EE3-19 Real Time Digital Signal Processing

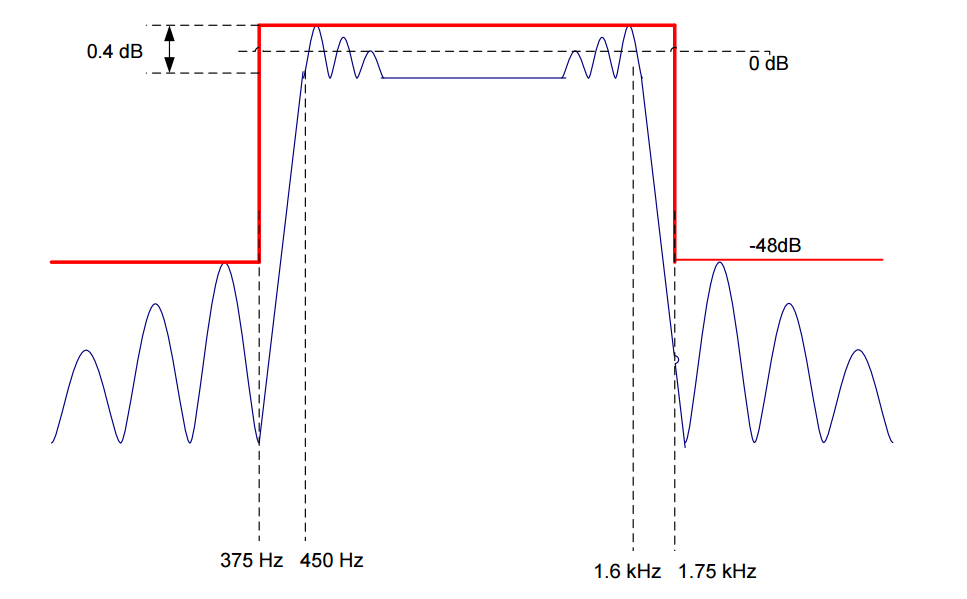
Lab 4 – FIR Design

## Group members: Ryan Dowse (rd613 / 00865495) Benjamin Ni (bwn13 / 00874386)

Introduction

The objective of this lab was to design a finite impulse response (FIR) filter that meets the following specifications:

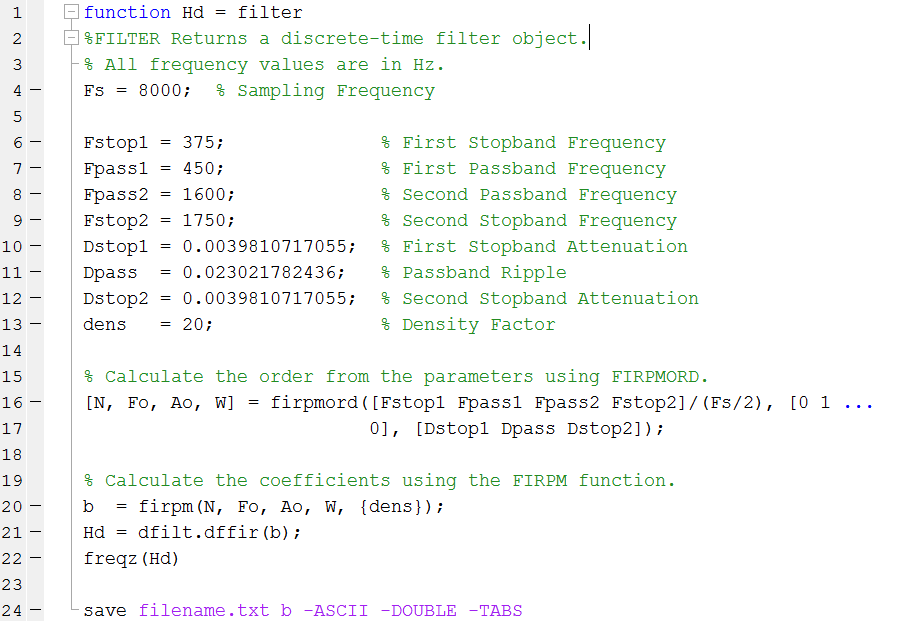
* The passband gain is 0 dB, and has a maximum ripple of 0.4 dB.
* The upper and lower limits of the passband are 450 Hz and 1.6 kHz respectively.
* The stopband attenuation is at least -48 dB.
* The upper limit of the first stopband is 375 Hz.
* The lower limit of the second stopband is 1.75 kHz.



