables:

Bindex – used to keep track of the indexing of the b coefficient array.

Upindex – used to keep track of the indexing of the most recent sample.

Downindex – used to keep track of the indexing of the oldest sample.

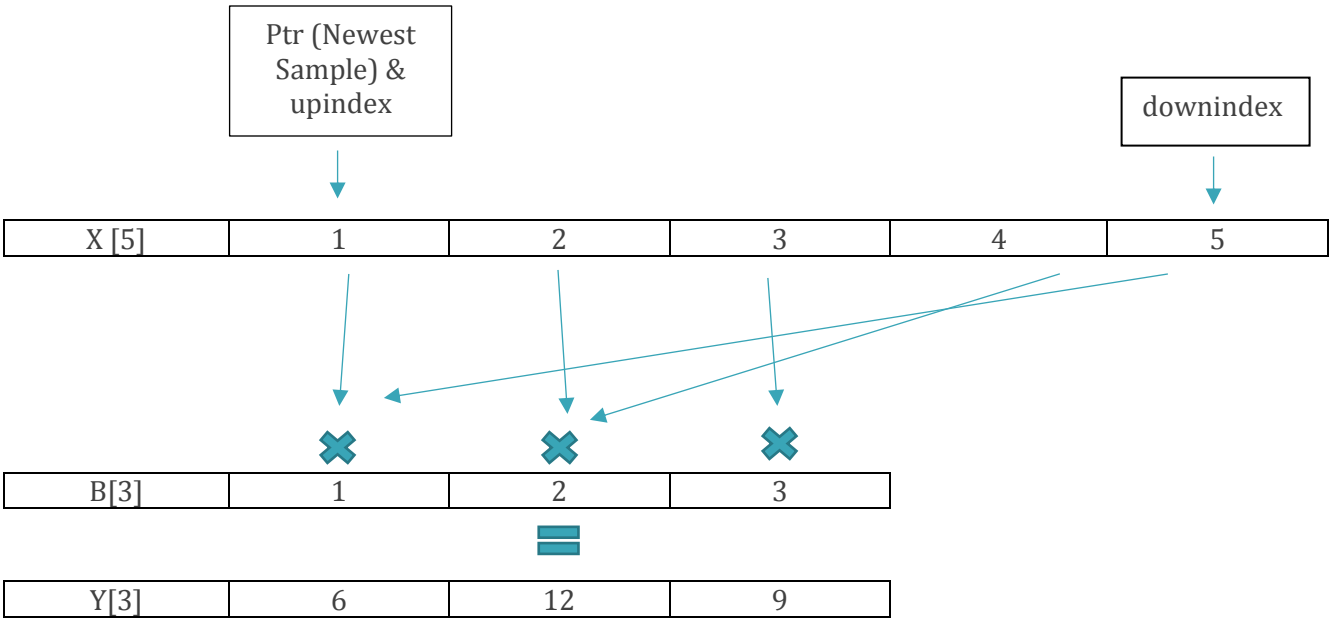
To gain better understanding on how these variable are used we will consider a basic case in which N = 5 and ptr = 4.



This visual representation illustrates what happens when the variable ptr $\geq \frac{N}{2}$. The pointer downindex points towards the oldest sample, whereas upindex points towards the newest sample. This is done via upindex = ptr and the if statement for ptr == 0 which is used to ensure downindex does not overflow. The convolution is then calculated like the diagram above.

We first store the input sample into read_sample and reset the output and bindex to 0. The 1st while loop convolutes all the samples from ptr -> N-1 and their corresponding counterpart by increment/decrement the upindex and downindex respectively also bindex is incremented to keep track of the number of factorized convolution done once this is done the upindex is reset to 0 as the indexing overflows. The 2nd while loop then convolutes all the remaining samples from 0 –> ptr – (N-ptr) by using the condition bindex $< \frac{N}{2}$, to ensure we do the correct number of factorized convolution ($\frac{N}{2}$ *or* $\frac{N}{2} + 1$ depending if N is odd or even) whilst still using the symmetric property. In this example above since N is odd we need to as the while loops will miss this sample thus we need calculate the midpoint of this circular buffer by using the equation as follows.

$$Midpoint = \left(\frac{(N-1)}{2} + ptr\right) \%N$$



This visual representation illustrates what happens when the variable ptr $< \frac{N}{2}$. In this example upindex and ptr = 0 hence the if statement causes downindex to now point at N-1 to avoid overflow errors. The convolution for this iteration is then calculated as above.

Similar to before we store the input sample into read_sample and reset the output and bindex to 0. The 1st while loop works similarly to 1st while loop in the previous case but with a slight adjustment since downindex resets to N-1 when ptr = 0, the downindex >= 0 is not sufficient enough as it would cause too many factorized convolution thus we added an additional condition which is bindex $< \frac{N}{2}$ to ensure the answer is correct.

```
void circ_FIR_V1(void)
{
    int upindex; // index for the recent sample.
    int downindex; // index for the oldest sample.
    int bindex; // index used to keep track of b.
    short read_sample = mono_read_16Bit();
    bindex = 0; // reset index of b everytime the interupt function is called.
    output = 0; // reset index of b everytime the interupt function is called.
    upindex = ptr; // Sets upindex equal to ptr.
    if (ptr == 0) // if ptr is equal to 0 ensures downindex does not overflow.
    { downindex = N - 1; }
    else
    { downindex = ptr - 1; }

    if (ptr >= N / 2) /* Check the position of ptr if it is in the upper half
                         or lower half of the array.*/
    {
        while (upindex < N) // Performing convolution from ptr -> N (Symmetric).
        {
            output += b[bindex] * (x[upindex] + x[downindex]);
            bindex++;
            upindex++;
            downindex--;
        }
        upindex = 0; // resets upindex to 0 as it overflows
        while (bindex < (N / 2)) /* Performing the remaining convolution missed
                                 from the previous loop and since b is symmetric
                                 we can use this as the condition for when the
                                 loop needs to stop.*/
        {
            output += b[bindex] * (x[upindex] + x[downindex]);
            bindex++;
            upindex++;
            downindex--;
        }
    }
    else
    {
        while (downindex >= 0 && bindex < (N / 2)) /* Performing convolution from
                                                   ptr -> 0 (Symmetric) // bindex
                                                   condition added in the case that
                                                   ptr = 0.*/
        {
            output += b[bindex] * (x[upindex] + x[downindex]);
            bindex++;
            upindex++;
            downindex--;
        }
        downindex = N - 1; // Resets downindex as it overflows.
        while (bindex < N / 2)
        {
            output += b[bindex] * (x[upindex] + x[downindex]);
            bindex++;
            upindex++;
            downindex--;
        }
    }

    if (N % 2 == 1) /* Checking if the buffer size is odd if odd multiply the midpoint
                    of b[i] & x[i].*/
    {
        output += b[bindex] * x[(((N - 1) / 2) + ptr) % N];
```

```
    }

    if (ptr == 0) // Check if ptr overflows
    {
        ptr = N - 1;
    }
    else
    {
        ptr--; // Decrement ptr to allow the current sample to be stored correctly.
    }
    x[ptr] = read_sample;

    mono_write_16Bit(output); // Write output to the audio port of DSP.
}
```

Circular FIR Filter Attempt 3

This attempt on the circular buffer is very similar to attempt 2, however we instead tried to use one for loop as oppose to using 2 loops by using modulo operations which finds the remainder.

