Red Pitaya

Thesis

Raphael Frey
Noah Hüsser

August 14, 2017
Version 0.0.1

Contents

List of Figures

List of Tables

List of Listings

Introduction

Measuring equipments has ever been very expensive. If one wants precise devices it can go up to hundreds or thousands of Francs. Modern, cheap FPGAs can often, if combined with a proper frontend, replace those expensive dedicated devices. The frontend consists of dedicated ADCs and DACs and oftentimes some analog filters. Those chips also have become a lot cheaper in recent years aand are thus way more accessible. Extreme equipment, for example with high sampling rates, in contrary can still not be replaced easily. This project aims at arming an FPGA board with logic that can record, filter and store electrical signals with adjustable sampling rates up to 125 MHz. To complement the hardware part a software that runs on an embedded Linux on the integrated ARM core will be coded such that it can transmit the recorded samples over the network. To read and visualize the samples at the other end of the network, another piece of software will be crafted, that will run on the users computer. The primary focus lies on enabling students to analyze audio signals. Since audio signals contain very low frequencyies only up to tens of thousands of Hz they can be sampled with rather low frequencies and thus making FPGAs an excellent choice. An FPGA not only shines in price competitiveness but also in flexibility. This means that the logic is not fixed in silicon and can be adjusted after the product has already been delivered.

For this project a RedPitaya board is used. It is ideal since it features a fast (125 MHz) 14-bit ADC. This poses a huge amount of data, that is not even required for audio signals. Furthermore it is not realisticly possible to transmit this huge amount of data over the network.

Thus the first primary target of this thesis is to decimate the recorded signals. To avoid aliasing effects that emmerge when decimating a signal appropriate filters have to be designed and impemented that are able to attenuate unwanted signal frequencies.

The second primary target of this thesis is the design and implementation of a software-based oscilloscope. This is a graphical user interface that communicates with the RedPitaya board and visualizes the recored samples. Traditional measuring equipment always has a built in display that visualizes the data on the device itself. This uses up a lot of space and provides very low flexibility. Since it can be assumed that every engineur is equipped with a computer, said device should be used to display the signals. This keeps cost and required space down. The data is then transmitted via the network which will be interfaced by both the users computer and the RedPitaya board.°

[**?**]

Part I

Project Report

CHAPTER 1

Theoretical Background

This chapter will present a brief synopsis on some aspects of digital data acquisition from an analog source and the processing of that data, and how those issues pertain to our project. It is not intended to be a comprehensive treatise on the subject but shall serve as a short refresher. At its end, the reader should have sufficient insight to understand the basic motivation of our project from a theoretical point of view.

TODO: references to more comprehensive literature.

1.1 The Digital Signal Processing Chain

Digitally acquiring a signal generally requires at least the following steps:

- Passing the signal through an analog low-pass filter.
- Sampling and quantizing the filtered signal.

The resulting sequence of values can then be further digitally processed. The necessary building blocks for this process are portrayed in Figure 1.1.

Of particular interest for our application is what happens in the ADC. The quantization process converts a value-continuous signal into a value-discreet one, with its resolution being a specification of the ADC which is being used. As an example, the ADC in our system has a resolution of 14 bits, meaning it can divite its valid input range into 16 384 values. Given an input range of 2 V V_{pp}, that equates to a resolution of roughly 122 µV (in theory). This quantization process is the source of what is generally known as *quantization noise*. For more on the topic, see `TODO`.

TODO: Check numbers. Give references for further reading.

Besides the quantization, the other step happening in the ADC is sampling; a time-continuous signal is converted into a series of time-discreet values. The time between those

Figure 1.1: The basic building blocks of the DSP chain from its analog input to its digitally processed output. From left to right: The analog low-pass filter (*LP*), the analog-to-digital converter (*ADC*), and an arbitrary digital signal processing system for further processing of the ADC's output (*DSP*).

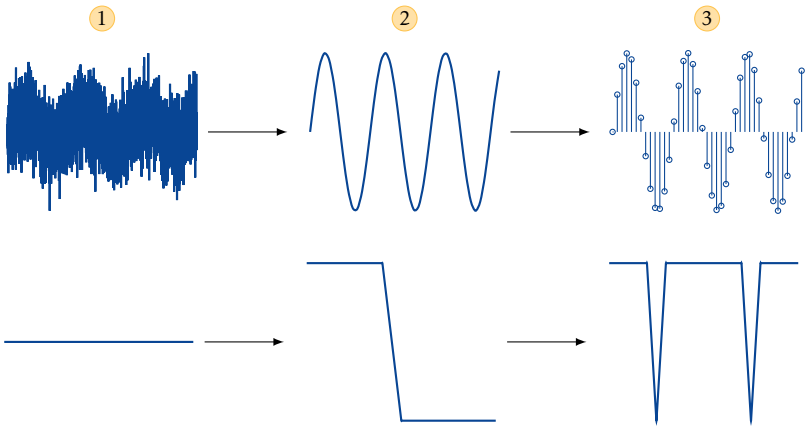


Figure 1.2: Simplified time-domain (top) and frequency-domain (bottom) view of the signal at different stages on its way through the DSP chain. The circled numbers correspond to the stages as outlined in Figure 1.1. Stage 1 is the signal before passing through the input low-pass filter, with a significant amount of high-frequency noise. The low-pass filter removes any frequency components above $f_s/2$ in an ideal scenario (in reality, it merely attenuates them, as we will see later), resulting in the signal at stage 2.
After having been filtered, the ADC samples and quantizes the signal, yielding a sequence of values, as schematically portrayed in the rightmost picture for stage 3. Note that due to the sampling process, the spectrum of the filtered signal is repeated at intervals of f_s. This is the source of the issue of *aliasing*.

values is known as *sampling time*, its inverse is the *sampling frequency*. Note that usually these are constant, at least during the time where the signal is measured. This need not strictly be the case in theory though. In our system, this sampling frequency is a fixed property of the ADC, and is 125 MHz.

The sampling step lies at the core of the problem our project intends to address: *aliasing*. Therefore, we will take a closer look at a few consequences of the sampling process, and how they are relevant to this project.

Descriptively, the sampling process can be thought of as looking at a signal at specific points in time and capturing its value. Mathematically, this amounts to multipliying the signal with a series of Dirac pulses in the time domain, and convolving with a series of Dirac pulses in the frequency domain[1]. . This convolution in the frequency domain lies at the heart of the problem of aliasing, because it results in the incoming signal's spectrum being repeated at intervals of f_s (see also: stage 3 in Figure 1.2). This is no problem as long as the spectrum of the incoming signal fits within the boundaries set by this repetition. But if the spectrum of the incoming singal is too broad, two or more recurrences of the spectrum will overlap. This effect is highlighted in Figure 1.3.

This overlap results in two primary problems:

- The digital signal may not be unambiguously reconstructable into an analog signal, if that is intended.
- Frequencies may occur in the digital signal stream which are not actually present in the original signal. This problem is often referred to as the *folding back* of frequency components. See Figure 1.4 for an illustration of how this might look.

[1] *Pro memoria*: A series of Dirac pulses in the time domain has as its spectrum a series of Dirac pulses as well.

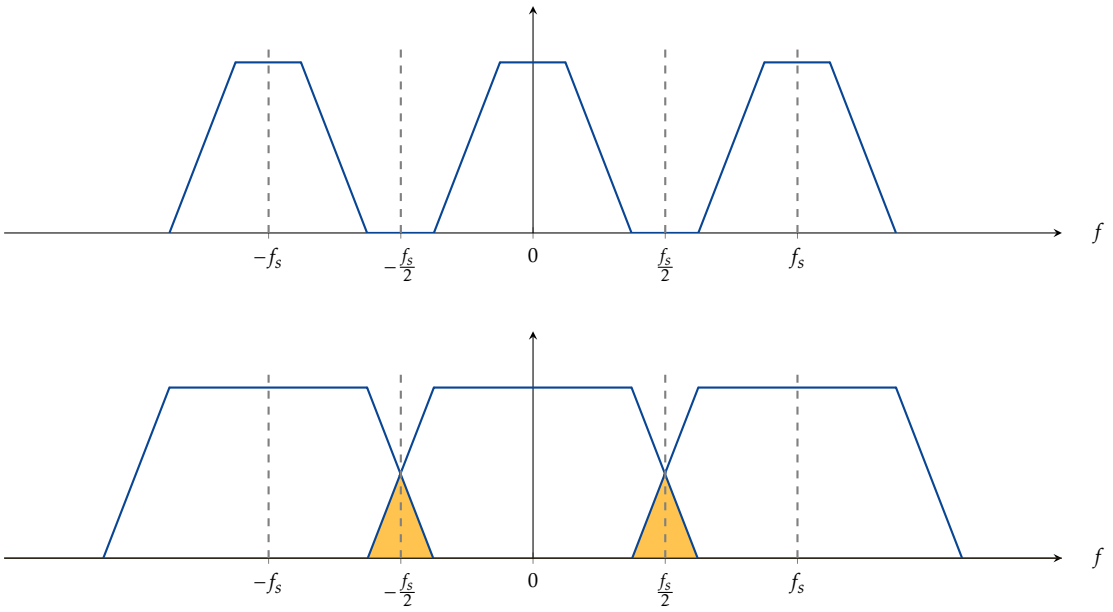


Figure 1.3: Simplified view of a signal which does not produce aliasing between its recurrences in the frequency spectrum (top), contrasted with a signal whose frequency band has components above half the sampling frequency, resulting in aliasing; its spectral copies overlap (highlighted areas in the bottom plot).

This problem is of particular interest to our application, as we will see later.

Once a signal has left the ADC and is handed down the DSP chain for further processing, the primary problem becomes one of resources, particularly in real-time applications. In most systems, the available hardware is a fixed constraint, and depending on what sort of processing is to be conducted on the digital data stream, the available resources may or may not suffice.

If available resources are found to be insufficient for real-time processing of the data stream, one may choose to

- not process the data in real time,
- reduce the complexity of the computations, or
- reduce the amount of data to be processed through *downsampling* of the signal.

The last case is the route which is chosen in our application. The main constraint on the Red Pitaya is that the data being generated cannot be moved off the device in real time, and the device itself does not offer sufficient storage for capturing a meaningful amount of data which can then be moved onto another device for further processing at a later point. Therefore the amount of data must be reduced before it can be moved off the device to a computer for viewing or further processing.

TODO: Amount of data being generated on the PITA.

Because downsampling a signal is in essence nothing more than the sampling of a signal which has already been sampled, a lot of the considerations which are valid for the step from an analog to a digital signal as outlined above are either very similar or even identical. Specifically, the same considerations for aliasing still apply: If the signal which

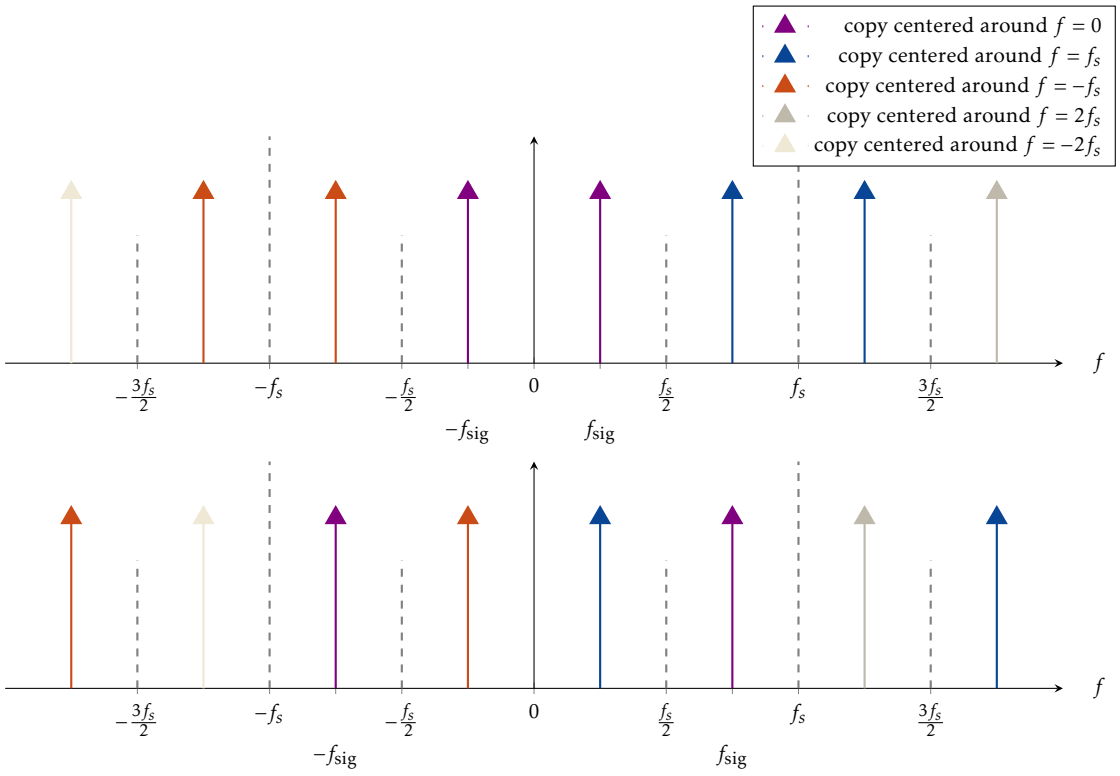


Figure 1.4: Example of two harmonic signals being sampled. In the top plot, the signal's frequency is below half the sampling frequency and there is no aliasing. The signal can be reconstructed without error. In the bottom plot, the signal's frequency is above half the sampling frequency. Consequently, the copies of the signal's frequency spectrum centered around the sampling frequency and its negative alias back into the band between $-f_s/2$ and $f_s/2$. If this signal is reconstructed, the resulting signal would have a frequency of $f_s - f_{\text{sig}}$ instead of f_{sig}.

is to be downsampled has frequency components above $f_{s,downsampled}/2$, aliasing will occur. And since the signal coming out of the ADC has the original signal's (filtered) spectrum recurring at intervals of its sampling frequency, this is always the case.`TODO: Correct?`

Therefore, the sampled signal must be filtered through a low-pass filter before being downsampled, just as the original analog signal was low-pass filtered before being passed into the ADC. In light of the signal to be downsampled being a *digital* signal instead of an analog one, that low-pass filter must naturally be a digital filter as well. Designing such a digital low-pass filter is the core mission of this project.