**TODO Fig: FIR specification**

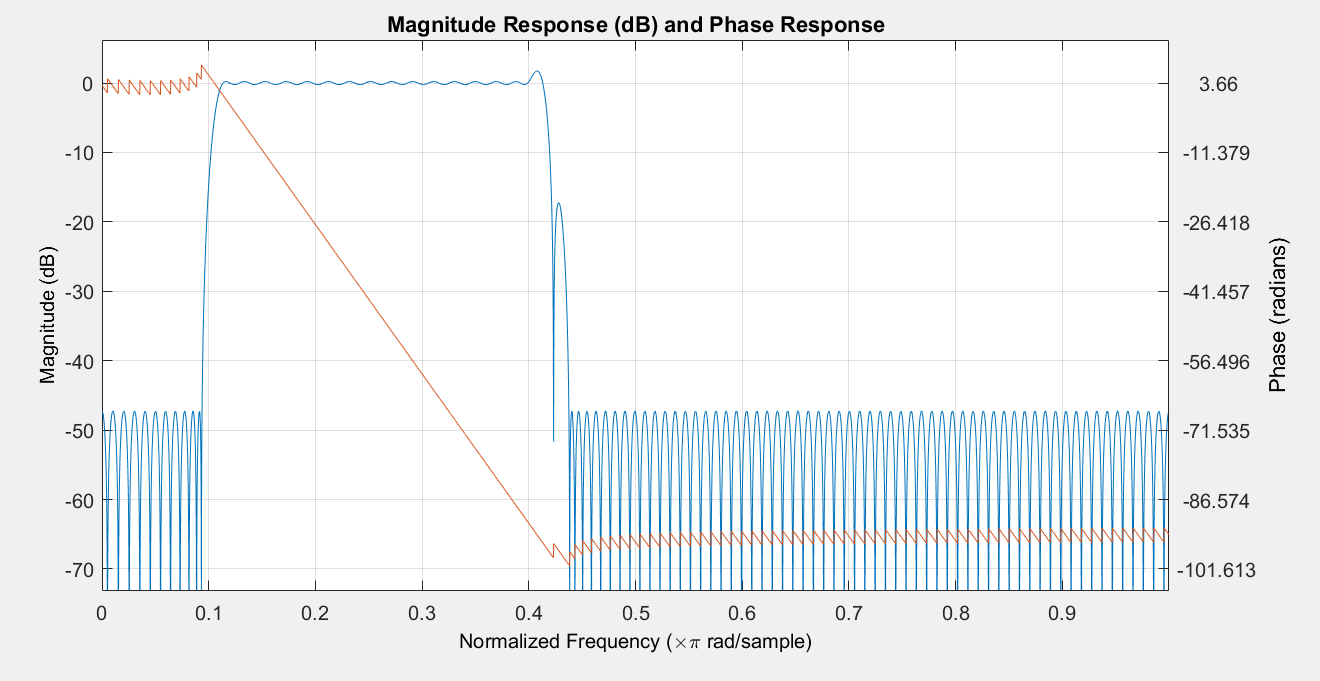
Filter Design

The filter was designed in Matlab. The following Matlab code was used to produce the coefficients for the specified FIR filter, using the Parks-McClelland algorithm. This code was generated using Matlab’s filter design and analysis tool, where the required filter specification was inputted into the tool, with a sampling frequency of 8 kHz. This was the first iteration of the Matlab code.