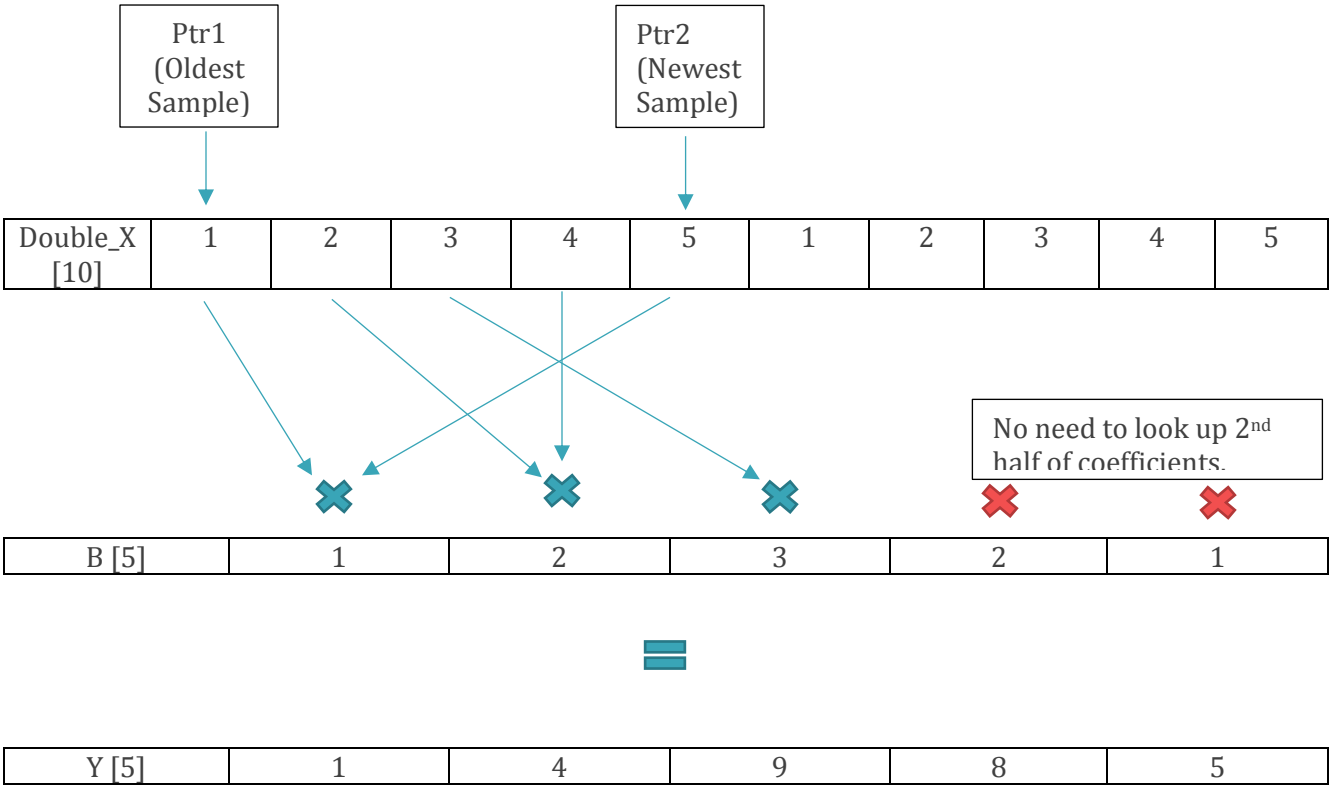
This attempts first stores the input sample into read_sample and resets the output to 0. Also, since we use the symmetric properties of b the for loop is, i then increments from 0 –> the modulo operator is then used to calculate the overflow so that the correct indexing is used for the array x. The if statement is used to calculate the midpoint of x given the array x is odd. We also include an if statement which checks for Ptr overflow and then it goes through mono_write and outputted.

However it is important to note this attempt was very inefficient in terms of cycles as the modulo operator is a cycle intensive operation thus this meant we needed an alternative method. This resulted in the double buffer optimization attempt.

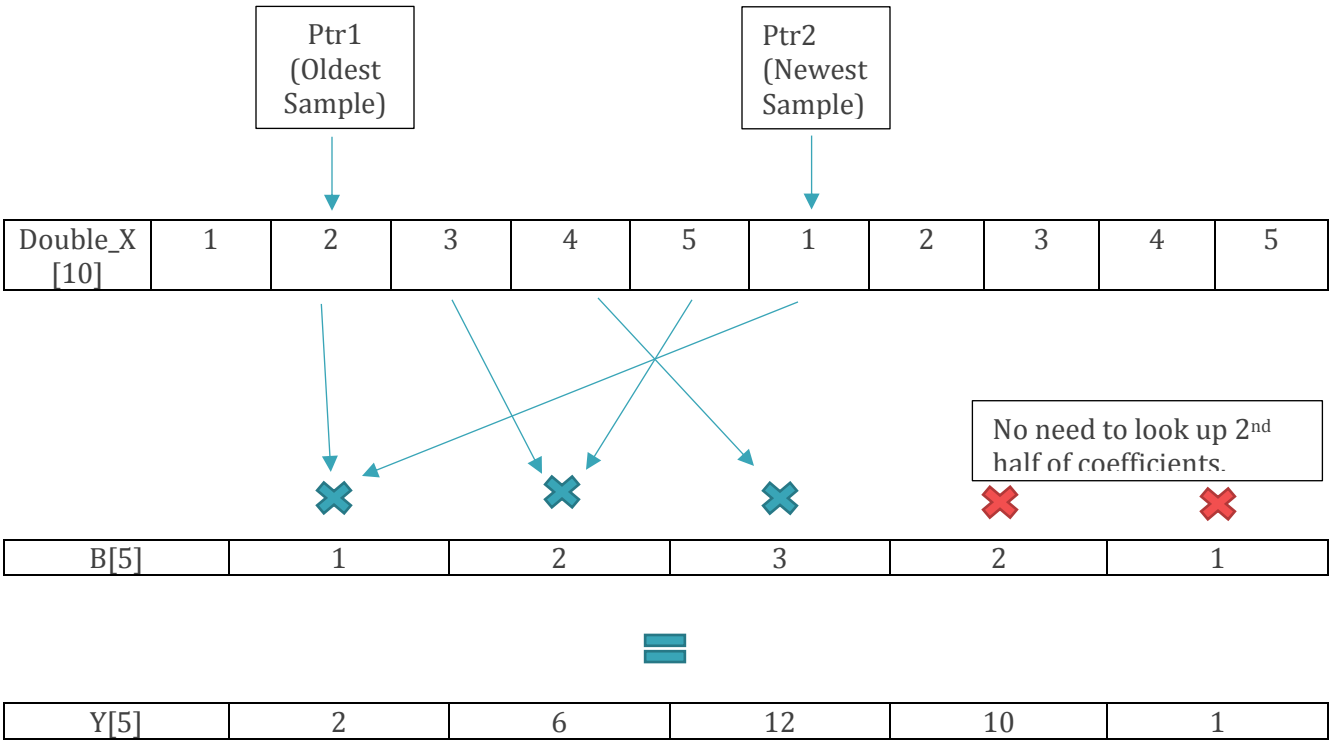
```
void circ_FIR_V2(void)
{
    short read_sample = mono_read_16Bit(); /* read_sample holds the input
                                              sample.*/
    output = 0; /* reset output to 0 after everytime the interrupt function
                is called.*/

    for (i = 0; i < N / 2; i++) // Performs symmetric convolution.
    {
     output += b[i] * (x[(ptr + i) % N] + x[((ptr - i - 1) % N + N) % N]);
     // Purpose of modulo operation is to fix the overflow.
    }

    if (N % 2 == 1) /* Checking if the buffer size is odd if odd multiply
                    the midpoint of b[i] & x[i].*/
    {
        output += b[(N - 1) / 2] * x[(ptr + (N - 1) / 2) % N];
    }

    if (ptr == 0) // Check if ptr overflows
    {
        ptr = N - 1; /* Decrement ptr to allow the current sample to be
                      stored correctly.*/
    }
    else
    {
        ptr--;
    }
    x[ptr] = read_sample;

    mono_write_16Bit(output); // Write output to the audio port of DSP.
}
```

Circular FIR Filter Attempt 4 –Double Buffer [Optimised]

The double buffer method is a method that uses an array X that is cloned and added to the end. We then have two pointers that remain the same distance apart and indicate the oldest and newest sample. With this method we can easily use the coefficients symmetry property to reduce the amount look up instructions. The best way to see how this works is through a visual example. Small example; N= 10, Ptr1=0, Ptr2 = N.



N= 10, Ptr1=1, Ptr2 = N+1.



...

We can see that that the two pointers show the oldest and newest samples in an easy enough way such that we can use the coefficient symmetry property which is far better then what was attempted in 'Circular FIR Filter attempt 3 ' where complex indexing was used even though it had just 1 for loop.