The key properites of such a filter which are relevant to our application are

- its transition band width (filter steepness), and
- its aliasing attenuation.

The aliasing attenuation refers to the fact that when a filter is being used for downsampling, copies of its frequency response will be created at intervals of the lower sampling rate (analogous to the sampling process producing spectral copies of a signal when sampling an analog signal).

The stopband components of these copies overlap with the intended passband, leading to aliasing (it should be noted that this phenomenon is also present in the case of the analog intput filter for the DSP chain). This effect is portrayed in Figure 1.5. The top plot shows the filter's frequency response along with four copies to illustrate the overlap effect. The bottom plot shows the aliasing effect more clearly by removing the spectral copies and retaining the aliased regions.

The overlapping parts of the spetrum are composed of spectral copies both to the right and left side of the original. Therefore, the aliased regions are alternately flipped around the vertical axis. This creates in essence the same effect as if the paper were folded along multiples of the lower sampling rate over the frequency range of the central copy (in the case of our example: $0.2f_s$, $0.4f_s$, $0.6f_s$ and $0.8f_s$) like an accordion. This is where the term *folding back* originates.

1.2 Digital Filters

Digital filters can be distinguished by several characteristics; common ways to categorize them are by topology, impulse response and their frequency response. There are two commonly used types of digital filters: Infinite impulse response (IIR) filters and finite impulse response (FIR) filters. Another important class of filters are cascaded integrator-comb (CIC) filters, however, in the strictest sense they are a special class of FIR filters rather than an entirely new type of LTI system [**?**]. While our system uses FIR and CIC filters, a brief overview of IIR filters is still presented here, for the sake of completeness.

1.2.1 IIR Filters

Infinite impulse response filters are so named because their imulse response continues into perpetuity, never reaching zero. In practice, the response usually comes sufficiently close to zero at a certain point that it can be considered zero for most intents and purposes.

IIR filters have feedback paths, resulting in a filter response equation with non-trivial denominator components. Their basic building blocks are delay elements, multipliers and adders.

$$H(z) = \frac{\sum_{k=0}^{N} b_k \cdot z^{-k}}{1 + \sum_{i=0}^{M} a_i \cdot z^{-i}} \tag{1.1}$$

TODO: Move sum limits above and below sigma sign.

IIR filters generally require a lower order (and therefore fewer resources) to approximate a certain frequency response specification than FIR filters do (particularly the constraint of a narrow transition band), but this comes at a cost: IIR filters have a non-linear phase response; linear-phase responses can only be approximated. Furthermore, IIR filters are not guaranteed to be BIBO stable due to their feedback paths.

Some of the generally used types of IIR filters are:

- Butterworth filter: Named after the British engineer and physicist Stephen Butterworth (1885 – 1958), who first described it in 1930. Characterized by a very flat passband (no passband ripple).
- Chebyshev filter (type I and II): Named after Russian mathematician Pafnuty Chebyshev (1821 – 1894). They are steeper than Butterworth filters, at the cost of suffering from ripple in the passband (type I) or stopband (type II).

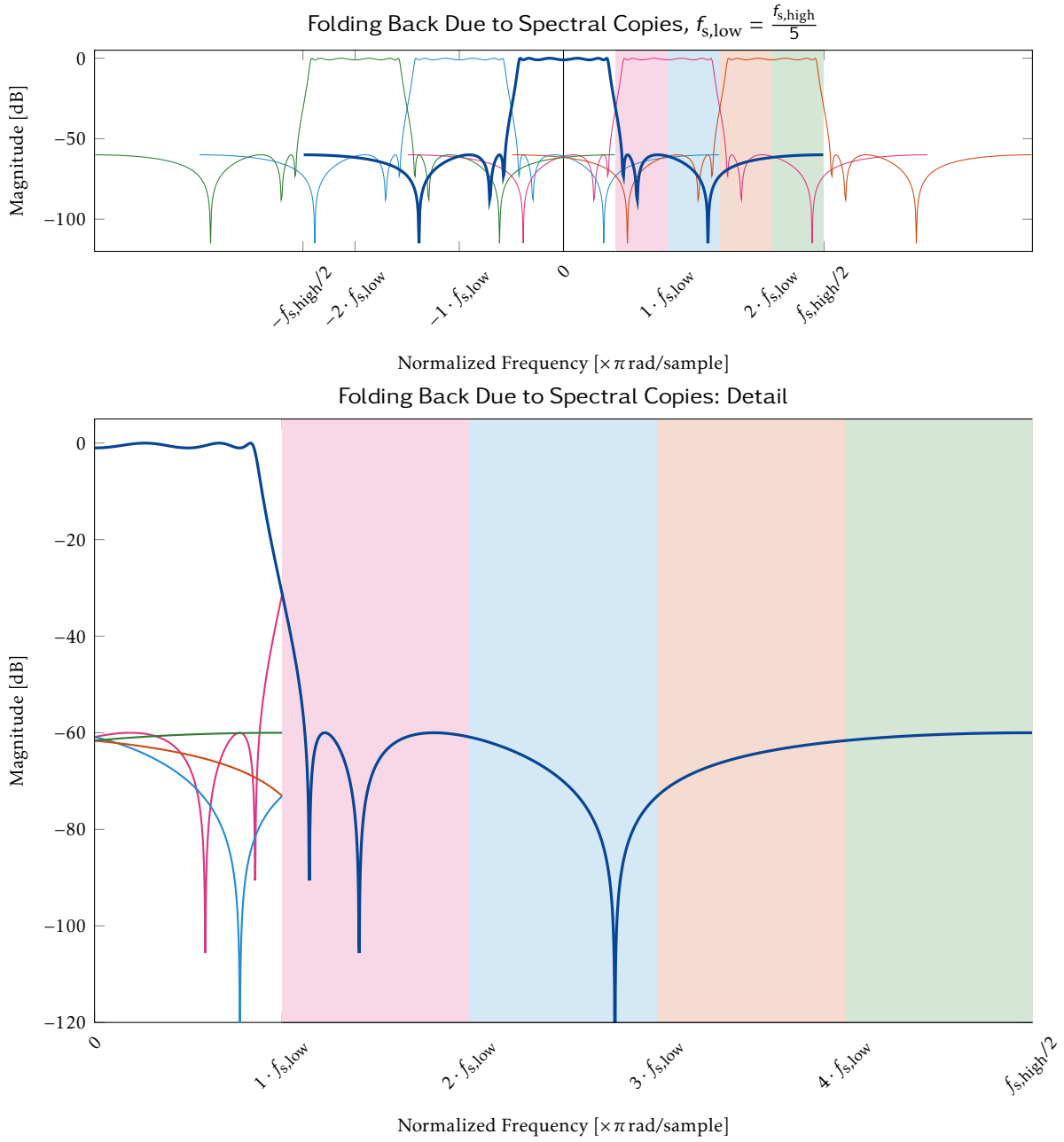


Figure 1.5: The phenomenon of folding back when downsampling, illustrated for a lowpass IIR filter with with a cutoff frequency of $0.2 \cdot f_s$ for a downsampling ratio of $R = 5$. The downsampling process produces copies of the filter's frequency response at intervals of the lower sampling frequency, visible in the top plot. The stopbands of these copies then overlap with the intended passband. The bottom plot shows a close-up view with the spectral copies for clarity.

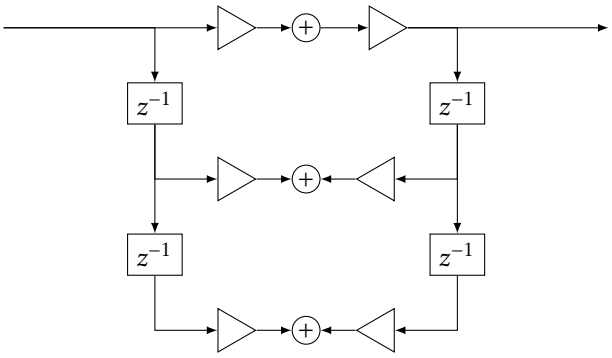


Figure 1.6: Example of an IIR filter topology for a biquad

- Bessel filter: Named for the German mathematician Friedrich Bessel (1784 – 1846). Optimized to have a maximally linear phase response in order to minimize the distortion of signals passing through the filter.
- Elliptical filters: Also known as Cauer filters, after the German mathematician Wilhelm Cauer (1900 – 1945), or Zolotarev filter, after Russian mathematician Yegor Zolotarev (1847 – 1878). Characterized by equiripple in the bassband and stopband and a very narrow transition band compared to other filters of the same order.

TODO: Check correct dashes for years.
Digital IIR filters are often designed by way of the bilinear transform.

1.2.2 FIR Filters

FIR filters are characterized by an impulse response which decays to zero in finite time (see Figure 1.9, unlike IIR filters. The filter response is characterized by Equation 1.2:

$$H(z) = \sum_{k=0}^{N} b_k \cdot z^{-k} \tag{1.2}$$

FIR filters have several advantages:

- They are inherently BIBO stable because they lack feedback paths.
- They can be easily designed to have a linear phase response, preventing signal distortion due to different group delays for signal components of different frequencies.
- The shape of their frequency response can be very finely tuned. This makes them ideally suited for certain purposes, such as compensation filters (see Section 1.2.3).
- Implemenation is usually rather straightforward.

TODO: Correct?

Their main disadvantage is that due to the lack of feedback, they generally require comparatively high filter orders for narrow transition band widths. Illustratively, this can be understood by the following considerations:

- The frequency response of an ideal low-pass filter is the brick wall filter, i.e. a rectangle.
- The inverse Fourier transform of a rectangle is a *sinc* function, which is infinitely long.

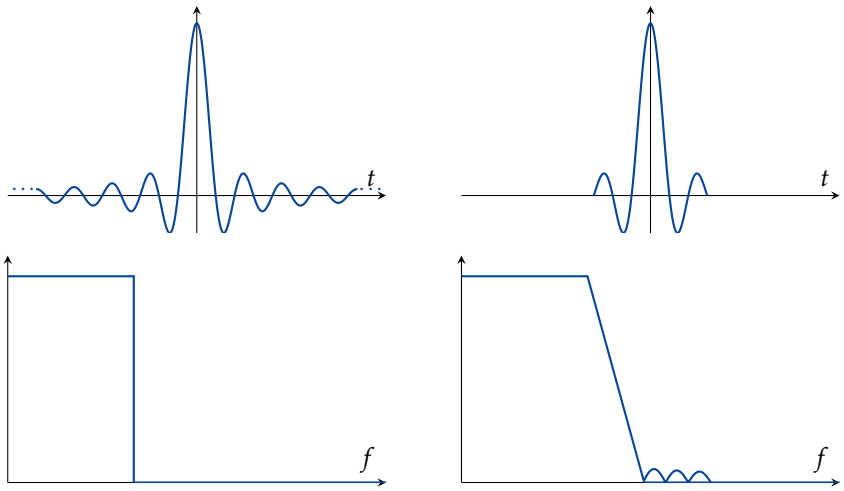


Figure 1.7: The effect of truncating a *sinc* function in the time domain on its spectrum (simplified)

- Therefore, the impulse response of the ideal brick wall filter would have an infinite number of taps.
- Truncation of the number of taps leads to a deviation of the filter's frequency response from the brick wall filter. As the number of taps (and therefore the FIR filter's impulse response) is reduced, its frequency response deviates more and more from the brick wall filter, resulting in a flatter transition between the passband and the stopband as well as the introduction of ripple.

This process is illustrated in simplified form in Figure 1.7.

The FIR filter's transition band width is particularly important for our application in order to reduce aliasing effects, as will be shown later TODO: actually show later.

TODO: Equation for order estimation.

Designing FIR filters is usually performed by specifying certain desired characteristics of the filter's frequency response. Figure 1.8 shows one possible way of doing this for FIR filters by specifying four constraint parameters:

- pass band ripple: A_P
- stop band attenuation: A_{St}
- pass band edge frequency: F_P
- stop band edge frequency: F_{St}

The resulting transition band width F_{Tb} is the difference between the pass band edge frequency and the stop band edge frequency, and serves as a useful indicator of how many coefficients (i.e. resources) the filter will end up using. Narrower transition bands tend to require a higher filter order, and therefore more resources. Coefficient counts of several hundred are not uncommon for steep FIR filters.

Other sets of constraint parameters can be used to design filters, but these are the ones used in this project, therefore the emphasis on them.

Figure 1.9 shows the resulting impulse response (coefficient set) for a FIR filter designed by using the four above mentioned parameters, with values given by Equations 1.3 through 1.6 handed to one of Matlab's FIR filter design algorithms.