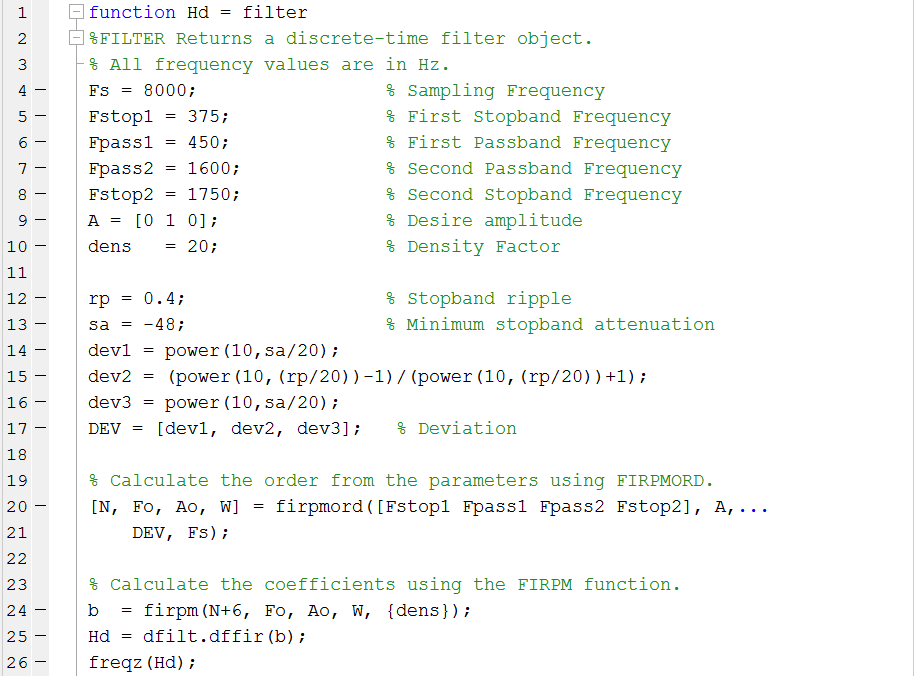
**TODO fig. First iteration of Matlab filter code**

The proof of frequency response of the designed filter is shown in TODO fig. It is linear phase and is of order 206. Note that the sampling frequency is 8 kHz, hence π rad/sample corresponds to 4 kHz.