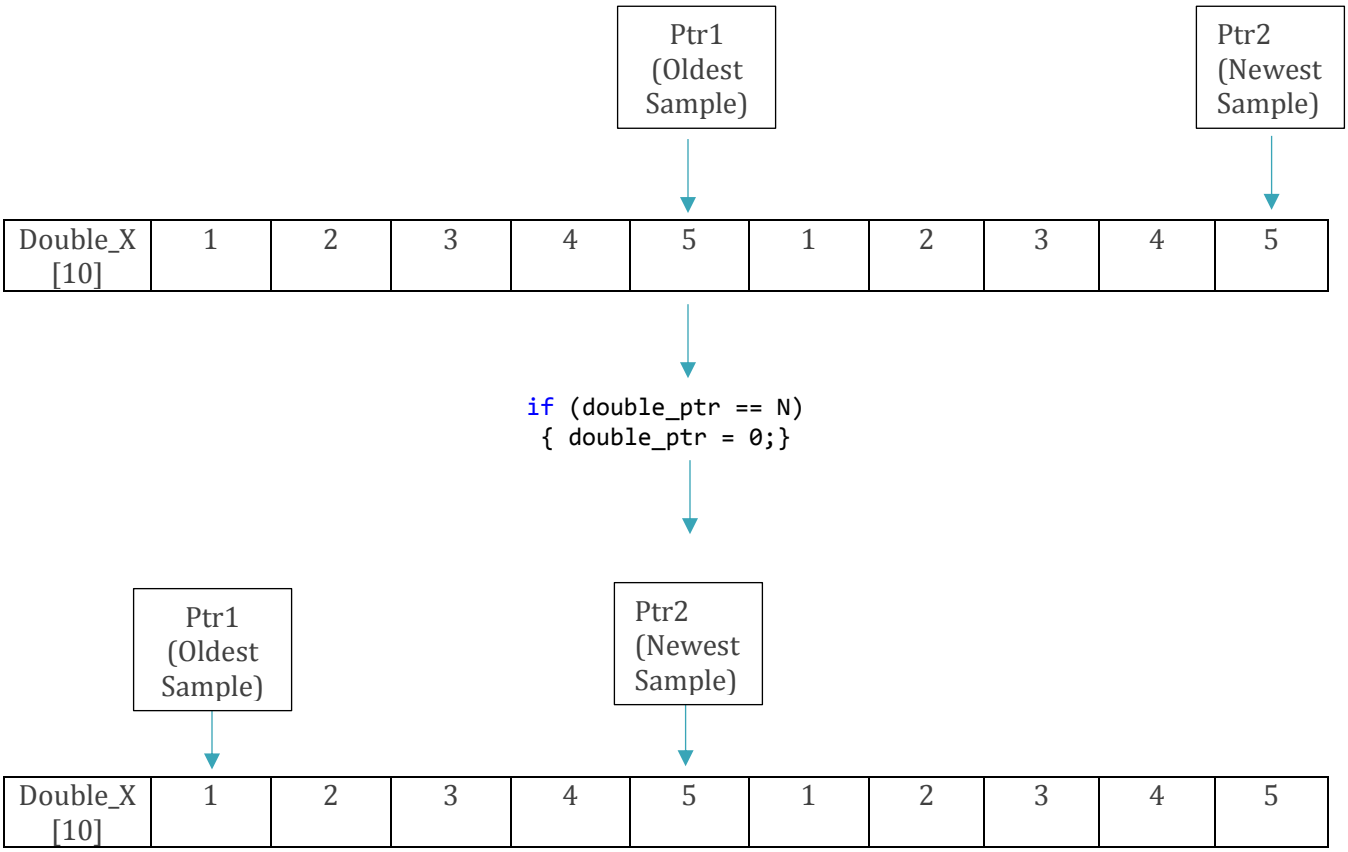
To implement this we first take in a sample and store it in variable of type short, This is because `mono_read_16Bit();` is 16 bit and so short is used. We then clone the samples to the double length buffer at points `double_ptr` and `N + double_ptr` through the entire buffer. We then enter

```
for (i = 0; i < N / 2; i++)
```

Which cycles through only half of the coefficients as we are using the symmetry property. Convolution is then performed just like the simple visual diagram above by reading values to the right from the `double_ptr1` which is the oldest samples and left from `double_ptr2.` The reason we read to the left for `double_ptr2` is because otherwise we will be reading a completely wrong sample.

We then have to check if N is odd, if it is odd the For loop would have missed out working the convolution of the midpoint. Hence we need to do an If condition to check if it is indeed ODD and if found to be odd we manually find the midpoint for convolution. It is found by first finding the midpoint of `double_x` then shifting it according by `double_ptr` to find the new midpoint and then multiply it by the mid coefficient term.

We then increment to `double_ptr` by 1 as shown in our visual example.



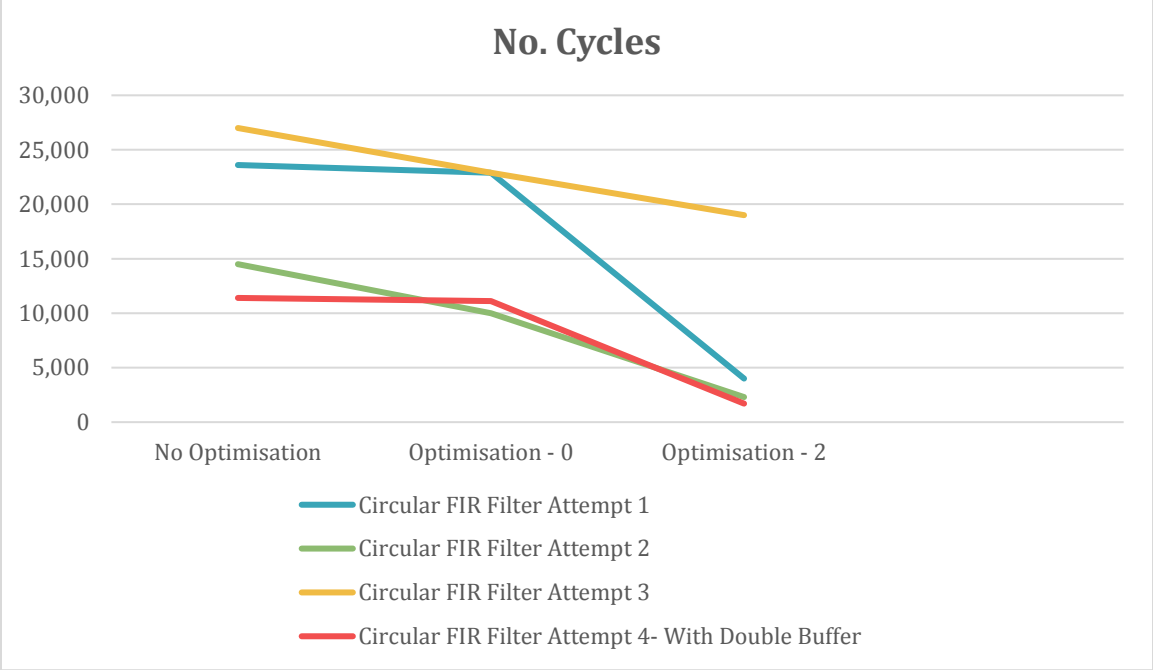
We then enter an IF statement should the double pointer reach N and hence wraps around and resets the pointers to original positions. We then do output write.

```
void double_buff_FIR(void)
{

short read_sample = mono_read_16Bit(); // read_sample holds the input sample.

output = 0; // reset output to 0 after everytime the interupt function is called.

double_x[double_ptr] = read_sample; // Stores input sample in the 1st half of the
array.

double_x[N + double_ptr] = read_sample; // Stores input sample in the 2nd half of the
array.

        // Performs symmetric convolution.
    {
       output += b[i] * (double_x[double_ptr + i + 1] + double_x[N + double_ptr - i]);
    }

    if (N % 2 == 1) // Checking if the buffer size is odd if odd multiply the midpoint
of b[i] & x[i].
    {
        output += b[((N - 1) / 2)] * double_x[((N + 1) / 2) + double_ptr];
    }

    double_ptr++; // Increment double_ptr++.
    if (double_ptr == N) // Ensuring the 2 halves of the array do not overlap.
    { double_ptr = 0; } // Resets double_ptr to 0.

    mono_write_16Bit(output); // Write output to the audio port of DSP.
}
```

Summary and Trace Scopes

After performing the Optimisation benchmarks we can see a variety of results with our best and most efficient method being the double buffer method. This is because it used by far the least amount of instructions.

No. Cycles	No Optimisation	Optimisation - 0	Optimisation - 2
Circular FIR Filter Attempt 1	**23,600**	**22,900**	**4000**
Circular FIR Filter Attempt 2	**14,500**	**10,000**	**2300**
Circular FIR Filter Attempt 3	**27,000**	**22,900**	**19,000**
Circular FIR Filter Attempt 4- With Double Buffer	**11,400**	**11,100**	**1,700**



It is possible to get even low amount of cycles on the double buffer method by using `float` types instead of `double` types for the coefficient array and sample array. This is because works because float are only 32 bit whilst `double` is 64 bit hence by using float you will have to trade precision in your readings for lower amount of cycles which would not be of great benefit as the amount of cycles lowered down by will not be substantial for the trade off in precision.