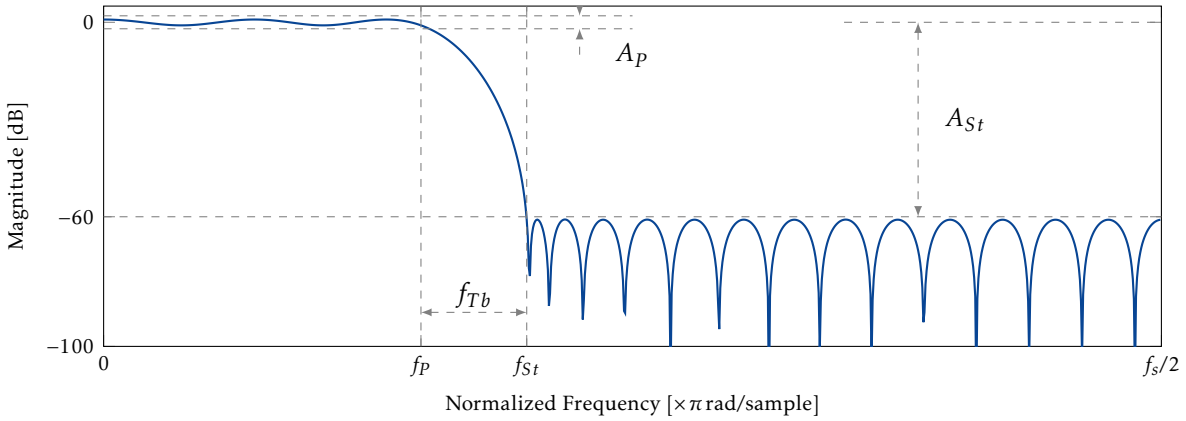


Figure 1.8: Specifications in the frequency domain and the resulting filter's frequency response as designed by Matlab.

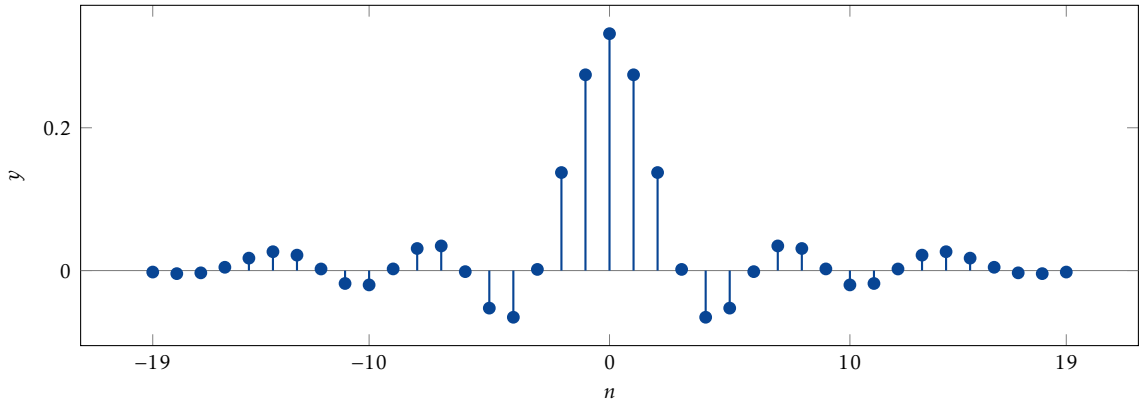


Figure 1.9: Impulse response (coefficients) for the filter from Figure 1.8 with the parameters as given by Equations 1.3 through 1.6 passed to one of Matlab's FIR filter design algorithms, resulting in a set of 39 coefficients. Note that the coefficients to the left and right of these values are zero, hence *finite* impulse response filters.

$$A_P = 2\,\text{dB} \tag{1.3}$$
$$A_{St} = 60\,\text{dB} \tag{1.4}$$
$$F_P = 0.3 \cdot f_s \tag{1.5}$$
$$F_{St} = 0.4 \cdot f_s \tag{1.6}$$

Figure 1.10 shows one possible topology for implementing a FIR filter, the so-called direct form. As can be seen, the basic building blocks of a FIR filter are delay elements, multipliers and adders, same as for IIR filters.

One particular form of a FIR filter is the so-called half-band filter. Half-band filters are used for downsampling by a ratio of $R = 2$. They are characterized by a point-symmetric

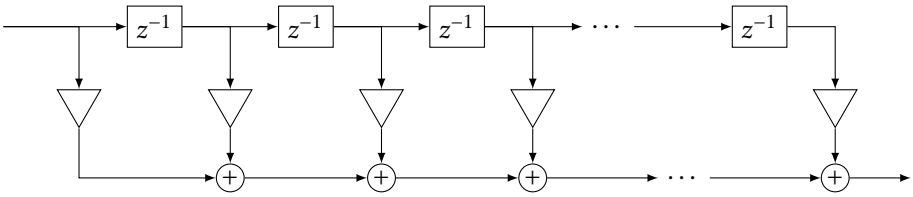


Figure 1.10: One possible topology for a FIR filter (direct form)

frequency response across the $(f_s/4, 0.5)$ point. Their advantage lies in the efficiency of their coefficient structure: Each second coefficient is zero, and all the non-zero coefficient are symmetrical around the center of the impulse response. For higher downsampling rates, multiple half-band filters can be cascaded. Figure 1.11 shows the amplitude frequency response and the coefficient set of an example filter.

1.2.3 CIC Filters

CIC filters were first introduced in 1981 in [**?**] by Eugene B. Hogenauer. They can be implemented both as decimation filters (reduction in sampling rate) and interpolation filters (increase in sampling rate).

1.2.3.1 General Description

A CIC filter is a cascade of integrator and comb stages, with either a sampling rate compressor (in case of a decimator) or a sampling rate expander (in case of an interpolator) between the integrator and comb sections. A single integrator stage is shown in Figure 1.12, and Figure 1.13 shows a single comb stage in feedforward form. Figure 1.14 shows a complete CIC filter with three stages.

The integrator stages have a transfer function of

$$H_I(z) = \frac{1}{1 - z^{-1}} \tag{1.7}$$

The comb stages run at the reduced frequency of f_s/R and have the transfer function

$$H_C(z) = 1 - z^{-RM} \tag{1.8}$$

where M is the *differential delay*, one of the filter's design parameters.

The transfer function of a complete CIC filter (referenced to the high sampling rate f_s) consisting of N stages is deduced by multiplying the transfer functions of the N cascaded integrator and comb stages, yielding

$$H_{CIC}(z) = H_I^N(z) \cdot H_C^N(z) = \frac{\left(1 - z^{-RM}\right)^N}{\left(1 - z^{-1}\right)^N} = \left[\sum_{k=0}^{RM-1} z^{-k}\right]^N \tag{1.9}$$

Looking at the last form of the CIC filter's transfer function, it becomes evident that it is in essence a FIR filter with unitary coefficients. Of particular note is the fact that this is so despite each stage having feedback or feedforward paths and the integrator stages having poles at $f = 0$ (i.e. the integrators by themselves are not in fact BIBO stable, even though the complete system is). The fact that the resulting filter has no poles can be intuitively understood by looking at the frequency responses of the integrator and comb stages, and finally their cascade (see Section 1.2.3.2).

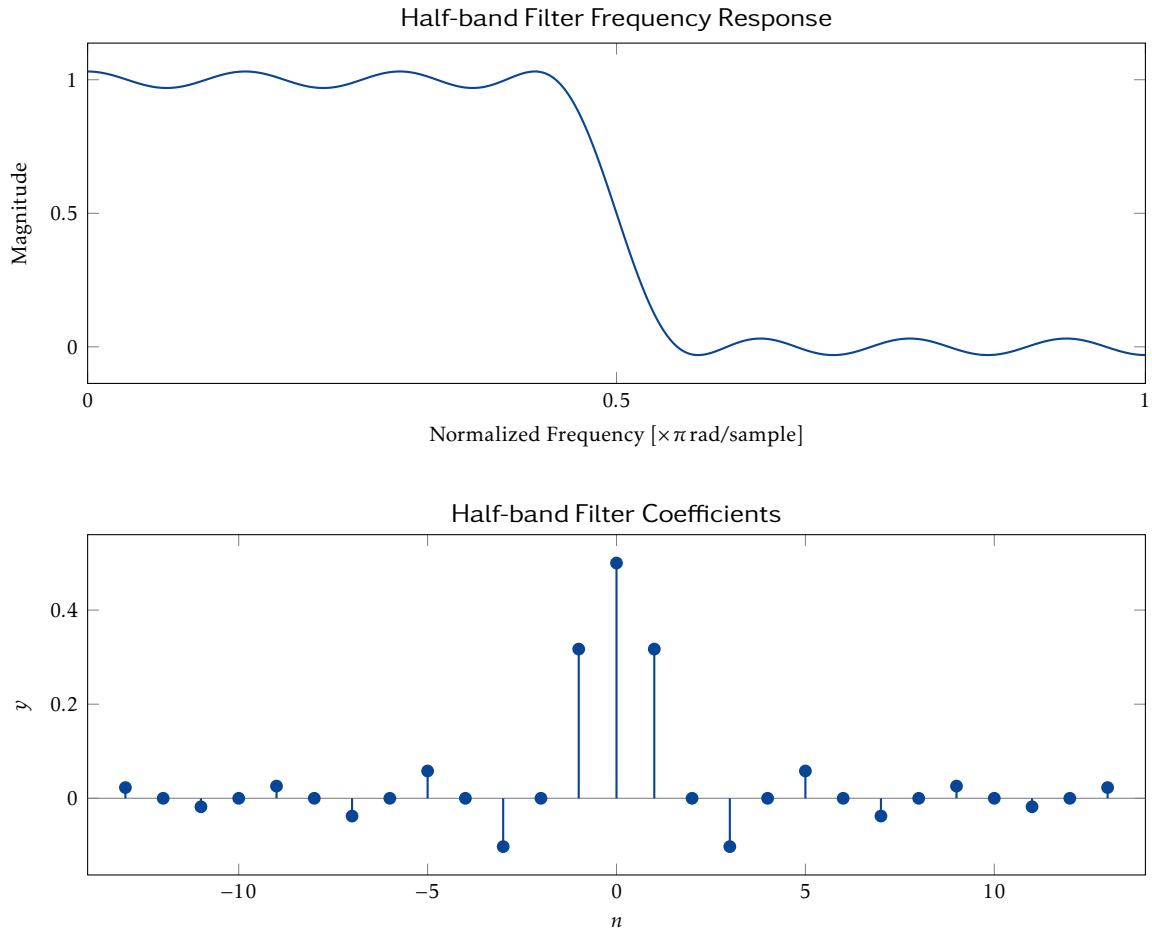


Figure 1.11: The frequency response and coefficient set of a half-band filter. The frequency response is the amplitude plotted linearly, not the magnitude plotted logarithmically, in order to emphasize the symmetry. The coefficient set is symmetrical around its midpoint. Also, every second coefficient outside the central peak is zero.

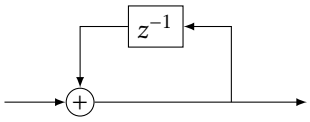


Figure 1.12: A single integrator stage

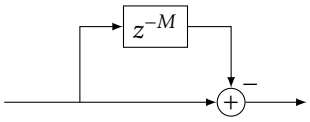


Figure 1.13: A single comb stage in feedforward form

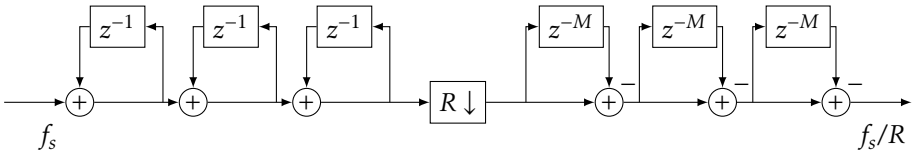


Figure 1.14: CIC decimation filter topology with three integrator and comb stages

CIC filters are well-suited to large reductions in sampling rates because they are very economical in their resource usage. This economy is based on six primary factors (see also [**?**]):

- The filter requires no multipliers.
- There are no filter coefficients to store.
- The amount of storage needed for intermediate results is reduced by running the comb stages at a lower sampling rate. A conventional FIR filter topology implementing the same transfer function would require more resources for storing its intermediate results because the entire filter would run at the incoming sampling rate.
- The topology of the filter has a high degree of regularity; consisting of two primary building blocks. This lends itself well to optimization.
- The control logic can be kept simple.
- The same filter design can be used for a large range of rate change factors R, requiring minimal adaption in circuitry. This effect can be seen in the frequency response plotted in the top plot of Figure 1.16.

However, CIC filters do suffer from some drawbacks. The two primary ones are:

- For large rate change factors R, the register growth of the filter can become very large. TODO: see section blabla
- A CIC filter has only three design parameters determining its frequency response: Rate change factor R, differential delay M, and the number of stages N. The amount of fine-tuning which can be conducted on the filter's frequency response is therefore extremely limited (more in Section 1.2.3.2).

As can be seen in Equation 1.7, the integrator stages have unity feedback coefficients. In the case of CIC decimators, the registers of the integrators will therefore suffer from register overflow. This causes no harm as long as two conditions are fulfilled:

- The filter's implementation is based on two's complement or another number system allowing wrap-around between its most positive and most negative numbers.
- The maximum magnitude which is expected at the output is within the range of that number system.

A numerical example to demonstrate this effect and better explain the inner workings of a CIC filter can be found in Appendix A.1, starting on page 76.

1.2.3.2 Frequency Characteristics

This section presents some of the more important frequency characteristics of the CIC filter. We will start with some considerations about how the integrators and comb sections interact in the frequency domain to create the CIC filter's frequency response.

As shown in the topmost plot in Figure 1.15, an integrator is in essence a lowpass filter, with a pole at $f = 0$. A comb filter is a filter which attenuates one specific frequency component along with its multiples (in a notch comb filter; there is also the inverse concept of a peak filter which only lets a certain frequency and multiples of it pass). It is also evident that comb filters have no poles (a fact which can be deduced from Equation 1.8 as well, of course).

Cascading integrators and combs results in a frequency response like the one in the bottom plot from Figure 1.15. The integrator's pole at $f = 0$ compensates for the comb section's zero at the same location, leading to a significant, but finite, DC gain of the CIC filter.