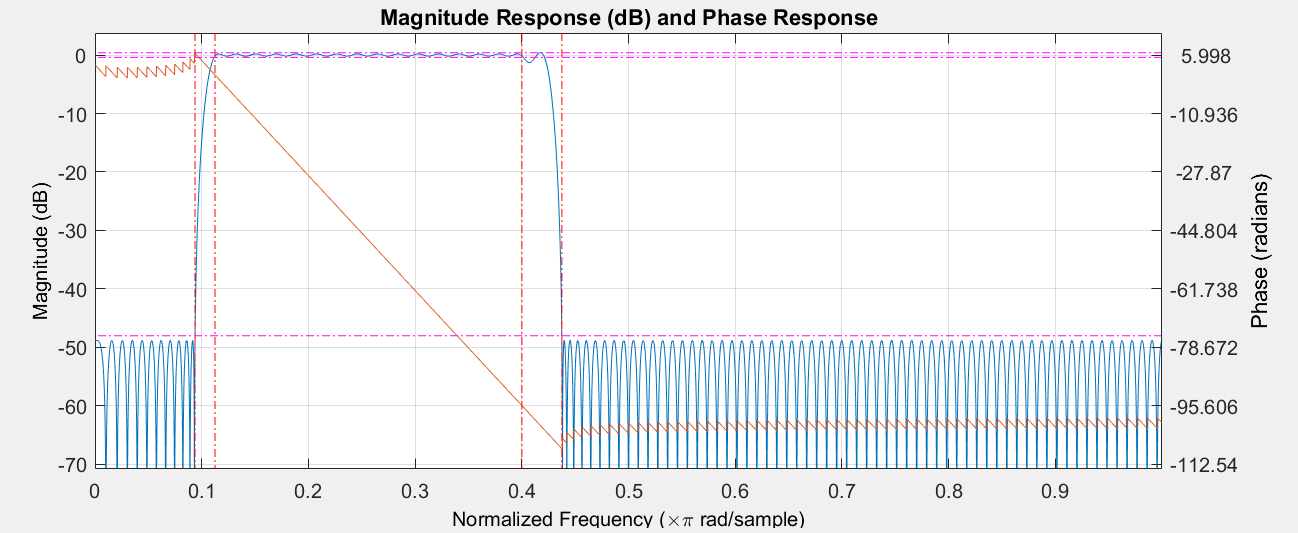


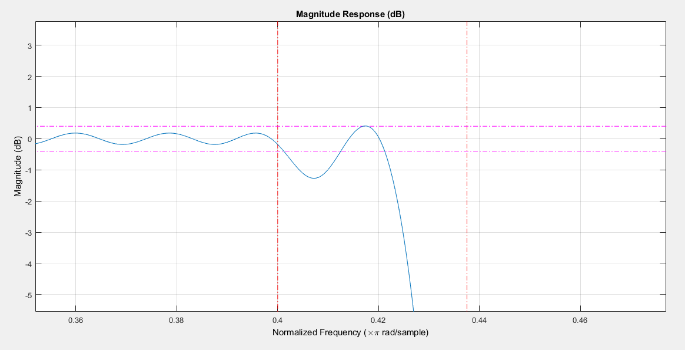
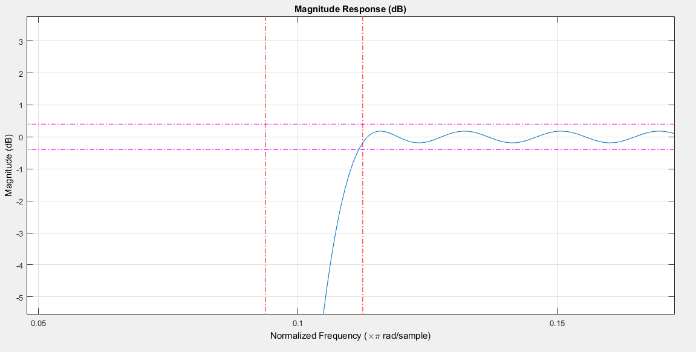
**TODO Fig. Minimum order FIR filter magnitude (dB) and phase (radians) response**

While this filter meets the specifications, there are ripples in the 1.6 kHz - 1.75 kHz transition band, one of which has a 1.6 dB gain. This could be undesirable; according to the filter specifications a gain of >0.4 dB is not expected, and could lead to voltage overload. We have solved this with our second iteration of filter design, with Matlab code and response graph shown in TODO figureS. Vertical dotted lines mark the respective stopband and passband frequencies as outlined in the specifications; horizontal dotted lines demarcate +0.2 dB, -0.2 dB and -48 dB. It can be seen from the zoomed in sections that the magnitude response between 450 Hz and 1600 Hz falls within the 0.4 dB ripple tolerance as required in the specification.



**TODO Fig: matlab code for improved filter design**

****



**TODO Fig.**

**Top: Order 212 FIR filter magnitude (dB) and phase (radians/sample) response**

**Left: Response zoomed in at 0.1125π rad/sample = 450 Hz**

**Right: Response zoomed in at 0.4 π rad/sample = 1600 Hz**

The first major change is increasing the order of the filter from 206 to 212, by passing in N+6 as the order argument to firpm(). This is done to remove the sudden jump in magnitude shown in the upper transition band of TODO (fig first graph). By increasing the order of the filter this cause a steeper roll off, allowing for the filter to have a smooth transition.

The deviation values dev = [dev1, dev2, dev3] were also recalculated, to ensure that the passband ripple is be within the strict specification of 0.4 dB. We used the following formula, where rp is the permitted passband ripple:

Filter Implementation

# Interrupt Service HandlerISR.PNG

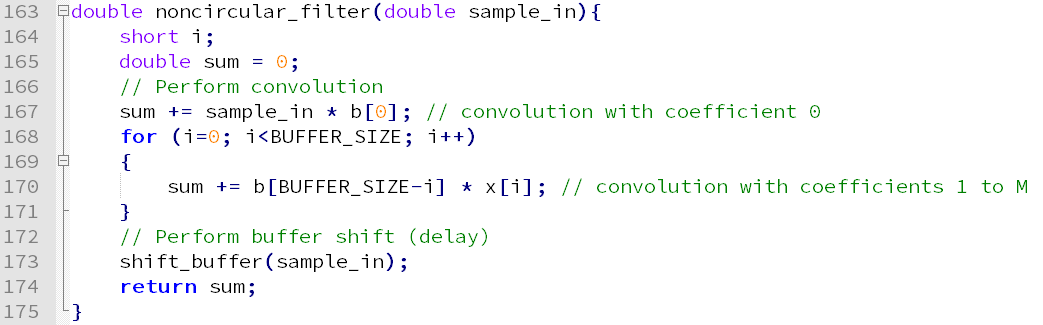
**TODO Fig: Code for Interrupt Service Handler (ISR)**