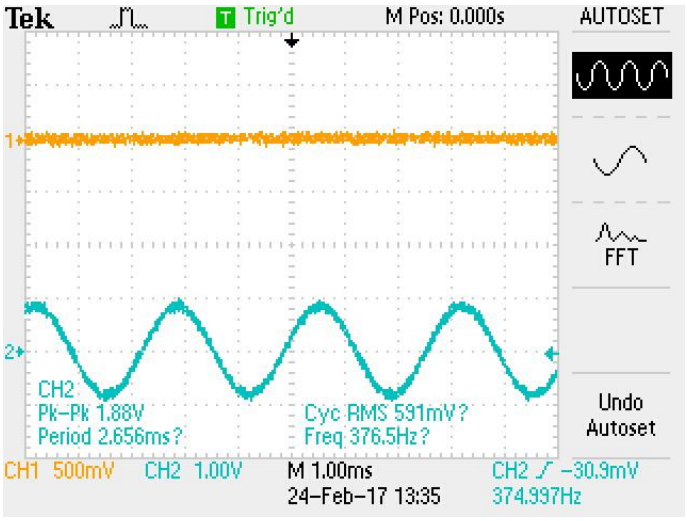


Figure 16: Image shows Double Buffer FIR filter with 375 Hz input. This is in the stop band of the filter and should result with a zero output as shown.

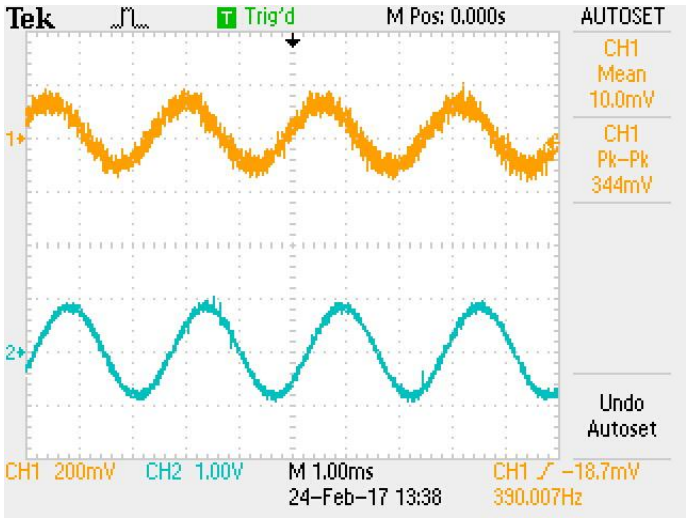


Figure 17: Image shows Double Buffer FIR filter with 390 Hz input. At this frequency we are approaching the cutoff and hence we should expect some attenuation at the output which we see here.

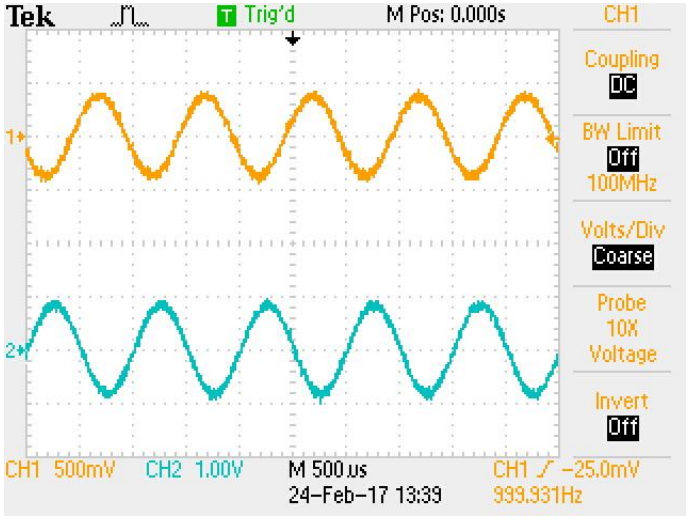


Figure 18: Image shows Double Buffer FIR filter with 1000 Hz input. At this frequency we are in the middle of the band pass filter and shouldn't expect attenuation. The reason why output is half is because of the potential divider at the input of the audio chip, halving the input. [Appendix]

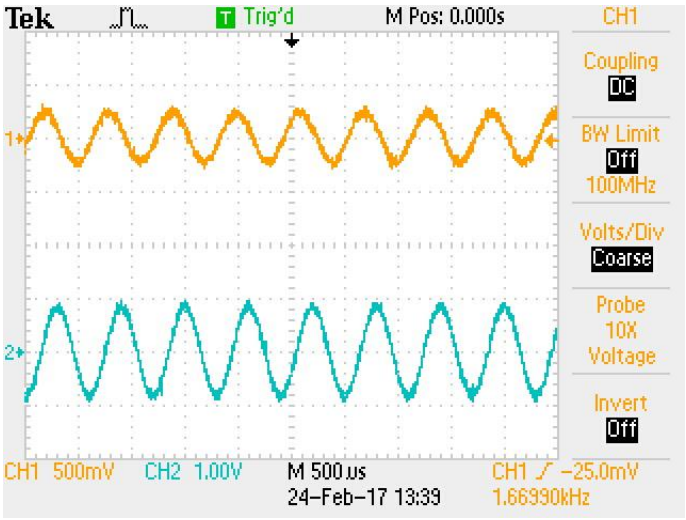


Figure 19: Image shows Double Buffer FIR filter with 1670 Hz input. At this frequency we are approaching the cutoff and hence we should expect some attenuation at the output which we see here.

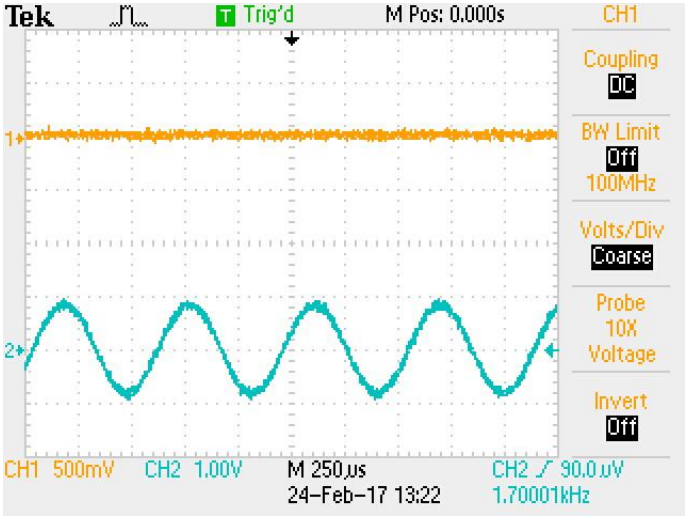
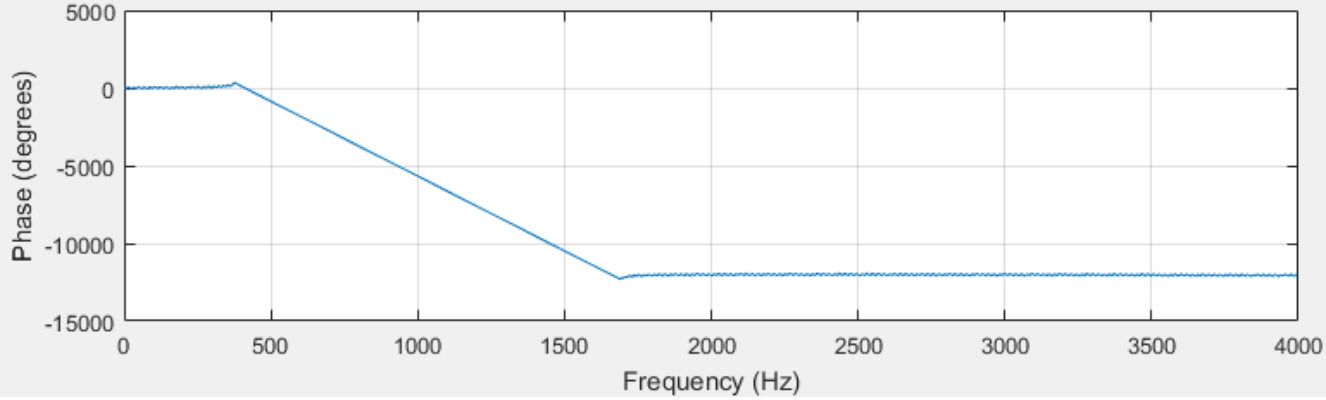
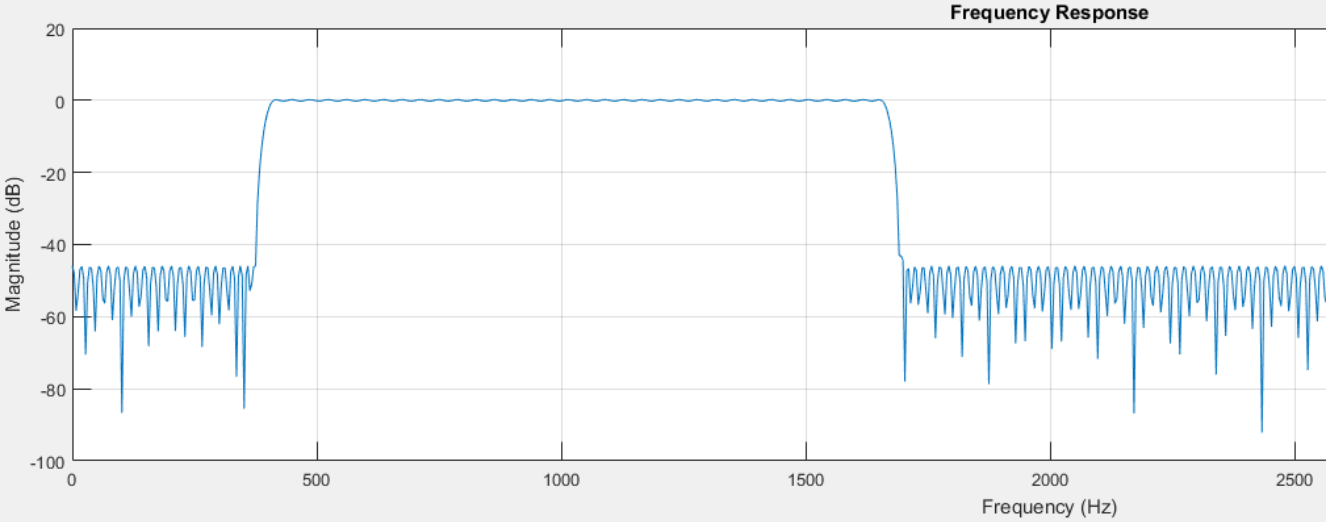