One drawback of CIC filtes is that they have no clearly defined passband as such. Rather, their frequency response starts dropping off right as the frequency axis goes beyond zero. This effect (also referred to as *passband droop* or *passband attenuation*) is visible in the magnified section of the bottom plot in Figure 1.15 and in Figure 1.17. Since CIC filters lack a clearly defined transition band edge, defining the frequency band which is to actually be used, i.e. the actual passband, is a design decision and can vary even when using the same filter, depending on the application.

The amount of passband droop is constant for a given product of the differential delay M and the cutoff frequency f_c, where f_c is a fraction of the lower sampling rate (i.e. a fraction of the first lobe's width). Figure 1.17 highlights this effect for two different filters. Table A.3 in Appendix A.2 on page 81 contains a list with more values for some common configurations.

Because of the passband droop, a CIC filer by itself is rarely a viable solution. Rather, it is generally deployed as the first element in a chain of filters, where the later stages are FIR filters. Due to the CIC filter's frugality in terms of resource usage, it is ideally suited as an initial stage, where the most samples per time need to be processed. The fact that FIR filters need to perform many more computations (and more complex ones) per sample is then no longer as much of a problem, since those FIR filters run at lower sampling frequencies and have therefore many more clock cycles available to compute each output. Also, because the frequency response of a FIR filter can be very finely tuned to a desired profile, they can be used to compensate for the CIC filter's passband droop; this is generally known as a *CIC compensation filter*. TODO: altera application note

Another effect which must be taken into consideration when designing CIC filters is the amount of aliasing which occurs from the stopband into the passband. A region of width f_c above and below each Mth null is folded back into the filter's passband. This effect is highlighted in Figure 1.18. The gravity of this effect depends on the width of the cutoff frequency f_c as well as the differential delay M. Table TODO in Appendix TODO contains some values for common ranges for M and f_c.

As mentioned, the CIC filter has only three design paramterers: Its rate change factor R, the differential delay M and the number of stages N. The influence of these paramters on the CIC filter's frequency response is portrayed in Figure 1.16. Some things of note are:

- Increasing R increases the amount of nulls as well as the overall gain of the filter.
- Increasing M also increases the number of nulls as well as the filter's gain. Note that for CIC decimators, the region around every Mth null is folded back into the passband.
  For practical purposes, M is usually set to 1 or 2, see [**?**].
- Adding more stages leads to a high increase in filter gain, since N occurs in the exponent of the filter's transfer function. It does not, however, change the number or placement of the nulls.

1.2.3.3 Compensators

In order to achieve a flat passband, the CIC filter's attenuation in the frequency range observed in Figure 1.17 can be compensated with a filter whose frequency response has the opposite shape. Operated in a cascade (see Section 1.3), the two filters create a frequency response with a flat passband and a sharop drop-off into the stopband. Figure 1.19 shows an example of such a system, with frequency responses of a CIC filter, its compensator, and the resulting cascade.

The compensation filter not only serves to compensate for the passband, but is also responsible for the transition band width of the cascade. Due to their flexibility, FIR fil-

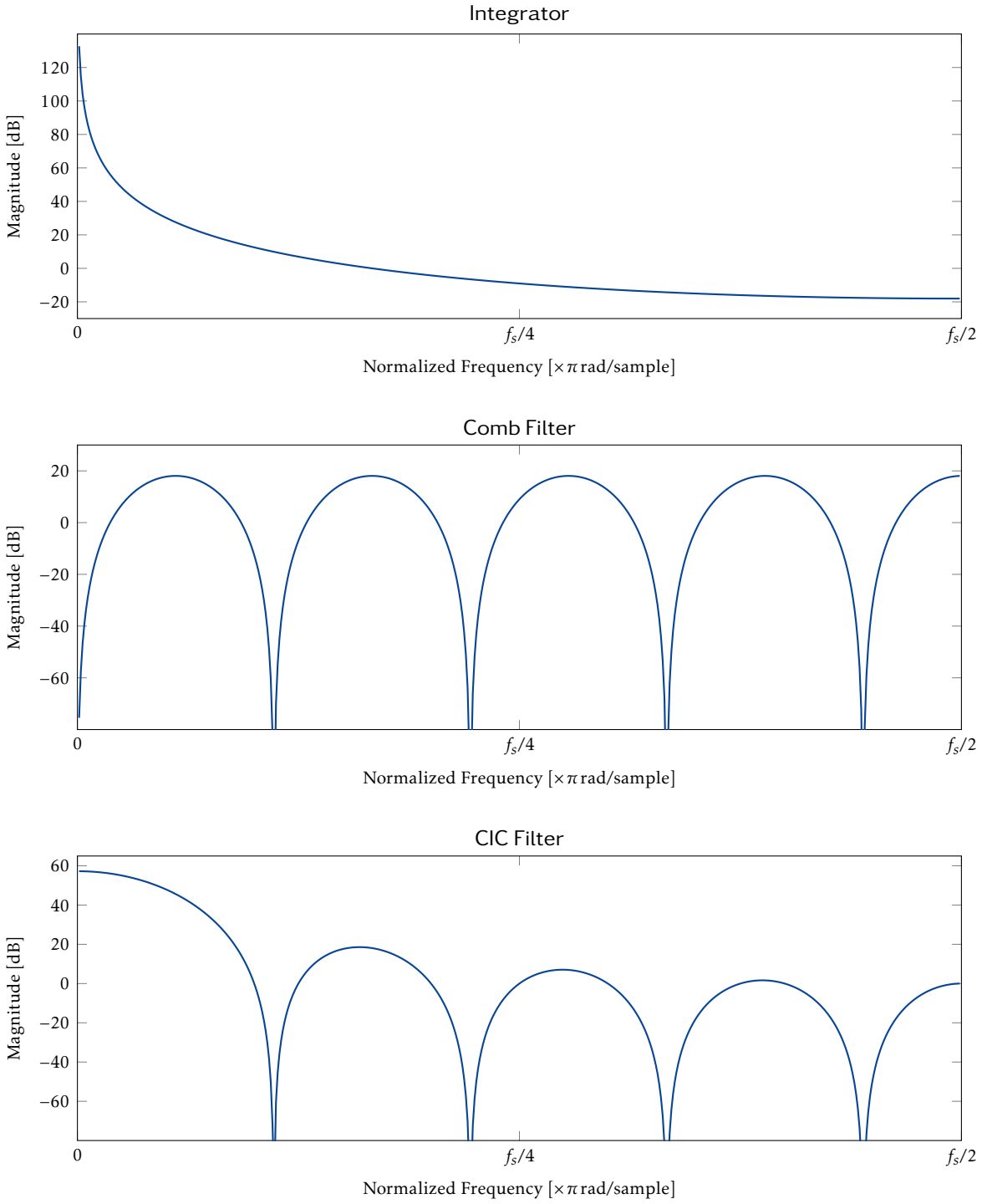


Figure 1.15: Frequency responses for integrators, combs and their combination into a three-stage CIC filter with a rate change factor of 9 and a differential delay of 1. Note that 4.5 lobes fit into the plot for the comb filter, due to $R \cdot M = 9$ (the order of the comb filter). The enlarged box shows a close-up of the CIC filter's passband droop.

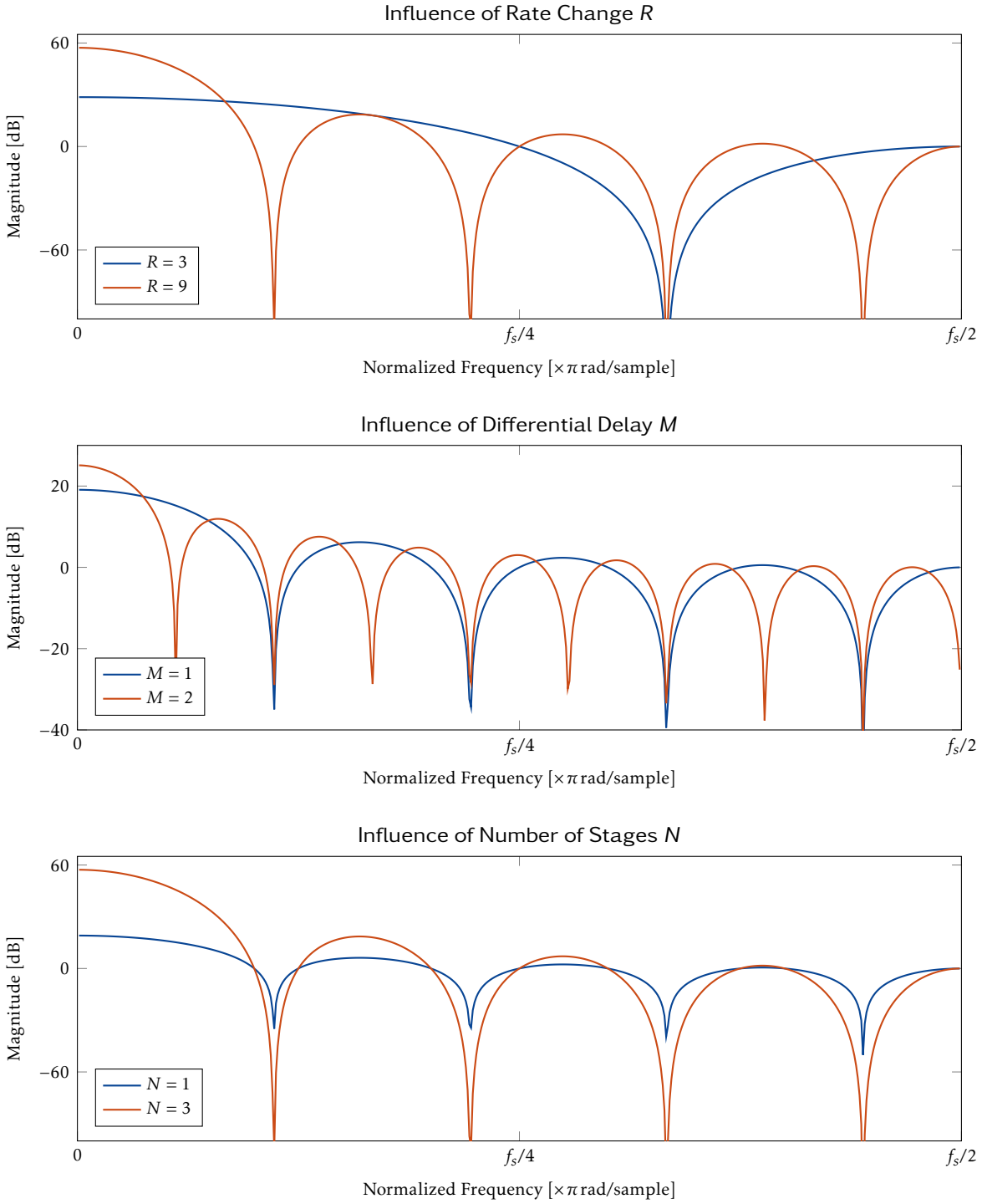


Figure 1.16: The influence of the design paramters R, M and N on a CIC filter' frequency response. Inreaseing R and M, respectively, leads to an increased number of nulls, as visible in the top two plots, as well as an increase in the DC gain. Adding more stages does not change the location of the nulls, but does add significant DC gain.

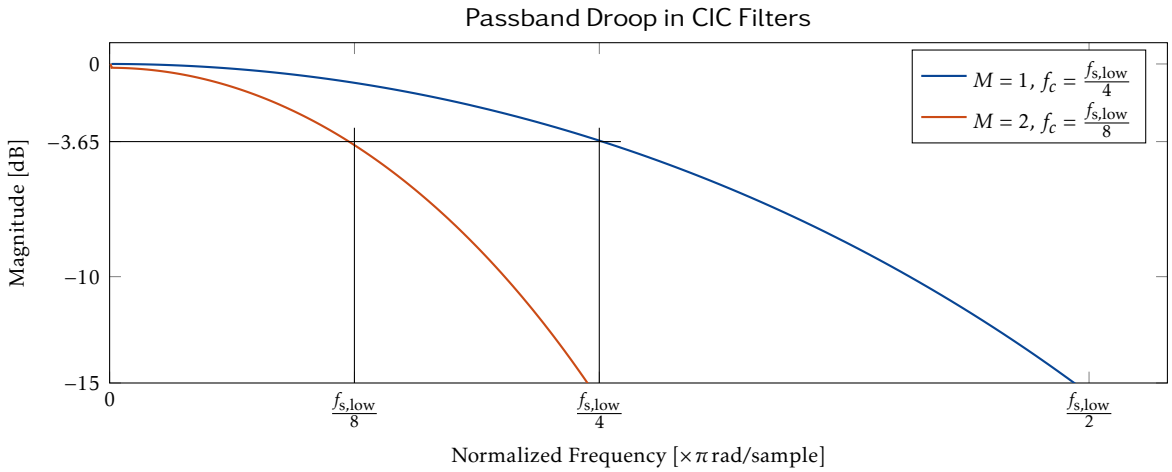


Figure 1.17: Passband attenation for two CIC filters with $R = 9$, $N = 4$ and $M = 1$ and $M = 2$, respectively. The attenuation is identical for the bandwidth-differential delay product, which is 1/8 for both of these configurations. The attenuation is −3.65 dB in both cases; the value can be found in Table A.3 on page 81.

ters are generally employed for this purpose. As mentioned in the previous section, the sharpness of their transition band can be controlled by adjusting their kernel size. If the compensator has more filters coming after it in the overall filter chain, its transition band need not be very narrow. A filter with a few dozen coefficients in size is often sufficient in such cases (the filter used for the example in Figure 1.19 has 50 coefficients).