The processor essentially performs three actions in sequence during each interrupt: read from input, process samples, send output. Reading from input is done with mono\_read\_16Bit() and output is done with mono\_write\_16Bit(). These functions use short as the type for the samples, but our FIR coefficients are of type double, and we need to multiply the two as part of the processing. As such, the samples would be (implicitly) cast to type double before multiplication takes place. Casting each time a multiplication takes place incurs additional clock cycles; instead, we store the samples in our buffer as doubles. Typecasting is done along with the input and output functions, hence our processing function takes in the current sample as a double.

The switch-case block selects the specific filter implementation to use, based on the variable select\_buffer. This expedites the debugging process, allowing us to change the filter at runtime by modifying select\_buffer.

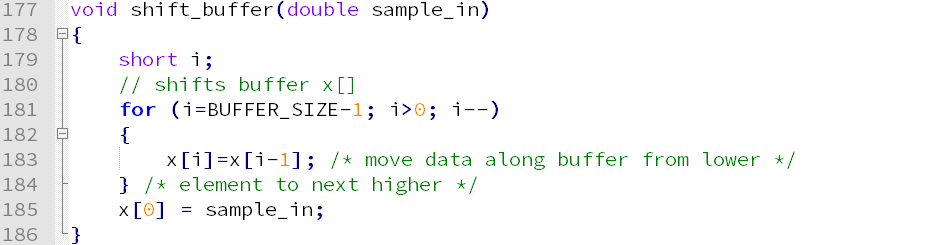
# Non-circular filter

The following code implements a simple non-circular filter. The function returns the convolution of filter coefficients b[i] with current and past input samples (the latter stored in a delay buffer x[i]). Since our filter is of order 206, it has 207 taps and requires 207 multiply-accumulates of b and x. We only require a delay buffer of size 206 however; this stores the past 206 samples, which are processed along with the currently read sample. This reduces the memory reads required from the buffer.



**TODO Fig: Basic non-circular filter, noncircular\_filter()**

Following the convolution, the function shift\_buffer shown in TODO fig is called. Here, each sample is shifted down one space in the array, so that the newest sample can be inserted at the start (index 0), and the last sample is then dropped from the end.



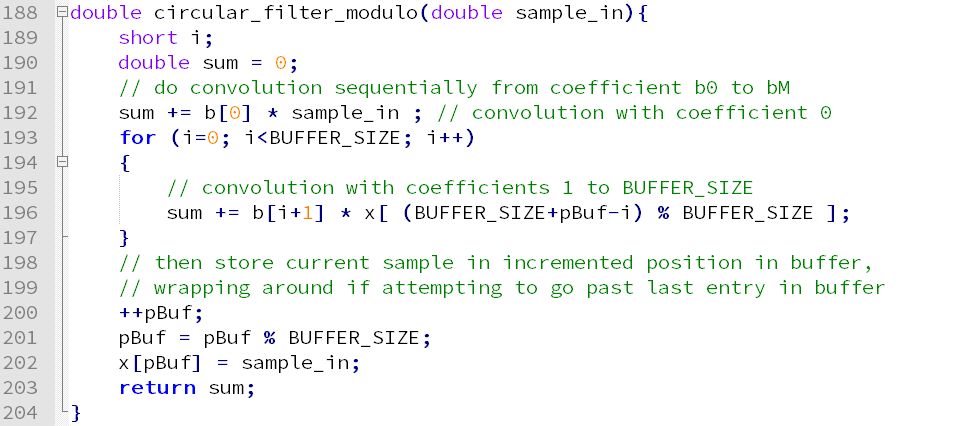
**TODO Fig: Shift buffer**

# Circular filter

Shown in the table below is an intuitive implementation of convolution with a circular buffer. We now require a short pBuf, the index to the most recent sample in the buffer. We thus obtain the following relation:

|  |  |
| --- | --- |
| **sample delay** | **variable name in code** |
| 0 | sample\_in |
| 1 | x[pBuf] |
| 2 | x[pBuf-1] |
| i+1 | x[pBuf-i] |

This however, does not account for wraparound of the circular buffer. This is easily solved by performing modulo by BUFFER\_SIZE, which allows the index to wraparound to within the valid buffer index range (0 to BUFFER\_SIZE-1). After convolution, we store the newest sample in the buffer, as with the noncircular buffer, then increment pBuf to update the index of the newest sample. There is no need to perform shift\_buffer since the buffer wraps around; eliminating the overhead of shift\_buffer is a significant advantage. However, it is necessary to ensure that pBuf remains within the index range, otherwise integer overflow of pBuf might occur if the program runs for an extended period of time. This is done by performing another modulo on pBuf by BUFFER\_SIZE.