Figure 20: Image shows Double Buffer FIR filter with 1700 Hz input. This is in the stop band of the filter and should result with a zero output as shown.

Network analyser & Linear Phase

We noticed previously from our MATLAB simulations we get a phase and amplitude response:





To compare how our filter worked in practice we used a network analyser. We analysed our fastest and most precise FIR Filter which was our Circular attempt 4 – Double buffer version at 1700 cycles.

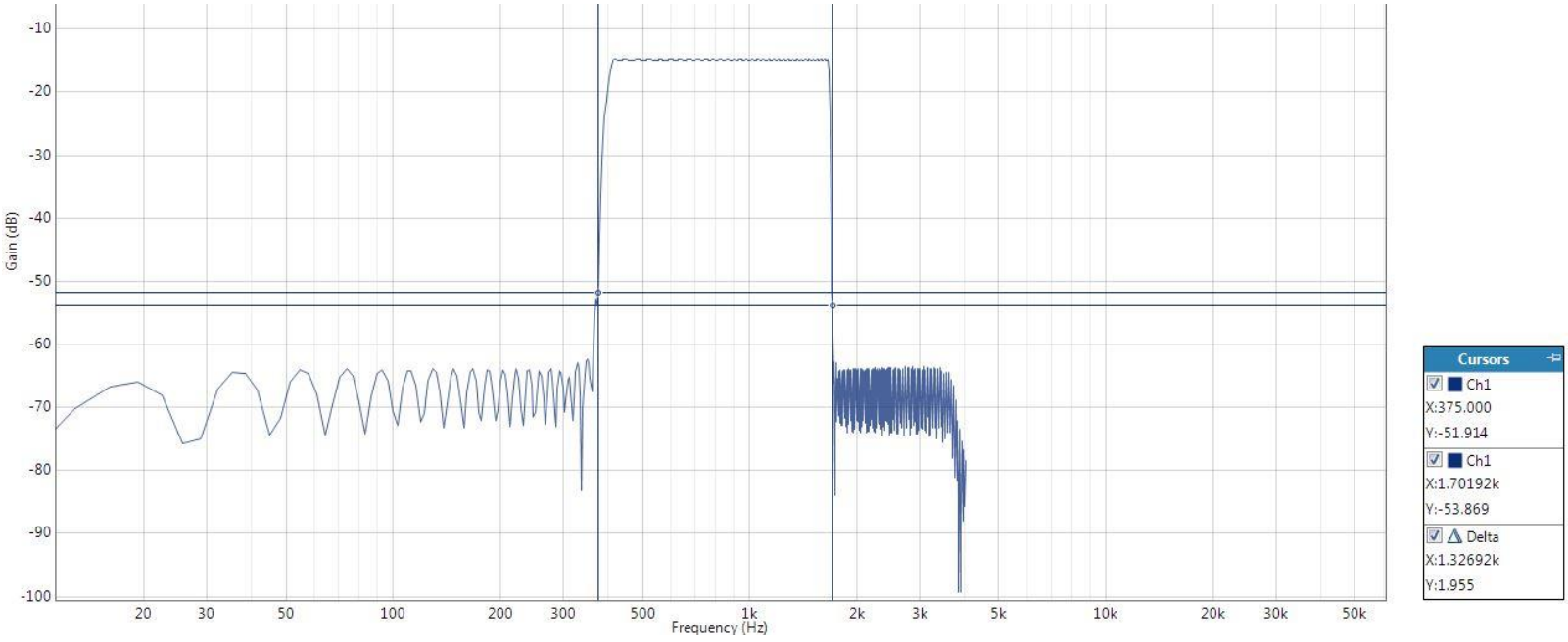


Figure 21: Image shows the frequency response of the Circular FIR filter. The cursors on the right show that at the cut off frequencies of 375 Hz and 1700 Hz that it reach less than -46 dB and hence within specifications. The reason gain in pass band is around -15 dB is because due to being connected to the network analyser the output is divided by a quarter hence it is indeed what we expect.

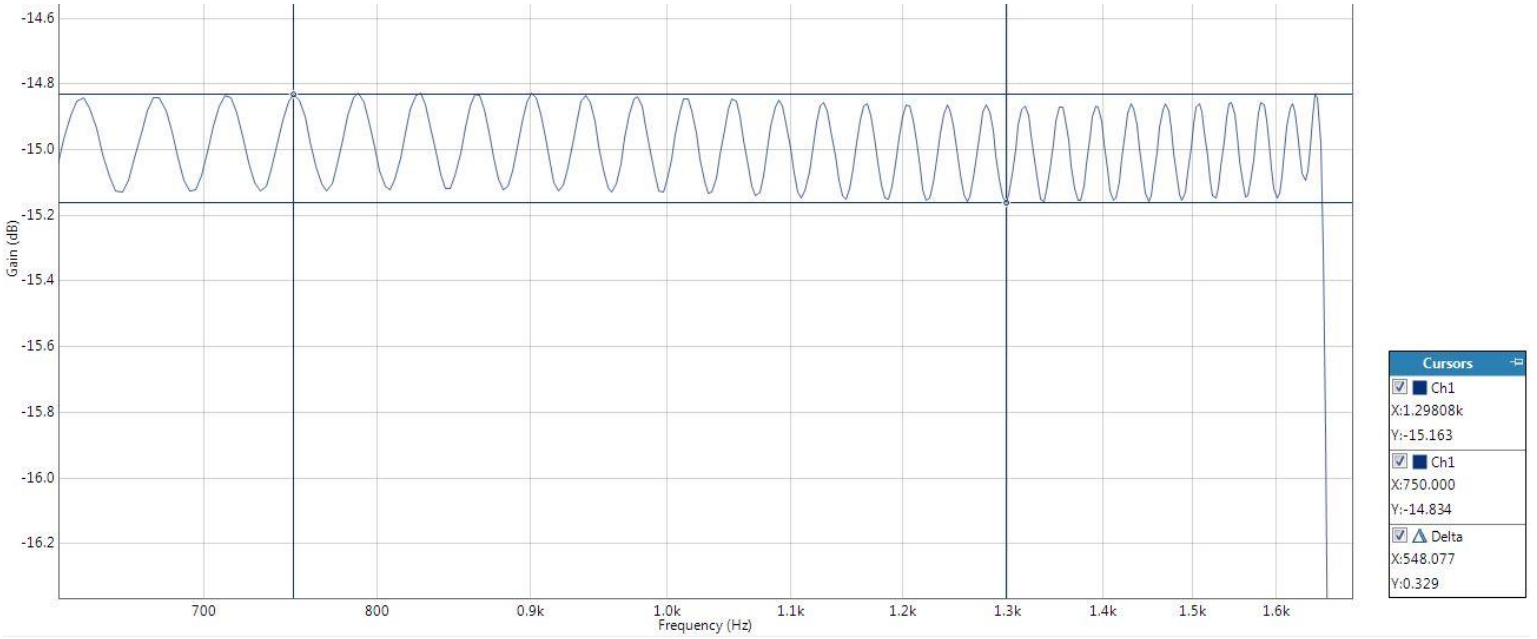


Figure 22: Image shows a zoomed in image of the pass band ripple. We can clearly see the ripple remains with 0.4 dB range and within specifications.