The design of CIC compensators is usually left up to software algorithms, for example with Matlab. For a more detailled introduction to the topic, including example code and more thorough explanations, Altera's Application Note from [**?**] is warmly recommended.

1.2.3.4 Register Growth

As shown by Hogenauer in [**?**], the maximum register growth is

$$G_{\max} = (R \cdot M)^N \tag{1.10}$$

The most significant bit $B_{\max}$ of the output register as well as for all stages (both the integrators and the comb stages) of the filter is determined to be

$$B_{\max} = \lceil N \log_2 RM + B_{\text{in}} - 1 \rceil \tag{1.11}$$

where B_{in} is the bit width of the input register. For high rate change factors, these values can become very large. A filter with three stages, a differential delay of 1, a rate change of 128 and an input width of 16 bits yields 36 bits output width at full precision.

1.2.3.5 Errors Due to Truncation and Rounding

In practical cases, it is often not feasible to retain full precision; in such situations, either truncation or rounding may be used at each filter stage to reduce register widths and keep resource usage within certain limits. For this purpose, it is necessary to know the system

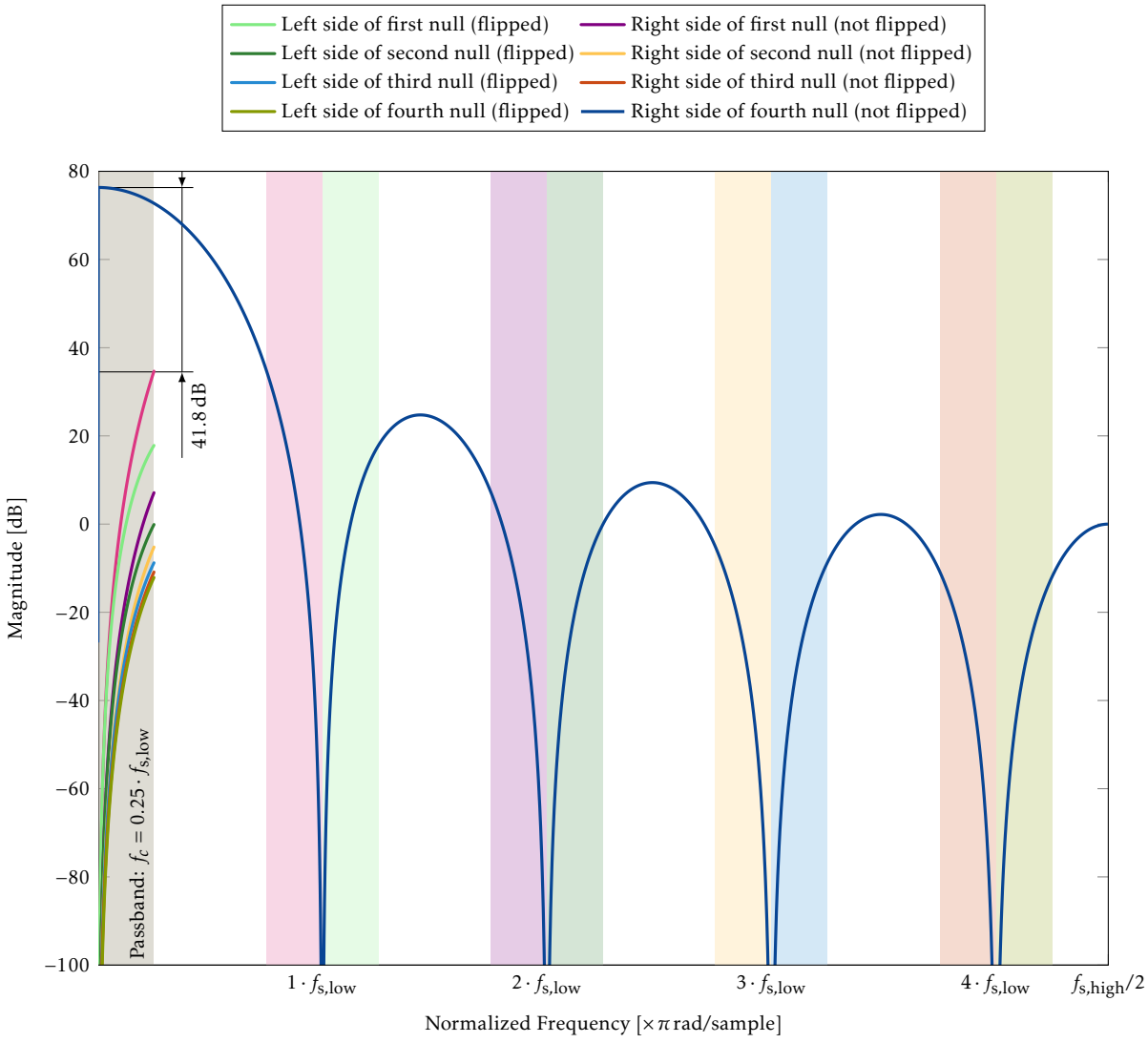


Figure 1.18: Passband aliasing for a CIC filter with $R = 9$, $N = 4$ and $M = 1$ and a cutoff frequency of $f_c = 0.25$, referenced to the lower sampling frequency $f_{s,low}$. The region of width f_c around every Mth null is folded back into the passband. The regions beyond that are of course folded back as well, but since we choose to arbitrarily limit the passband, those regions are not of interest to us. The resulting passband aliasing attenuation is 41.8 dB, as indicated in Table A.4 on page 81.

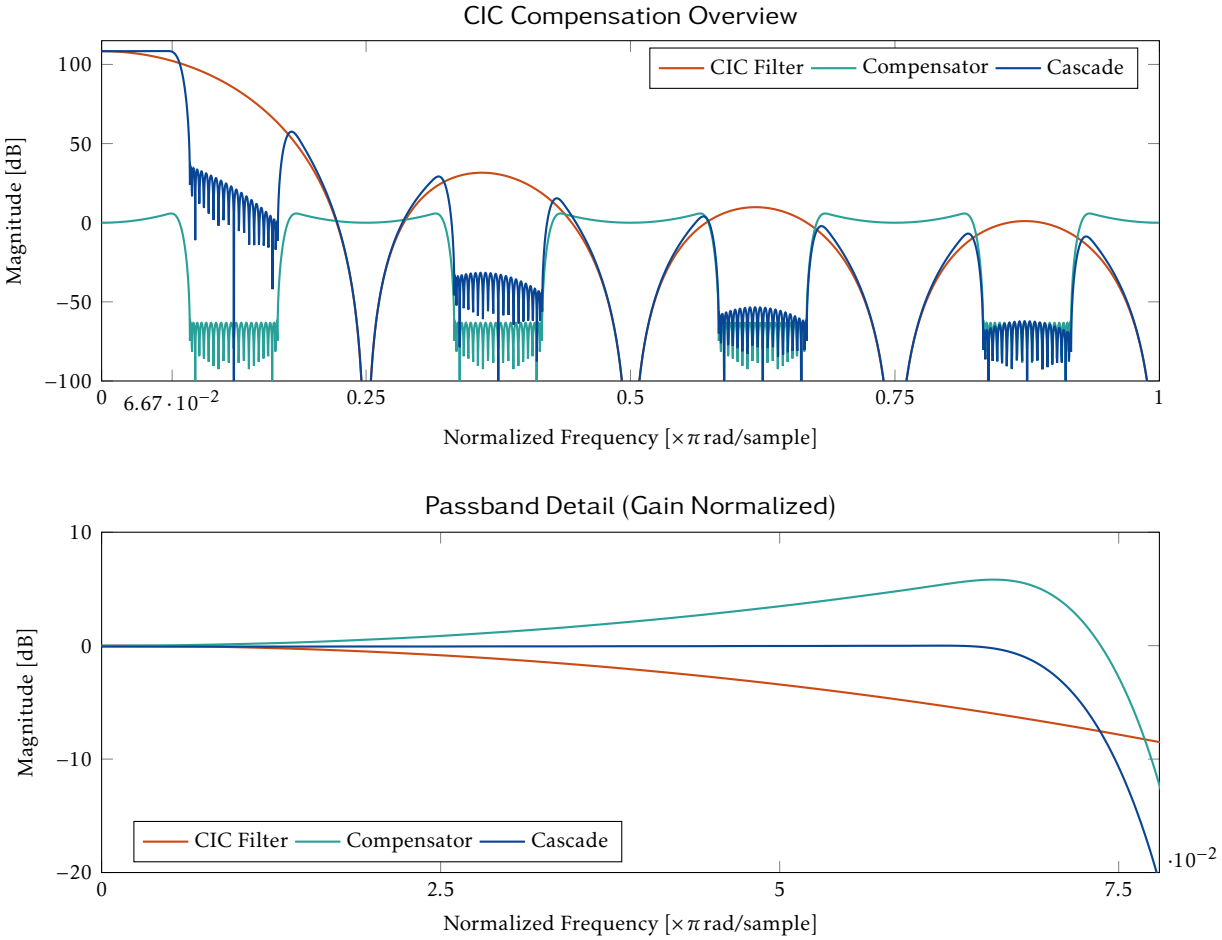


Figure 1.19: Frequency behavior of a CIC filter, its compensator, and the cascade of the two. Note the spectral copies of the compensator around the nulls of the CIC filter, i.e. the multiples of its outgoing sampling rate.

function from the jth stage up to and including the last:

$$H_j(z) = \begin{cases} H_I^{N-j+1} H_C^N = \displaystyle\sum_{k=0}^{(RM-1)N+j-1} h_j[k] z^{-k} & j = 1, 2, \cdots, N \\ H_C^{j-N} = \displaystyle\sum_{k=0}^{2N+1-j} h_j[k] z^{-kRM} & j = N+1, \cdots, 2N \end{cases} \tag{1.12}$$

where

$$h_j[k] = \begin{cases} \displaystyle\sum_{l=0}^{\lfloor k/(RM) \rfloor} (-1)^l \binom{N}{l} \binom{N-j+k-RMl}{k = RMl} j & = 1, 2, \cdots N \\ (-1)^k \binom{2N+1-j}{k} j & = N+1, \cdots 2N \end{cases} \tag{1.13}$$

are the impulse response coefficients. These functions are also derived by Hogenauer in [**?**].

In a filter with N stages, there are $2N+1$ error sources in the case of limited precision: Each stage, and the output register. Each error source is presumed to have white noise characteristics, i.e. its noise is uncorrelated to its input as well as other error sources. The error at the jth source is assumed to have a uniform probability distribution with a width of

$$E_j = \begin{cases} 0 & \text{without truncation or rounding} \\ 2^{B_j} & \text{otherwise} \end{cases} \tag{1.14}$$

where the number of bits discarded at the jth error source is B_j. The mean of this error is

$$\mu_j = \begin{cases} \frac{1}{2}E_j & \text{for truncation} \\ 0 & \text{otherwise} \end{cases} \tag{1.15}$$

and the variance comes out to

$$\sigma_j^2 = \frac{1}{12}E_j^2. \tag{1.16}$$

The total mean error at the filter's output due to the jth stage is

$$\mu_{T_j} = \mu_j D_j \tag{1.17}$$

where

$$D_j = \begin{cases} (RM)^N & j = 1 \\ 0 & j = 2, 3, \cdots, 2N \\ 1 & j = 2N+1 \end{cases} \tag{1.18}$$

is the *mean error gain* for the jth error source. Note that only the first and the last error source contribute to the filter's mean error at the output. This is because the sum of the impulse response coefficients is zero for all other stages. Consequently, whether one chooses to truncate or round is without consequence except in the case of the first and last error sources. In an analogous manner, the total variance computes to

$$\sigma_{T_j}^2 = \sigma_j^2 F_j^2 \tag{1.19}$$

where

$$F_j = \begin{cases} \sum_k h_j^2[k] & j = 1, 2, \cdots, 2N \\ 1 & j = 2N+1 \end{cases} \tag{1.20}$$

is called the *variance error gain* for the jth error source.

We can now compute the global mean error and variance of the filter:

$$\mu_T = \sum_{j=1}^{2N+1} \mu_{T_j} = \mu_{T_1} + \mu_{T_{2N+1}} \tag{1.21}$$

$$\sigma_T^2 = \sum_{j=1}^{2N+1} \sigma_{T_j} \tag{1.22}$$

These equations are used to calculate the properties of the CIC filter as deployed in our design. TODO: see section blabla

1.2.3.6 Compensators

1.2.3.7 Summary

In conclusion, the key properties of CIC filters are:

- They can be implemented both as decimators and interpolators.
- Neither multipliers nor storage for coefficients are needed.
- CIC decimation filters have a high gain, leading to significant register growth. Truncation or rounding can be used to limit the resource usage, both at the filter's output and internally.
- The three design parameters are the rate change R, the differential delay M and the number of stages N.
- The presence of passband droop requires a compensation filter to achieve a flat passband response.

Compensation Filter?

1.3 Multi-Stage Filter Designs

At first sight, the most obvious way to implement a downsampling system might appear to be to design one filter for each desired rate change factor. However, this would be highly impractical. Instead, multi-stage designs are usually used in practice. An in-depth discussion of their advantages and drawbacks was offered by Crochiere and Rabiner in [**?**]. A few aspects of multi-stage filter design which are relevant to our application shall be presented here.

To reduce aliasing effects in the passband, it is generally desirable to keep the width of the transition band roughly constant in relation to the width of the passband (visible in the filter's flank in Figure 1.5). As the downsampling ratio increases and the passband width decreases, the transition band therefore becomes progressively narrower, necessitating higher filter orders in a single-stage design.

This effect is illustrated in Figure 1.20, comparing two filters with a transition band 1/5 as wide as the passband for downsampling ratios of 2 and 4. The filter for $R = 4$ requires 60 coefficients, compared to 30 coefficients for the filter designed for $R = 2$. The other specifications (passband ripple, stop band attenuation) are identical. As an extreme case, a filter designed by Matlab with the same parameters for a downsampling ratio of $R = 625$ is 8860 coefficients in size.