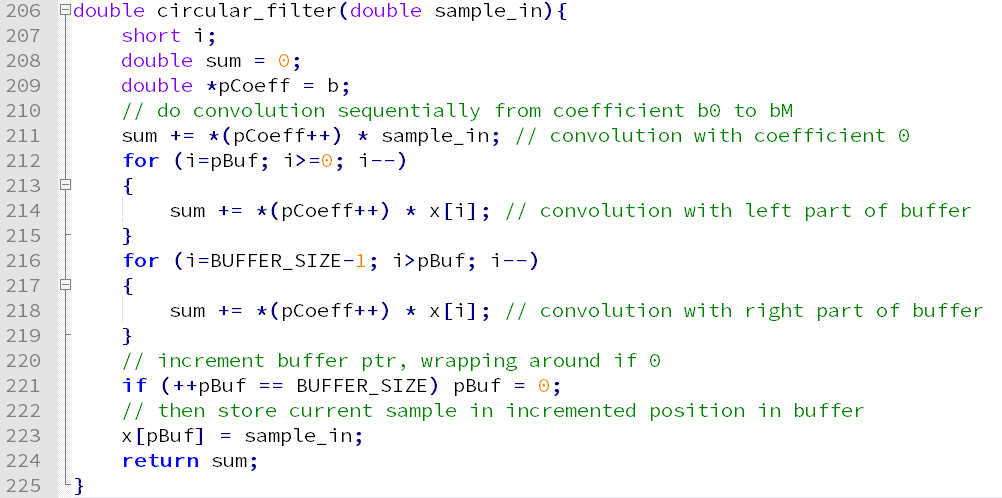


**TODO Fig: Initial circular filter design, circular\_filter\_modulo()**

This design may be improved in two ways. Firstly, the modulo function is slow, which impedes the performance of the for-loop. This can be remedied by splitting the for-loop into two: one dealing with the left part of the buffer up to index pBuf (i.e. indices 0 to pBuf), the other for the remaining part of the buffer (indices pBuf+1 to BUFFER\_SIZE-1).

Secondly, up to this point we have been performing convolution sequentially from coefficient b[0] to b[BUFFER\_SIZE-1]. Sequential array references can be optimised for speed by using incremented pointers instead. By defining a pointer pCoeff that starts at address b[0], we may invoke \*(pCoeff++) repeatedly to access each coefficient. The increment operator performs pointer incrementing, making it point to the next element in the coefficient array. We also replace the modulo that keeps pBuf within buffer index range with an “if” conditional check.

With these two design optimisations, we obtain the code for circular\_filter shown in TODO fig:

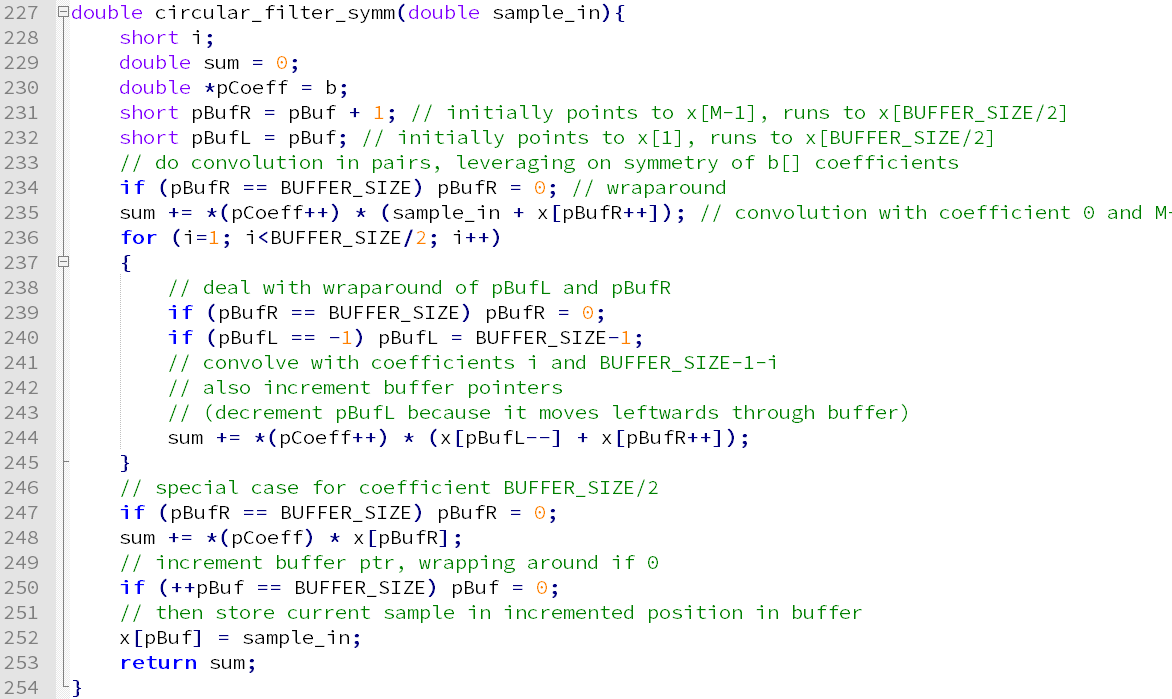


**TODO Fig: Circular filter with modulo and array reference optimised, circular\_filter()**

Now the code is quite optimal, but we can still improve on this further by considering the nature of the filter. As it is linear phase, its coefficients are symmetric about the middle coefficient, i.e. . As such, we may perform and in the same multiply-accumulate by factorising into the following expression:

This would require two pointers for the buffer, one for (pBufL) and (pBufR). As we work through the convolution, i iteratively increases, which implies that in our implementation pBufL will decrease while pBufR will increase. As before, since this is a circular buffer we must account for wraparound with every increment of i.

We iterate as above until pBufL and pBufR converge. If there are an odd number of taps, which is the case for our filter, both pBufL and pBufR would be pointing at the sample corresponding to the convolution with b[BUFFER\_SIZE/2]. Thus as shown in TODO fig, we need an additional multiply-accumulate for this special case upon completion of the for-loop. Either pointer can be used for this final multiply-accumulate; we used pBufR in our implementation.



**TODO Fig: Circular filter utilising filter coefficient symmetry**

Benchmark Data

Our methodology of obtaining filter speed is to measure the number of clock cycles it takes to move between two breakpoints, the first placed at the first instruction in the ISR, the second placed at the end of the ISR. As such, all measured clock cycle values include the 146 cycles it takes to perform input/output, and the approximately 30 cycles for the filter selection (implemented as switch-case statements). TODO Fig tabulates the results for each filter and optimisation level:

|  |  |  |  |
| --- | --- | --- | --- |
| **Optimisation Level** | **None** | **-o0** | **-o2** |
| **noncircular\_filter** | 14425 | 10274 | 1262 |
| **circular\_filter\_modulo** | 16477 | 12509 | 1421 |
| **circular\_filter** | 10859 | 7734 | 1829 |
| **circular\_filter\_symm** | 8871 | 6280 | 993 |