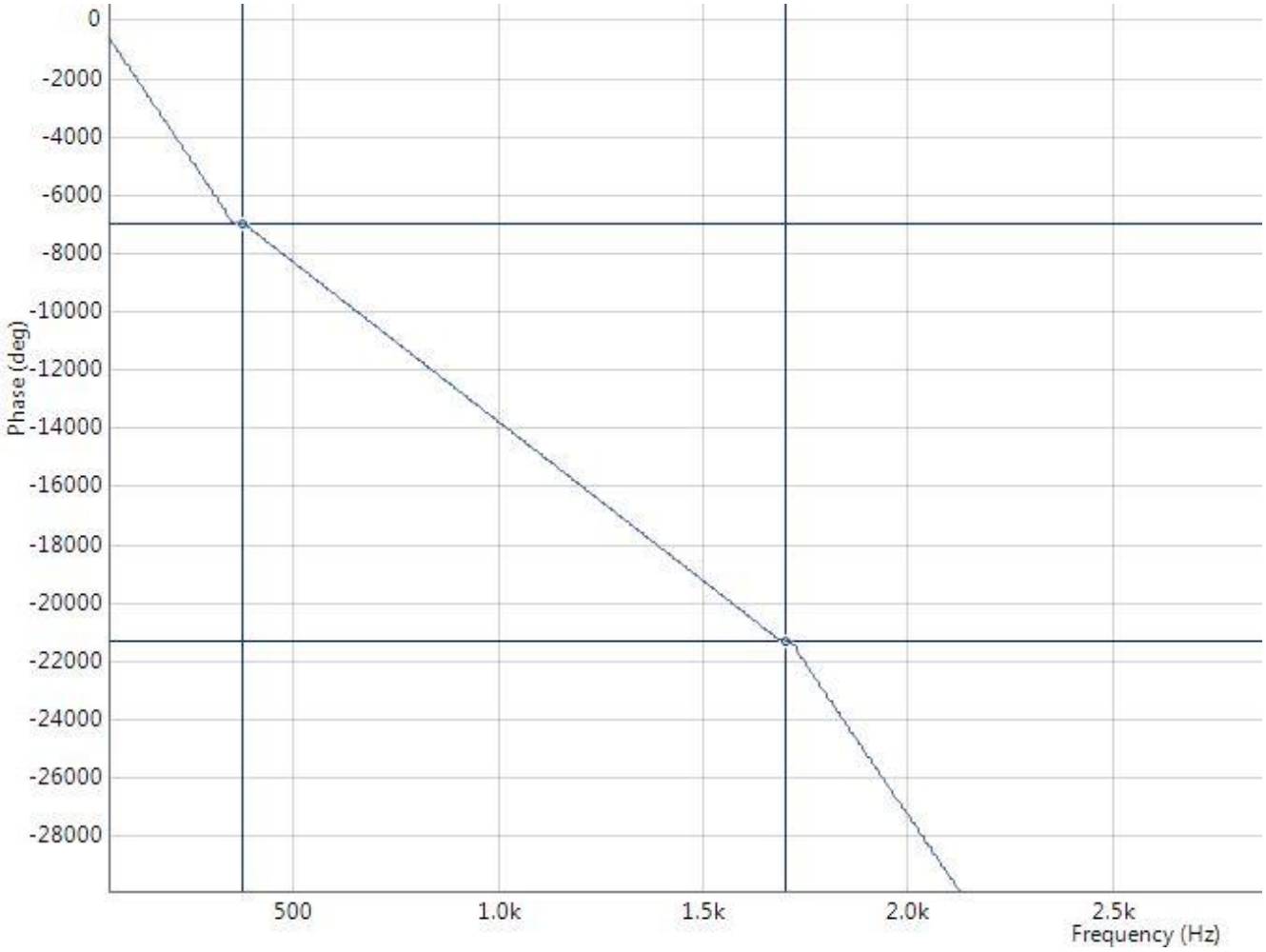


Figure 23: Image shows the phase response of the Circular FIR Filter, we can clearly see it behaves linearly from 375Hz and 1700 Hz which is to be expected as our coefficients are symmetrical hence our filter is linear phase.

Final Code

```
/*************************************************************************
      DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
      IMPERIAL COLLEGE LONDON

       EE 3.19: Real Time Digital Signal Processing
      Dr Paul Mitcheson and Daniel Harvey

         LAB 3: Interrupt I/O

             ********* I N T I O. C **********

  Demonstrates inputting and outputting data from the DSK's audio port using interrupts.

 *************************************************************************
  Updated for use on 6713 DSK by Danny Harvey: May-Aug 2006
Updated for CCS V4 Sept 10
 *************************************************************************/
/*
 * You should modify the code so that interrupts are used to service the
 *  audio port.
 */
/***************************** Pre-processor statements ****************************/

#include <stdlib.h>
//  Included so program can make use of DSP/BIOS configuration tool.
#include "dsp_bios_cfg.h"

/* The file dsk6713.h must be included in every program that uses the BSL.  This
   example also includes dsk6713_aic23.h because it uses the
   AIC23 codec module (audio interface). */
#include "dsk6713.h"
#include "dsk6713_aic23.h"
#include "coefs.txt"
// math library (trig functions)
#include <math.h>

// Some functions to help with writing/reading the audio ports when using interrupts.
#include <helper_functions_ISR.h>

/******************************* Global declarations ********************************/
#define N 429 // Size of buffers
double x[N]; // Stores the input sample
double output; // Stores the result of the convolution between b[N] & x[N]
int i; // Variable used for for loops
int ptr = N - 1; // Used for circular buffer where the most recent sample is at ptr
/*****************************Double Buffer Variables*****************************/
double double_x[2 * N]; // Size of double buffer
int double_ptr = 0; // Ptr for double buffer

/* Audio port configuration settings: these values set registers in the AIC23 audio
   interface to configure it. See TI doc SLWS106D 3-3 to 3-10 for more info. */
DSK6713_AIC23_Config Config = { \
/*************************************************************************/
/*   REGISTER            FUNCTION      SETTINGS         */
/*************************************************************************/\
    0x0017,  /* 0 LEFTINVOL  Left line input channel volume  0dB                */\
    0x0017,  /* 1 RIGHTINVOL Right line input channel volume 0dB                */\
    0x01f9,  /* 2 LEFTHPVOL  Left channel headphone volume   0dB                */\
    0x01f9,  /* 3 RIGHTHPVOL Right channel headphone volume  0dB                */\
```

```
    0x0011,  /* 4 ANAPATH    Analog audio path control       DAC on, Mic boost 20dB*/\
    0x0000,  /* 5 DIGPATH    Digital audio path control      All Filters off       */\
    0x0000,  /* 6 DPOWERDOWN Power down control              All Hardware on       */\
    0x0043,  /* 7 DIGIF      Digital audio interface format  16 bit                */\
    0x008d,  /* 8 SAMPLERATE Sample rate control             8 KHZ                 */\
    0x0001   /* 9 DIGACT     Digital interface activation    On                    */\
/************************************************************************/
};


// Codec handle:- a variable used to identify audio interface
DSK6713_AIC23_CodecHandle H_Codec;

/******************************* Function prototypes *******************************/
void init_hardware(void);
void init_HWI(void);
void ISR_AIC(void);  // ISR function to call the FIR functions
void non_circ_FIR(void); // Basic non circular buffer method
void non_circ_FIR_V1(void); // Calculating 2 Outputs at the same time
void non_circ_FIR_V2(void); // Calculating 9 Outputs at the same time
void non_circ_FIR_V3(void); // Using the symmetric properties of the coefficients in b
void circ_FIR(void); // Basic circular buffer
void circ_FIR_V1(void); // circular buffer using the symmetric properties of the coefficients
in b
void circ_FIR_V2(void);
void double_buff_FIR(void);

/********************************* Main routine *********************************/
void main()
{


    // initialize board and the audio port
    init_hardware();

    /* initialize hardware interrupts */
    init_HWI();

    /* loop indefinitely, waiting for interrupts */

    while (1)
    { };

}

/********************************* init_hardware() *********************************/
void init_hardware()
{
    // Initialize the board support library, must be called first
    DSK6713_init();

    // Start the AIC23 codec using the settings defined above in config
    H_Codec = DSK6713_AIC23_openCodec(0, &Config);

    /* Function below sets the number of bits in word used by MSBSP (serial port) for
    receives from AIC23 (audio port). We are using a 32 bit packet containing two
    16 bit numbers hence 32BIT is set for  receive */
    MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);

    /* Configures interrupt to activate on each consecutive available 32 bits
    from Audio port hence an interrupt is generated for each L & R sample pair */
```

```
    MCBSP_FSETS(SPCR1, RINTM, FRM);

    /* These commands do the same thing as above but applied to data transfers to
    the audio port */
    MCBSP_FSETS(XCR1, XWDLEN1, 32BIT);
    MCBSP_FSETS(SPCR1, XINTM, FRM);


}

/********************************* init_HWI() ************************************/
void init_HWI(void)
{
    IRQ_globalDisable(); // Globally disables interrupts
    IRQ_nmiEnable(); // Enables the NMI interrupt (used by the debugger)
    IRQ_map(IRQ_EVT_RINT1, 4); // Maps an event to a physical interrupt
    IRQ_enable(IRQ_EVT_RINT1); // Enables the event
    IRQ_globalEnable(); // Globally enables interrupts
}

/******************** WRITE YOUR INTERRUPT SERVICE ROUTINE HERE***********************/


void ISR_AIC(void)
{
    double_buff_FIR();
}

void non_circ_FIR(void)
{
    short read_sample = mono_read_16Bit(); // read_sample holds the input sample.
    output = 0; // reset output to 0 after everytime the interupt function is called.


    for (i = 0; i < N - 1; i++) // Performing convolution and storing the multiplication of
x[i] & b[i]
                                // inside output.
    {
        output += x[i] * b[i];
    }

    for (i = N - 1; i > 0; i--) // Shifting current sample to the right by 1 to allow space for
the input sample.
    {
        x[i] = x[i - 1];
    }
    x[0] = read_sample;

    mono_write_16Bit(output); // Write output to the audio port of DSP.
}

void non_circ_FIR_V1(void) // Multiple of 2
{
    short read_sample = mono_read_16Bit(); // read_sample holds the input sample.
    output = 0; // reset output to 0 after everytime the interupt function is called.

    for (i = 0; i < (N - 1) / 2; i++) // Performing convolution and storing the multiplication
of x[i] & b[i]
                                      // inside output,performing 2 multiply & add in 1
iteration of the for loop.
    {
```