Using multi-stage designs helps to avoid the need to implement filters with such large kernels. When cascading filters, it is the last stage of the chain which defines the overall passband and transition band width. The same overall transition band in absolute terms can be achieved with smaller filters in multi-stage designs, because a filter's transition band width is relative to the sampling rate at which it is running. This is shown in the bottommost plot in Figure 1.20.

Since cascading multiple filters does increase coefficient count (and storage), one might be inclined to think that not much has been won. Indeed, the overall number of coefficients is identical for both filters in Figure 1.20 (though, this obviously need not be so in other examples). However: Merely 30 multipliers and 29 adders (those of the first filter in the cascade) run at the incoming sampling frequency in the case of the cascade, while the components of the second filter run at half that. In the case of the single-stage filter, all its 60 multipliers and 59 adders run at the full sampling frequency. Distributing the calculations over two stages has therefore yielded an overall reduction in needed computation

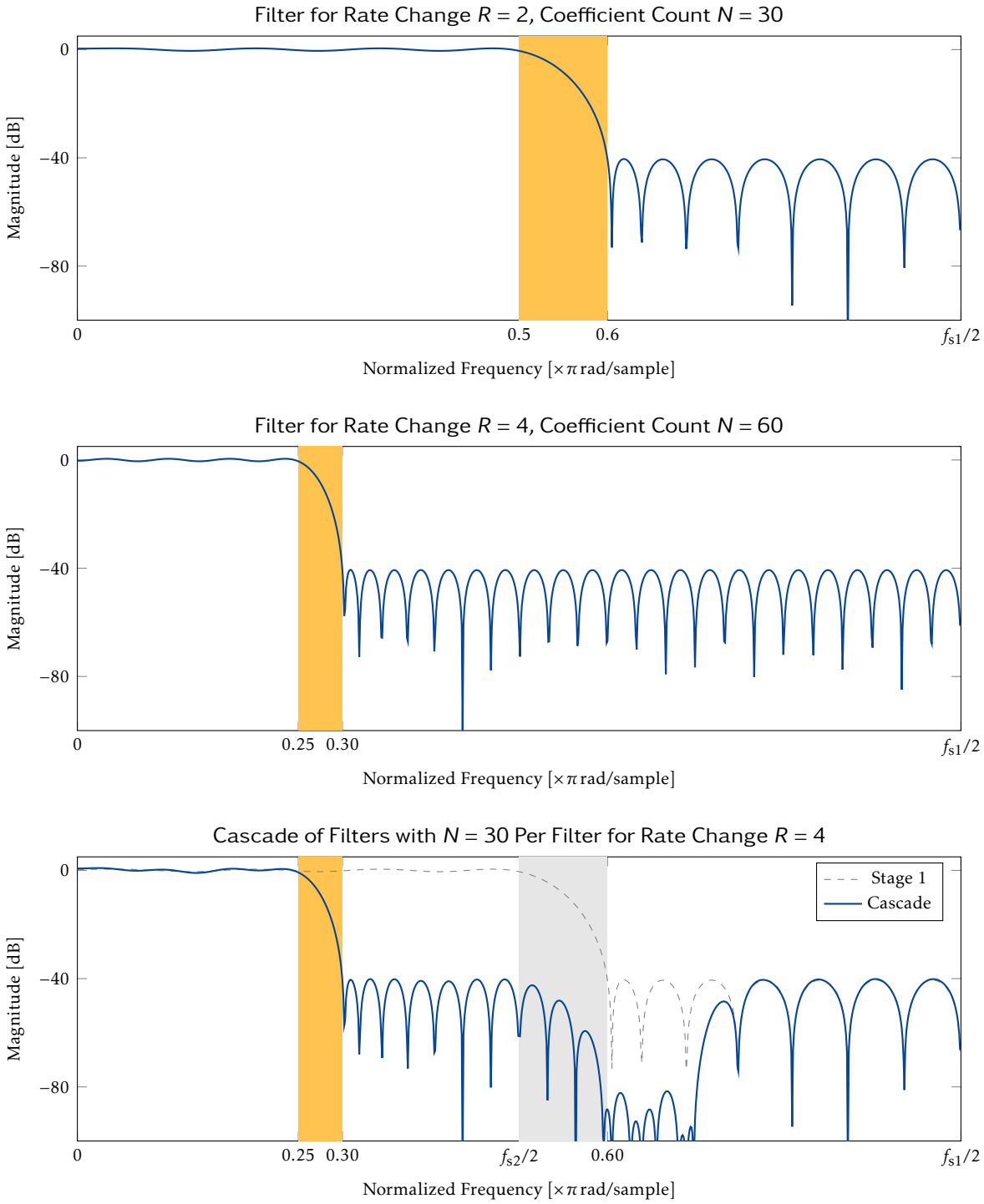


Figure 1.20: Two filters are used here: Both have a transition band 1/5 as wide as their passbands. The top filter is designed for a downsampling ratio $R = 2$ and has a coefficient count of $N = 30$. The second filter is designed for $R = 4$ and has 60 coefficients. Cascading two of the top filters into a two-stage design, depicted in the bottommost plot, results in the same overall passband and transition band width as the single-stage design.

power. As rate change factors increase, the benefits of multi-stage designs become even more pronounced [**?**].

Another advantage of cascading filters is that successive stages can be used to shape the overall frequency response, as seen in the case of the CIC compensator in Section TODO:reference. A drawback of cascades is that the ripple in their passband shows additive behavior, so the stages in a cascade of filters have more stringent ripple requirements in the passband then a single-stage. However, the cost for this is usually offset by the advantages of multi-stage designs.

When designing multi-stage filters, it can happen that the transition band of an earlier stage overlaps with the spectral copy of a later stage running at a reduced sampling rate. In that case, the stopband response of the cascade can have peaks exceeding the desired overall stopband attenuation. To prevent this, the following condition must be satisfied:

$$f_{\mathrm{st},1} < \frac{f_{\mathrm{s},1} - f_{\mathrm{st},2}}{R_1} \tag{1.23}$$

Where:

$f_{\mathrm{s},1}$: high sampling rate
$f_{\mathrm{st},1}$: stopband frequency of first filter
$f_{\mathrm{st},2}$: stopband frequency of second filter
R_1 : rate reduction in first filter

Figure 1.21 shows some examples for this condition being broken or fulfilled.

One last effect of note when cascading filters concerns stopband attenuation: When cascading two filters with different stopband attenuations, two things can happen:

- The second filter attenuates more strongly than the first one. This results in peaks above the second filter's stopband attuation in the regions where the spectral copies of the second filter's passband are located. This can be seen in the top plot in Figure 1.22.
- The first filter attenuates more strongly than the first one. In that case, the stopband region of the cascade right next to its transition band is less strongly attenuated than the stopband regions farther away from the edge. This case is shown in the middle plot in Figure 1.22.

Either of these two effects is usually not desired, barring special scenarios. In both cases, the resources invested into the steeper filter's stronger attenuation are wasted by the other filter's weaker stopband attenuation. It therefore makes more sense for the various stages in a cascade to have the same stopband attenuation, in general.

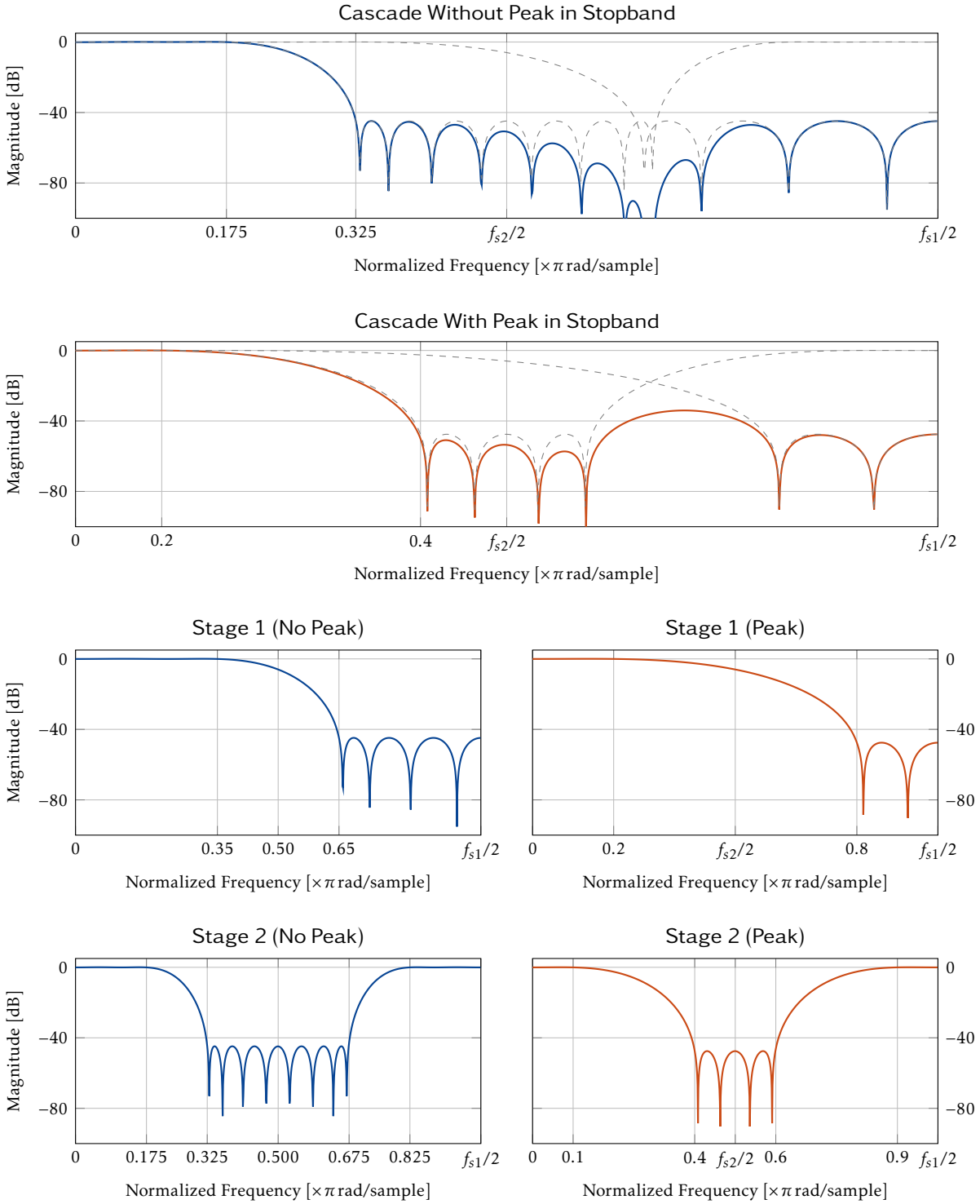


Figure 1.21: Comparison of two cascades: The first cascade (blue) has sufficient distance between the start of its stopband ($0.65 \cdot f_{s1}$) and the start of the transition band of the second stage's first copy around f_{s2} ($0.675 \cdot f_{s1}$). The second cascade (orange) has a peak in its stopband because the transition band of its first stage overlaps with the copy of the second stage ($0.8 \cdot f_{s1}$ vs. $0.6 \cdot f_{s1}$). *Note:* All frequencies are referenced to the high sampling rate f_{s1}.

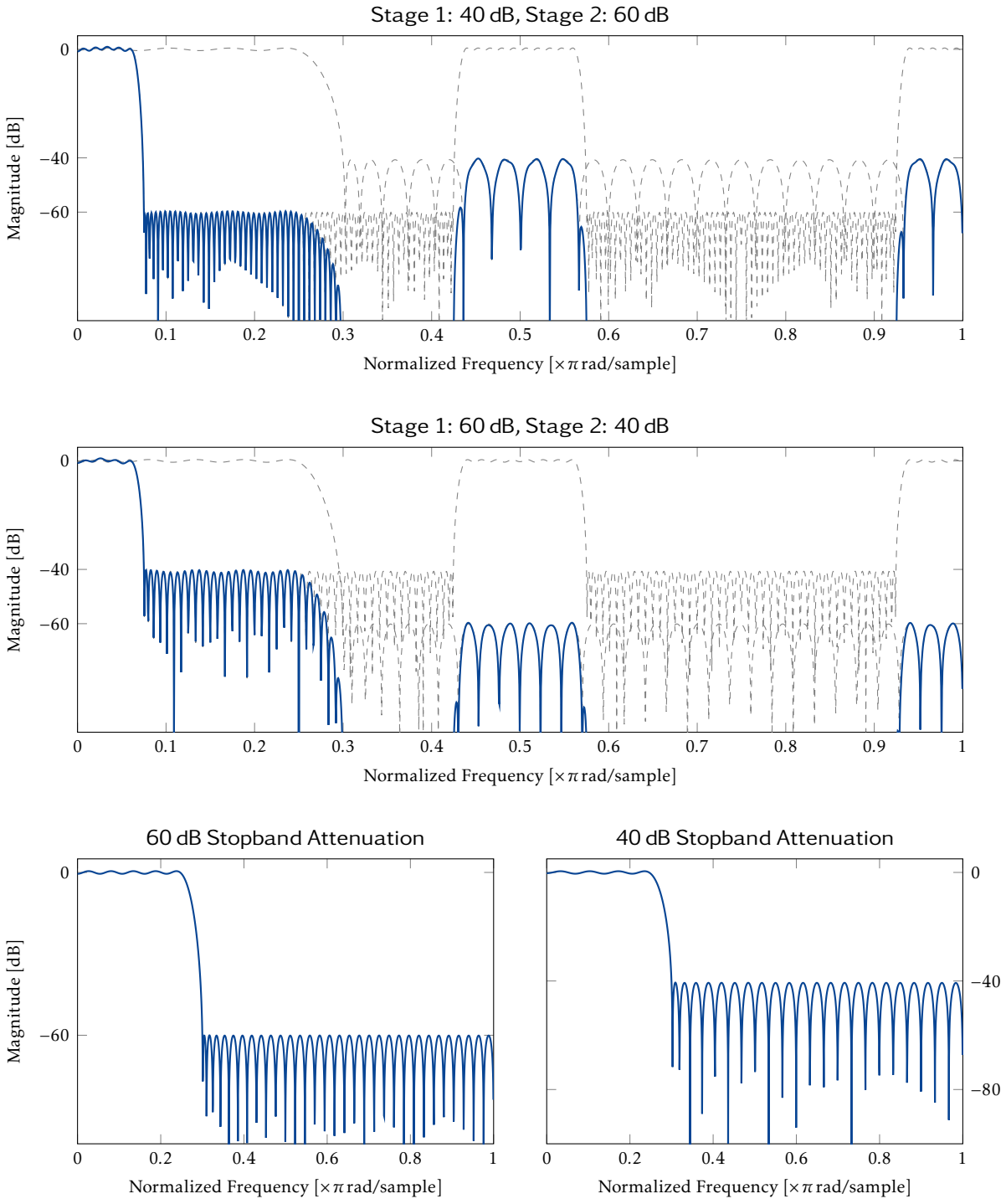


Figure 1.22: Cascading two filtes with different stopband attenuations: If a filter with stronger stopband attenuation is cascaded after a filter with weaker attenuation, the resultant cascade has peaks above the second filter's stopband attenuation (top plot). If the stronger filter is first in the cascade, the drop-off in the stopband right next to the transition band is weaker. The two filters by themselves are shown in the bottom plots.