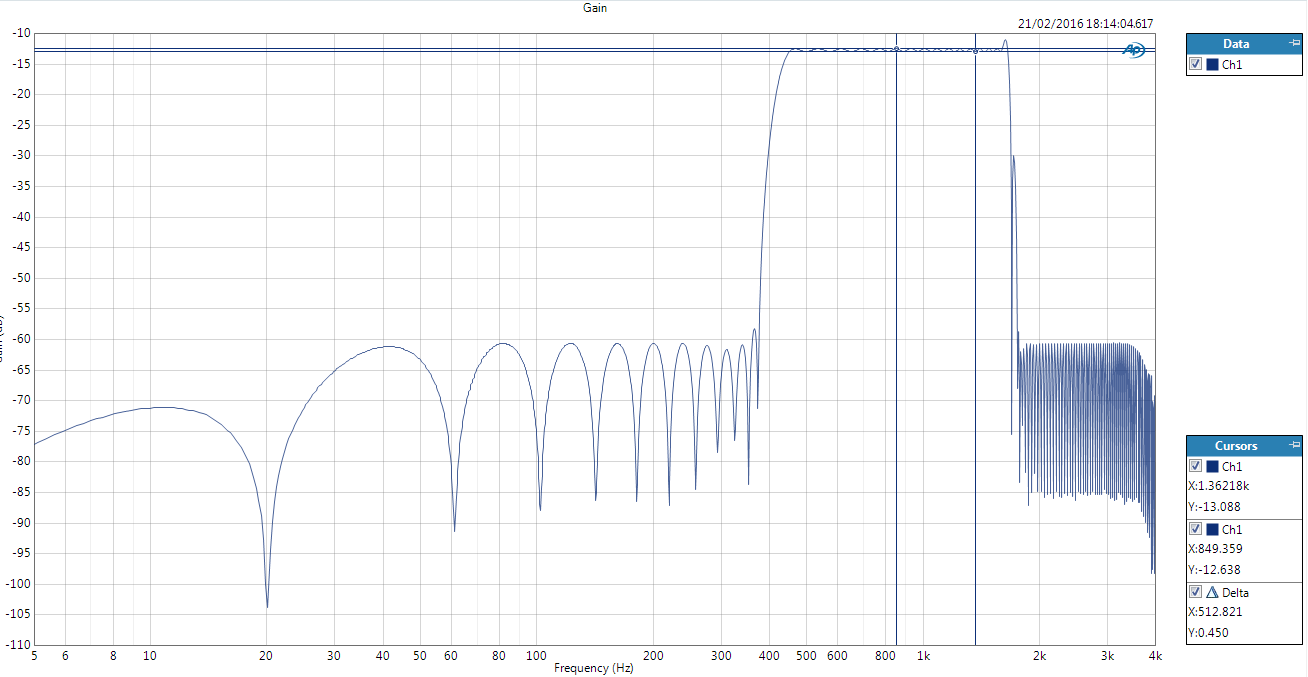
**TODO Fig. Clock cycles for ISR for each corresponding filter and optimisation level**

In the context of our filter code, level -o0 performs two key optimisations: control graph simplification and allocation of variables to registers. This is due to the for-loops taking up most of the processing overhead, and the use of variable sum as an accumulator in said for-loop. This accounts for the significant reduction in processing time from no optimisation to -o0.

Level -o2 most notably performs software pipelining, which in the context of the TI C6000 enables up to 8 multiply-accumulates to occur in parallel. This is observed in the data above across -o0 to -o2 for all filters, where the clock cycle count drops by a factor of 4 to 8. Additionally, -o2 performs loop unrolling and conversion of array references in loops to incremented pointer form. The latter is a contributing factor to circular\_filter\_modulo beating circular\_filter in terms of speed in -o2: the suboptimal usage of array references in circular\_filter\_modulo is optimised to be on pair with circular\_filter. We can also infer that the loop unrolling optimisation contributes to noncircular\_filter being faster than both circular\_filter\_modulo and circular\_filter in -o2: the repeated branching caused by having two for loops (one for convolution, one for shifting the buffer) has been optimised away with -o2 compilation.

Network Analyser

The magnitude response of our filter is as shown in TODOfig. The many ripples in the stopbands and passband are expected, and are typical of a high-order FIR filter. Observing the passband, the gain is centered approximately at -12 dB, despite the filter requirement being 0 dB. Referring to the DSK circuit diagram in Appendix C from the lab handout, we notice an attenuation of ¼ on each input port. This corresponds to a gain of -12.09 dB, accounting for the aforementioned disparity. The required stopband attenuation of 48 dB from the passband is observed in the plot as well; at -60 dB on the plot, it is 48 dB lower than the passband gain of -12 dB.



TODOfig: Magnitude (dB) versus frequency (Hz) plot of our filter, range 5 Hz - 4 kHz

The same plot zoomed in on the passband is shown in TODOfig. The passband ripple is 0.45 dB, which deviates slightly from the 0.4 dB required by the specifications. 