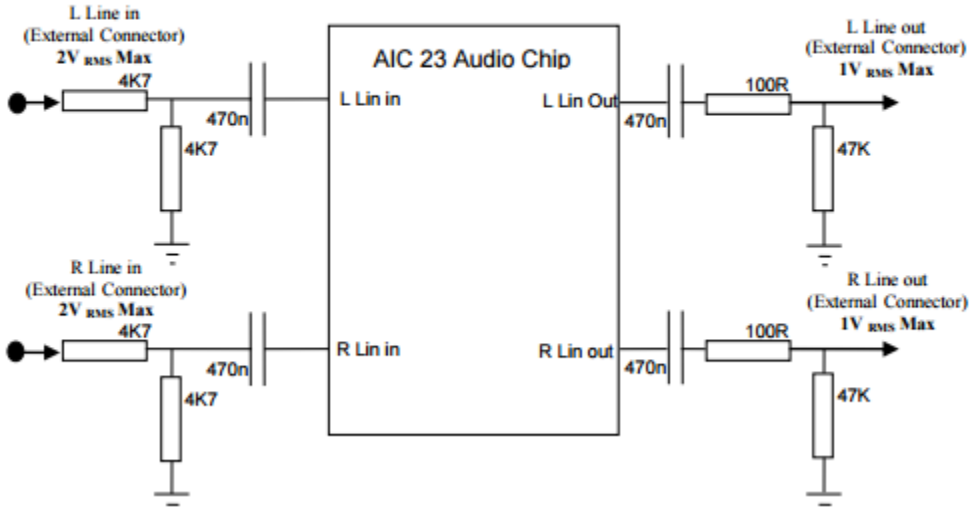
```
        output += x[i] * b[i] + x[N - 1 - i] * b[N - 1 - i];
    }
    if ((N - 1) % 2 == 0) // Checking if the buffer size is odd if odd multiply the midpoint of
b[i] & x[i].
    {
        output += x[(N - 1) / 2] * b[(N - 1) / 2];
    }

    for (i = N - 1; i > 0; i--) // Shifting current sample to the right by 1 to allow space for
the input sample.
    {
        x[i] = x[i - 1];
    }
    x[0] = read_sample;

    mono_write_16Bit(output); // Write output to the audio port of DSP.

}

void non_circ_FIR_V2(void) // Multiple of 9.
{
    short read_sample = mono_read_16Bit(); // read_sample holds the input sample.
    int indexjumper; // Used to skip samples that are already calculated.
    output = 0; // reset output to 0 after everytime the interupt function is called.

    for (i = 0; i < N / 9; i++)
    {
        indexjumper = 9 * i;  /* Used to skip samples that are already calculated being 9 since
                                 we do 9 calculation in one for loop.*/
        output += x[indexjumper] * b[indexjumper] + x[indexjumper + 1] * b[indexjumper + 1]
        + x[indexjumper + 2] * b[indexjumper + 2] + x[indexjumper + 3] * b[indexjumper + 3]
        + x[indexjumper + 4] * b[indexjumper + 4] + x[indexjumper + 5] * b[indexjumper + 5]
        + x[indexjumper + 6] * b[indexjumper + 6] + x[indexjumper + 7] * b[indexjumper + 7]
        + x[indexjumper + 8] * b[indexjumper + 8];
    }

    if (N % 9 != 0) // Check for the remaining input samples that we may have missed out from
the for loop above.
    {
        for (i = 0; i < N % 9; i++)
        {
            output += b[N - 1 - i] * x[N - 1 - i];
        }
    }

    for (i = N - 1; i > 0; i--) // Shifting current sample to the right by 1 to allow space for
the input sample.
    {
        x[i] = x[i - 1];
    }
    x[0] = read_sample;

    mono_write_16Bit(output); // Write output to the audio port of DSP.
}

void non_circ_FIR_V3(void) // Using the symmetric properties of b.
{
    short read_sample = mono_read_16Bit(); // read_sample holds the input sample.
    output = 0; // reset output to 0 after everytime the interupt function is called.

    for (i = 0; i < N / 2; i++) // For loops N/2 as we know the symmetric properties of b.
```