CHAPTER 2

Mission

This chapter presents a brief summary of the hardware platform and some of its key characteristics which are of interest to us TODO: (impersonal). From that information, the objectives of this project are derived. Possible approaches to reach those objectives are presented and evaluated, and a decision is reached on how to achieve this project's aims. At the end of this chapter, the reader will know what we TODO: (impersonal) intend to do and why.

2.1 The Red Pitaya STEMlab 125-14

First, some key specifications of the hardware are given, to then proceed to the key problems which occur when downsampling with its stock configuration.

2.1.1 Hardware Overview

The device on which this project is based is the Pitaya STEMlab 125-14, pictured in Figure 2.1. STEMlab is a compact measurement instrument (it fits into the palm of a hand) which can replace more expensive devices like oscilloscopes by using a computer for data storage, processing and presentation.

Some of its key specifications relevant to us TODO: (impersonal) are:

- two high-speed analog inputs and outputs via coaxial connectors
- Uses a Linear Technology LTC2145-14 converter [**?**] on those inputs: 14 bits resolution at 125 $MS \cdot s^{-1}$ per channel.
- Xilinx SOC with an FPGA component for data processing on the device itself and two ARM Cortex9 cores for general-purpose tasks
- Ethernet and USB PHYs for data transmission and device control
- Has its own operating system, a GNU/Linux distribution (Ubuntu is used in our TODO: (impersonal) project), running on the ARM cores.
- The software used is open-source and available under [**?**].

More comprehensive documentation can be found at [**?**]. A block diagram with the system's key components is shown in Figure 2.2.

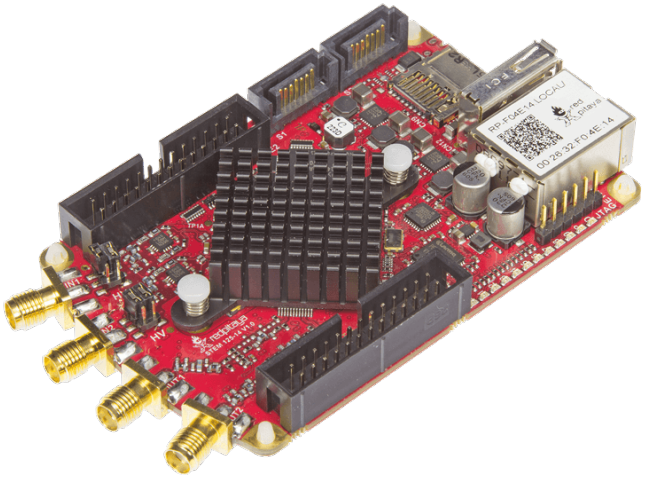


Figure 2.1: Photo of the STEMlab. Source: [**?**]

2.1.2 Downsampling on the STEMlab With Stock Configuration

The STEMlab enables downsampling by powers of 2 in its stock configuration, in a range between $1 = 2^0$ and $65536 = 2^{16}$. The exact manner in which this downsampling process is performed is not easily deducible from public information; the documentation on the internals of the FPGA codebase is rather sparse. However, work conducted by our predecessors has shown that the STEMlab uses a moving averaging filter for this process [**?**].

The moving averager's transfer function is

$$H(z) = \frac{1}{N+1} \sum_{k=0}^{N} z^{-k}. \tag{2.1}$$

A high degree of similarity is immediately recognizable when comparing this to the transfer function of a CIC filter in Equation 1.9 (page 12). Indeed, a cascade of moving averagers would almost yield a CIC filter, save for the gain, which could be easily adjusted after the fact if needed.

A moving averager as a decimation filter is relatively cheap to implement when using only decimation rates of powers of two, as is the case for the stock software of the STEMlab. The weight for each coefficient of z^{-k} will be $\frac{1}{N+1} = \frac{1}{R} = 2^{-m}, m \in \mathbb{N}_0$, meaning the computations can be performed without multipliers by performing a bit-shift operations to the right. The primary disadvantage which results from these rate reduction factors is that the resulting reduced sampling rates are not very "nice" numbers, so to speak, since the incoming sampling rate is 125 MHz, and therefore not a power of 2.

Just like a CIC filter by itself, however, a moving averager also makes for a rather poor lowpass filter, for two reasons: Passband droop and poor stopband attenuation (equivalent

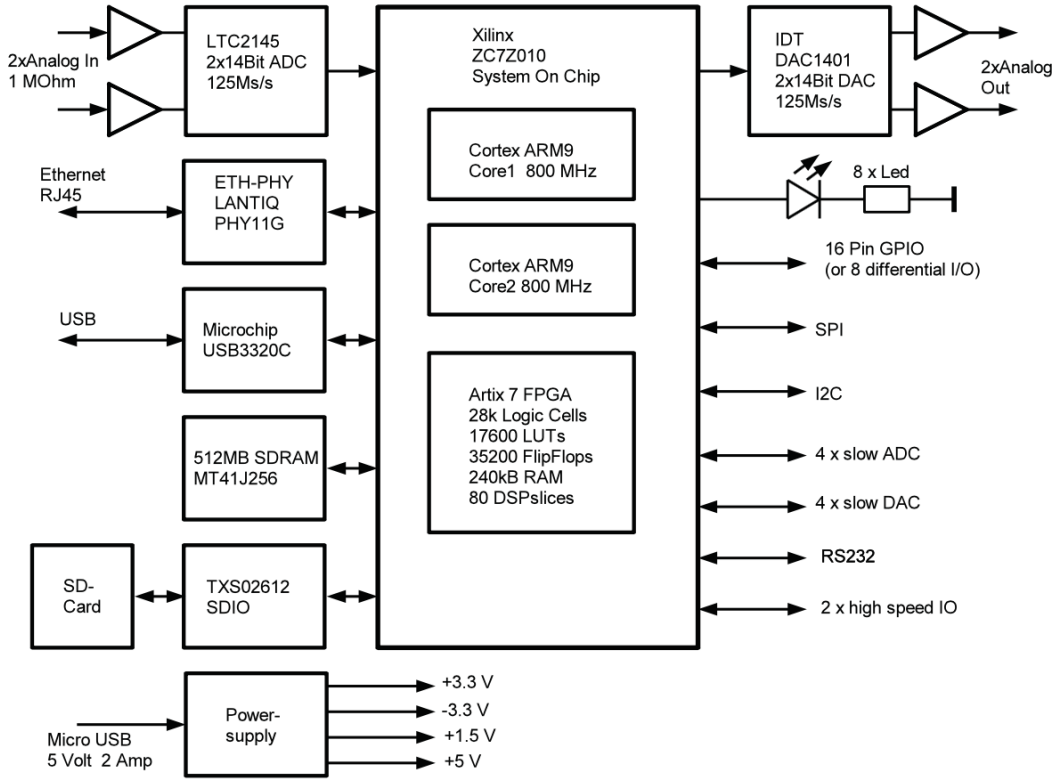


Figure 2.2: Block diagram of the Red Pitaya STEMlab. Source: [**?**]

to a single-stage CIC filter). Figure 2.3 shows the frequency response for the case of $R = 8$ and a signal of 5 MHz. The incoming sampling rate is 125 MHz, the reduced sampling rate is 125 MHz ÷ 8 = 15.625 MHz. Therefore, anything above half that frequency is aliased back into the region below 15.625 MHz ÷ 2 = 7.8125 MHz. In the case of a signal at 5 MHz, this results in an aliasing attenuation of a paltry TODO: (objectivity) 8.06 dB. Combined with the passband droop of 1.48 dB (a signal attenuation of almost 16 %) at that frequency, this makes for a margin of a mere 6.58 dB between the signal's attenuation and the aliasing attenuation!

As a benchmark for or own solution, we TODO: (impersonal) will use measurements conducted by our predecessors with the STEMlab's stock configuration, listed in Table 2.1. The measurements do not actually suggest a very bad performance of the filter. This is because they were conducted at specific harmonic frequencies. The primary issue with a moving averager is less SNR for specific frequencies, but aliasing when measuring a non-harmonic signal which has frequency components above $f_{s,low}/2$. In that case, the considerations from Figure 2.3 become crucial to understanding the aliasing issue. This behavior was also confirmed in [**?**]. Our TODO: (impersonal) primary aim is therefore more to reduce these aliasing effects rather than purely trying to improve SNR for specific single frequencies.

In conclusion, we can formulate three main objectives:

- Reduce aliasing of out-of-range frequency components into the passband.

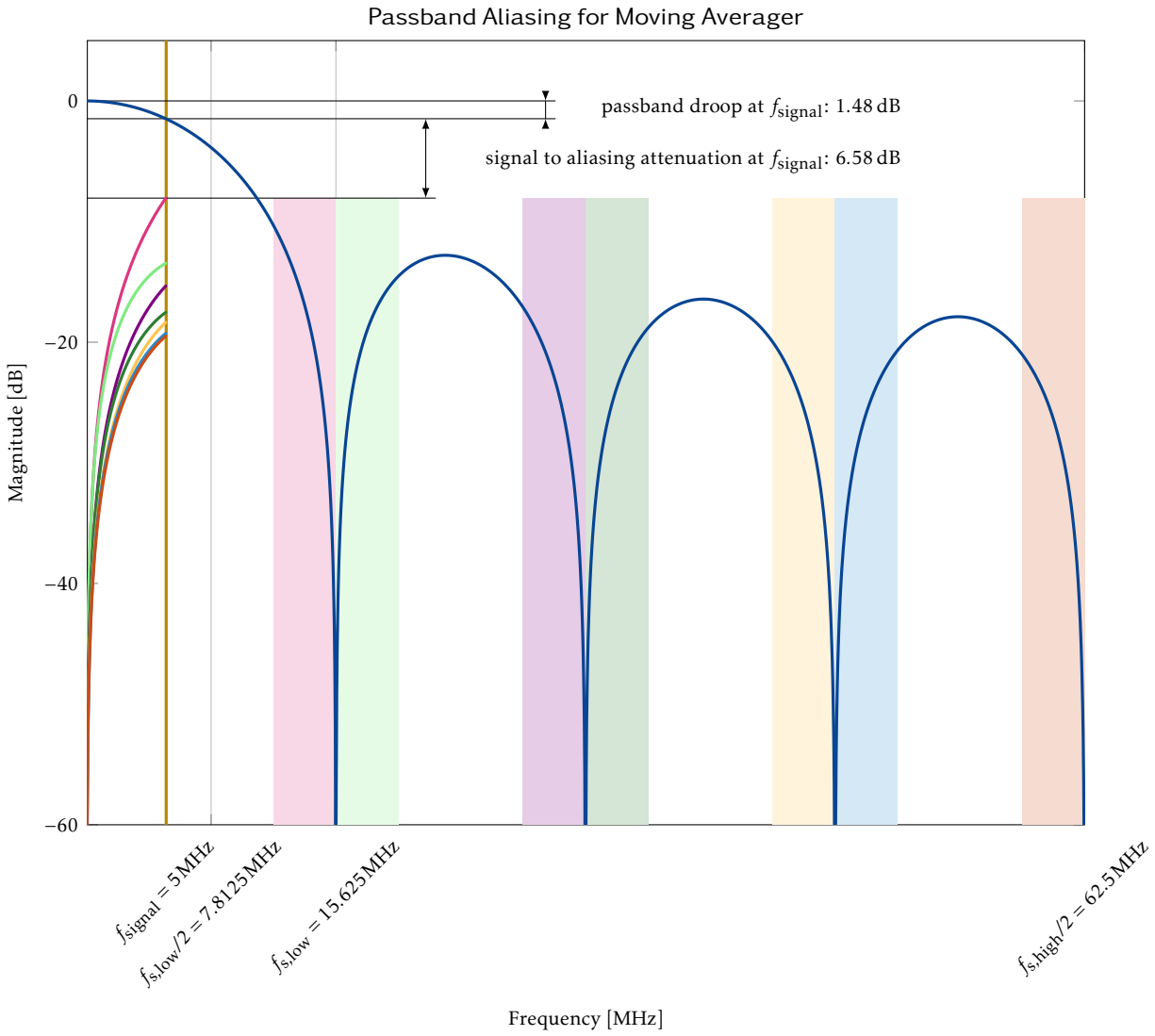


Figure 2.3: Frequency response, passband droop and aliasing attenuation for a moving average filter of order 7 for a decimation by a factor of 8. The incoming sampling frequency is 125 MHz, the outgoing sampling frequency is 15.625 MHz, the measured signal has a frequency of 5 MHz.