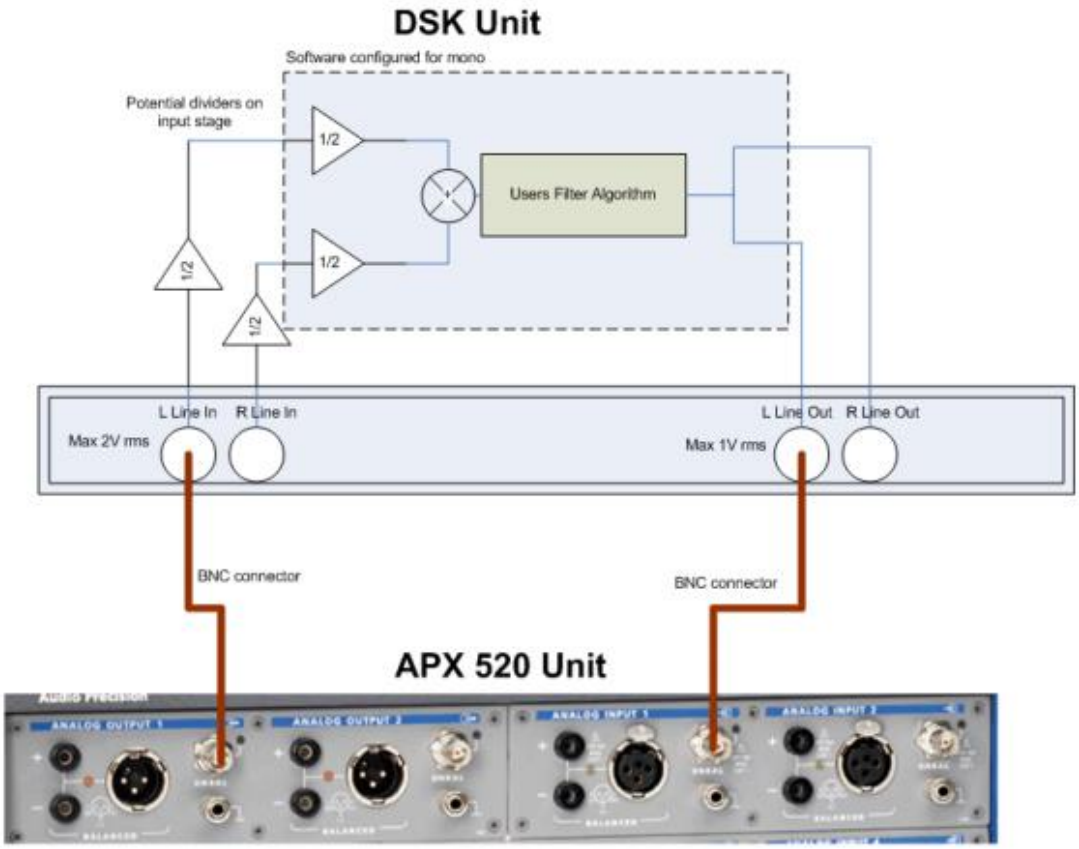
```
    {
        output += b[i] * (x[i] + x[N - 1 - i]);
    }
    if ((N - 1) % 2 == 0) // Checking if the buffer size is odd if odd multiply the midpoint of
b[i] & x[i].
    {
        output += x[(N - 1) / 2] * b[(N - 1) / 2];
    }

    for (i = N - 1; i > 0; i--) // Shifting current sample to the right by 1 to allow space for
the input sample.
    {
        x[i] = x[i - 1];
    }
    x[0] = read_sample;

    mono_write_16Bit(output); // Write output to the audio port of DSP.
}

void circ_FIR(void)
{
    short read_sample = mono_read_16Bit(); // read_sample holds the input sample.
    output = 0; // reset output to 0 after everytime the interupt function is called.

    for (i = ptr; i < N; i++) // Performing convolution upwards from ptr to the end of the
buffer.
    {
        output += x[i] * b[i - ptr];
    }

    for (i = 0; i < ptr; i++) // Perform convolution upwards from 0 to ptr
    {
        output += x[i] * b[N - ptr + i];
    }


    if (ptr == 0) // Check if ptr overflows
    {
        ptr = N - 1;
    }
    else
    {
        ptr--; // Decrement ptr to allow the current sample to be stored correctly.
    }
    x[ptr] = read_sample;

    mono_write_16Bit(output); // Write output to the audio port of DSP.
}

void circ_FIR_V1(void)
{
    int upindex; // index for the recent sample.
    int downindex; // index for the oldest sample.
    int bindex; // index used to keep track of b.
    short read_sample = mono_read_16Bit(); // read_sample holds the input sample.
    bindex = 0; // reset index of b everytime the interupt function is called.
    output = 0; // reset index of b everytime the interupt function is called.
    upindex = ptr; // Sets upindex equal to ptr.
    if (ptr == 0) // if ptr is equal to 0 ensures downindex does not overflow.
    { downindex = N - 1; }
    else
```

```
    { downindex = ptr - 1; }

    if (ptr >= N / 2) // Check the position of ptr if it is in the upper half or lower half of
the array.
    {
        while (upindex < N) // Performing convolution from ptr -> N (Symmetric).
        {
            output += b[bindex] * (x[upindex] + x[downindex]);
            bindex++;
            upindex++;
            downindex--;
        }
        upindex = 0; // resets upindex to 0 as it overflows
        while (bindex < (N / 2)) /* Performing the remaining convolution missed
            from the previous loop and since b is symmetric we can use this as the condition
for when the loop needs to stop.*/
        {
            output += b[bindex] * (x[upindex] + x[downindex]);
            bindex++;
            upindex++;
            downindex--;
        }
    }
    else
    {
        while (downindex >= 0 && bindex < (N / 2)) /* Performing convolution from ptr -> 0
(Symmetric)
                                                   // bindex condition added in the case that ptr
= 0.*/
        {
            output += b[bindex] * (x[upindex] + x[downindex]);
            bindex++;
            upindex++;
            downindex--;
        }
        downindex = N - 1; // Resets downindex as it overflows.
        while (bindex < N / 2)
        {
            output += b[bindex] * (x[upindex] + x[downindex]);
            bindex++;
            upindex++;
            downindex--;
        }
    }

    if (N % 2 == 1) // Checking if the buffer size is odd if odd multiply the midpoint of b[i]
& x[i].
    {
        output += b[bindex] * x[(((N - 1) / 2) + ptr) % N];
    }

    if (ptr == 0) // Check if ptr overflows
    {
        ptr = N - 1;
    }
    else
    {
        ptr--; // Decrement ptr to allow the current sample to be stored correctly.
    }
    x[ptr] = read_sample;
```

```
    mono_write_16Bit(output); // Write output to the audio port of DSP.
}


void circ_FIR_V2(void)
{
    short read_sample = mono_read_16Bit(); // read_sample holds the input sample.
    output = 0; // reset output to 0 after everytime the interupt function is called.

    for (i = 0; i < N / 2; i++) // Performs symmetric convolution.
    {
        output += b[i] * (x[(ptr + i) % N] + x[((ptr - i - 1) % N + N) % N]);
        // Purpose of modulo operation is to fix the overflow.
    }

    if (N % 2 == 1) // Checking if the buffer size is odd if odd multiply the midpoint of b[i]
& x[i].
    {
        output += b[(N - 1) / 2] * x[(ptr + (N - 1) / 2) % N];
    }

    if (ptr == 0) // Check if ptr overflows
    {
        ptr = N - 1; // Decrement ptr to allow the current sample to be stored correctly.
    }
    else
    {
        ptr--;
    }
    x[ptr] = read_sample;

    mono_write_16Bit(output); // Write output to the audio port of DSP.
}

void double_buff_FIR(void)
{
    short read_sample = mono_read_16Bit(); // read_sample holds the input sample.
    output = 0; // reset output to 0 after everytime the interupt function is called.
    double_x[double_ptr] = read_sample; // Stores input sample in the 1st half of the array.
    double_x[N + double_ptr] = read_sample; // Stores input sample in the 2nd half of the
array.

    for (i = 0; i < N / 2; i++) // Performs symmetric convolution.
    {
        output += b[i] * (double_x[double_ptr + i + 1] + double_x[N + double_ptr - i]);
    }

    if (N % 2 == 1) // Checking if the buffer size is odd if odd multiply the midpoint of b[i]
& x[i].
    {
        output += b[((N - 1) / 2)] * double_x[((N + 1) / 2) + double_ptr];
    }

    double_ptr++; // Increment double_ptr++.
    if (double_ptr == N) // Ensuring the 2 halves of the array do not overlap.
    { double_ptr = 0; } // Resets double

    mono_write_16Bit(output); // Write output to the audio port of DSP.
}
```

Appendix


L Line in
(External Connector)
2V RMS Max
4K7
470n
4K7
AIC 23 Audio Chip
L Lin in
L Lin Out
470n
100R
L Line out
(External Connector)
1V RMS Max
47K
R Line in
(External Connector)
2V RMS Max
4K7
470n
4K7
R Lin in
R Lin out
470n
100R
R Line out
(External Connector)
1V RMS Max
47K



References

1. D. Mitcheson, P.P. (2014) *EE3-19 Real Time Digital Signal Processing*. Available at: https://bb.imperial.ac.uk/webapps/blackboard/execute/content/file?cmd=view&content_id=_885311_1&course_id=_9509_1 (Accessed: 20 February 2017).

2. W. H. House, A. (2004) *The Convolution Sum for Discrete-Time LTI Systems*. Available at: http://www.eecg.toronto.edu/~ahouse/mirror/engi7824/course_notes_7824_part6.pdf (Accessed: 20 February 2017).