Table 2.1: Measurement results for STEMlab 125-14 from [**?**]. SNR was determined for a specific harmonic frequency signal for each sampling rate.

R			$f_{s,low}$ (kHz)	f_{signal} (kHz)	SNR
2^0	=	1	125 000	10 000	62.8 dB
2^3	=	8	15 625	5000	69.8 dB
2^6	=	64	1953	500	76.0 dB
2^{10}	=	1024	122.1	40	76.4 dB
2^{13}	=	8192	15.26	5	82.4 dB
2^{16}	=	65536	1.907	0.5	83.5 dB

- Improve SNR TODO: (all thre points sloppy formulated?).
- Have a nicer set of reduced sampling rates.

2.2 Possible Solutions

Based on the previous section, it is clear that the downsampling filter from the STEMlab's stock configuration must be replaced if better aliasing attenuation is to be attained. Furthermore, since a new filtering system is being implemented anyway, one may wish to change the rate change factors in order to achieve a set of nicer-looking reduced sampling rates. The question now becomes: *How*?

Problematic is that as of spring 2017, the official code by Red Pitaya base has been split into two: There is a new development branch bringing major changes, while the old code (which has been used in the projects preceding this one) is no longer being maintained, as explained by one of the developers (see [**?**]): TODO: (citation wrong? gertiser will incapacitate us?)

> Current development is done on the `mercury` branch on the FPGA subproject of the same name. The code is still under heavy development and not stable. All other FPGA subprojects will not be developed further.

Essentially, this presents us with three options on how to proceed:

- Use the old Red Pitaya codebase, and implement our new filtering system based on that.
- Use the new Red Pitaya codebase. Due to this codebase being unstable according to the developers themselves as of the time of this project, we consider this to be an unviable option.
- Use an alternative ecosystem, either by a third party (if one can be found) or developed by ourselves.

In order to have a somewhat objective measure to compare the first and last option, a decision matrix is used; see table 2.2. In the following paragraphs, the criteria and weighings in the table are elaborated upon. The table uses a scale of 1 through 6, with 1 being the worst and 6 being the best score. At the end, totals are tallied and compared.

Reliance upon others: Using the existing Red Pitaya ecosystem would also couple us to any problems inherent to that platform and to the solutions developed by preceding projects based on it. Any bugs encountered in the Red Pitaya ecosystem would either require a bugfix by the manufacturer or a workaround on our part. Since the old Red Pitaya codebase is no longer being maintained by the developers, bugfixes will not be available, leaving us to clean up any potential issues inherent to the codebase. If we were to implement our own solution instead, we would be less reliant upon others to fix any inherent problems to the code. These factors lead us to give the existing codebase a rather weak rating of, compared to a strong rating for the choice of pursuing our own solution.

Flexibility: Obviously, implementing our own system would give us much higher flexibility than using the existing ecosystem. The only two true limitations would be the time and hardware resources available to us. the official codebase would limit us to its capabilities. Adding new functionality to the existing codebase would be possible, but adapting

software for purposes for which it was not originally intended does tend to be a time-consuming procedure in our experience. Therefore, the option of developing our own solution wins out again here.

Complexity of complete system: This refers not just to the complexity of the components we work on, but of the entire platform and ecosystem. As anyone who bothers to peruse the Red Pitaya codebase ([**?**]) can dedude, it is a large ecosystem with many features and capabilities, most of which are not relevant to our needs. A custom system developed for the specific needs of this project would be much leaner and have fewer points of failure.

Labor costs: Here is where using the existing codebase would be beneficial in our view. While understanding its inner workings would doubtlessly be required and take a significant amount of time (particularly in view of the sparse documentation at this point in time), developing a completely new system more or less from scratch would be a much costlier undertaking in terms of required man-hours, we estimate.

Chances of success: Basing our work on the existing codebase would unburden us of the legwork needed to get basic functionality up and running. We could exchange the filter components of the existing system with our own, but leave most of the remaining system untouched. The challenge would be to understand the system well enough to do this. Building our own system would require re-implementing more components of the Red Pitaya platform than just the filters. As each additional task increases the risk of failure, this is not without risk to the overall success. Overall, we consider this factor to be roughly even between the two choices.

Robustness of third-party components: This criterion takes into account our assessment of the reliability and dependability of any component we use which is not developed by ourselves, along with its documentation and manufacturer support. In the case of the Red Pitaya codebase, this would primarily comprise the codebase itself as well as any components for it developed by other parties. If we were to develop our own solution, the building blocks would primarily be the Xilinx toolchain and libraries for the SoC.

Because the Red Pitaya codebase is rather large, not well documented, comparatively new and no longer being maintained, and the company is small and more likely to be stretched thin, we do not score the Red Pitaya codebase highly here. The Xilinx toolchain and FPGA libraries, while undoubtedly not without bugs, have had a lot of time to mature on the other hand. Also, Xilinx is a big company with vast resources, so any bugs which are encountered in their products are more likely to be addressed in our view. For these reasons, we score developing our own solution higher than using the existing cosebase.

Reparability: If any issues are found in the resulting product, we are more likely to be able to fix them if it is our own codebase rather than that by a third party.

Long-term viability: Developing our own solution would allow us (or anyone else wishing to base their work off of ours) to address future needs relatively easily. Using the existing code base would make this more difficult in our view.

Conclusion: Developing our own solution does carry a significant risk and is likely to require more work than basing our application on existing work. But in light of the mentioned drawbacks of the latter approach, we still find that it is the preferable approach and is more likely to lead to a successful outcome.

Table 2.2: Decision matrix comparing the usage of the existing Red Pitaya ecosystem against building our own data acquisition system. Weighing: Scale of 1 (worst) to 6 (best). More total points is better.

	RP	Custom
reliance on others	2	5
flexibility	2	5
complexity of complete system	1	4
labor costs	4	2
chances of success	4	4
available documentation for used building blocks	3	5
robustness of third-party components	2	5
reparability	2	5
long-term viability	2	4
Total	21	37

2.3 Concept

Having concluded that we shall develop our own solution from scratch, we can now develop a concept for how that solution will look. On the most fundamental level, it will require the following components:

- a custom FPGA firmare for data acquisition and filtering
- a new scope application for data visualization
- an interfacing layer between the FPGA and the scope
- optionally, the possibility to connect to other applications like Matlab

The following sections elaborates on the general shape of our solution. Chapters 4 through 6 explain the components in detail, while Chapter 3 documents the filter design process, which is rather separate from the firmware and software and therefore a dedicated chapter.

2.3.1 FPGA Components

On the most abstract level, the FPGA part of our system will need to be able to

- acquire data from the ADC,
- decimate and filter it,
- and pass it on for further processing.

Due to the platform's open nature, there exist some projects for the STEMlab by parties other than Red Pitaya themselves. One of these is *Red Pitaya Notes* by Pavel Demin, available at [**?**]. As part of that project, Mr. Demin has implemented an ADC core for the FPGA which acquires data from the ADC's two channels and passes it on via an AXI4 streaming interface[1]. Since this project is open-source and has a permissive license (MIT license [**?**], template text can be found in Appendix C.1 on page 92) and fulfills our main technical requirement (easily interfaceable), it is used in our project to interface the FPGA with the ADC.

[1] Xilinx's proprietary general-purpose interconnect for moving large amounts of data around an FPGA

For filtering, one can either implement custom filter topologies from basic FPGA building blocks (adders, multipliers, etc.), write completely custom VHDL or Verilog code, or use ready-made blocks, if available. Xilinx provides such FPGA blocks for FIR and CIC filters, both of which come with excellent documentation (see [**?**] and [**?**], respectively). In order to avoid re-inventing the wheel, so to speak, and take advantage of some of the advanced features offered by these two blocks, we use these two building blocks in our design.

Interfacing between the FPGA and the outside world requires a component which can take data from the filters and hand it off to the GNU/Linux running on the ARM cores. Additionally, the user must be able to trigger measurements from the operating system. Luckily, Mr. Hüsser TODO: (impersonal) already developed such a data logging core for a Xilinx FPGA in an earlier project [**?**], which comes with a kernel module to interface with GNU/Linux. With some minor adaption work, this core should suit our needs nicely.

The last component needed for the FPGA is a control logic to set the decimation rate and enable or disable specific components of the data processing chain. This should be fairly straightforward, and will be implemented by ourselves as a custom block.

In summary, the concept for the FPGA data processing system looks as follows:

- The ADC is accessed via Pavel Demin's ADC core.
- Filtering the data is conducted with Xilinx's CIC and FIR compilers.
- For interfacing with GNU/Linux, Mr. Hüsser's data logging core is used.
- A custom control logic configures the data processing chain on-the-fly through user input.

2.3.2 Interfacing Layer

The interfacing layer is responsible for sending the data from the STEMlab to the user application running on a computer. This is done via a Gigabit Ethernet connection. As is easily apparent, this connection is far too slow for moving all data off the device which is generated by the ADC (about $3.7\,\mathrm{GiB\,s^{-1}}$), hence the need for decimation (among other reasons).

For the interfacing layer, a server application is run on the STEMlab, to which a client can connect. The server takes data from the logger's kernel module, processes and packages it as necessary, and sends it out over Ethernet. The application is documented in detail in Chapter 5, starting on page 42.

TODO: More blabla?

2.3.3 Scope

The scope is the main interface between the end user and the STEMlab in our system. Through it, data can be visualized and analyzed in both the time and frequency domain.

In order to ensure maximum portability, the scope is implemented as a web application rather than a cusom binary. This allows it to function on any operating system with a reasonably modern browser. A through documentation of its capabilities and implementation is available in Chapter 6, beginning on page 6.

TODO: more blabla?

2.3.4 Summary

in summary, our concept looks as follows:

CHAPTER 3

Filter Design

Some key points underlying the theory of filters have been treated in Chapter 1, and Chapter 2 defines the overall objectives of this project; implementing a custom filtering system on the FPGA among them. This chapter will now develop a concrete concept for that filtering system, address some of the issues encountered when moving from the theory of filters to the practice of designing them, and then define the actual filters which are to be used on the FPGA. The implementation of those filters on the FPGA is addressed in the following chapter, beginning on page 41.

3.1 Requirements

The overarching objective is downsampling the signal coming out of the ADC. This section will derive upper and lower boundaries for the downsampled frequency range, and then define the specific downsampling ratios to be used.

The total data rate from the ADC is

$$\begin{aligned} S &= 125\,\mathrm{Msample}\cdot\mathrm{s}^{-1} \\ N_{\mathrm{ch}} &= 2 \\ B_{\mathrm{ch}} &= 14\,\mathrm{bit}\cdot\mathrm{sample}^{-1} \\ B_{\mathrm{ch,pad}} &= 2\,\mathrm{bit}\cdot\mathrm{sample}^{-1} \\ B_{\mathrm{ADC}} &= N_{\mathrm{ch}}\cdot\left(B_{\mathrm{ch}}+B_{\mathrm{ch,pad}}\right) = 32\,\mathrm{bit}\cdot\mathrm{sample}^{-1} \\ R &= S\cdot B_{\mathrm{ADC}} \qquad\qquad = 4\,\mathrm{Gbit}\cdot\mathrm{s}^{-1} \end{aligned} \tag{3.1}$$

Where:

S	: sampling rate
N_{ch}	: number of channels
B_{ch}	: channel width
$B_{\mathrm{ch,pad}}$	: padding per channel
B_{ADC}	: total width of bit stream out of ADC
R	: total data rate out of ADC in bit

Table 3.1: The chosen downsampling ratios, their prime factor decompositions, the downsampling ratios distribet across stages, and the resultant sampling rates

R	Decomposition	Stages	f_s (kHz)
5	$5 = 5^1$	5	25 000
25	$5 \cdot 5 = 5^2$	$5 \rightarrow 5$	5000
125	$5 \cdot 5 \cdot 5 = 5^3$	$25 \rightarrow 5$	1000
625	$5 \cdot 5 \cdot 5 \cdot 5 \cdot 5 = 5^4$	$25 \rightarrow 5 \rightarrow 5$	200
1250	$2 \cdot 5 \cdot 5 \cdot 5 \cdot 5 \cdot 5 = 2^1 \cdot 5^4$	$125 \rightarrow 5 \rightarrow 2$	100
2500	$2 \cdot 2 \cdot 5 \cdot 5 \cdot 5 \cdot 5 \cdot 5 = 2^1 \cdot 5^4$	$125 \rightarrow 5 \rightarrow 2 \rightarrow 2$	50

The upper boundary for the resulting sampling rate is set by the STEMlab's network connection, which has a data rate of $1000\,\mathrm{Mbit \cdot s^{-1}}$. A downsampling factor of at least 4 is therefore required for real-time data transmission. Because 125 is not divisible by 4, a factor of 5 is chosen instead. This makes for a resulting data rate of $800\,\mathrm{Mbit \cdot s^{-1}}$, which should also be easily sufficient for protocol overhead.

On the lower end of the spectrum, the system should still be able to process audio signals. Common sampling frequencies for audio are 44.1 kHz for audio CDs, and 48 kHz for the audio component of audio-visual applications. Neither of these frequencies fit nicely into 125 MHz (requiring large prime factor for the rate change), so the lower boundary is specified as 50 kHz, corresponding to a downsampling ratio of 2500.

To cover additional use cases, some additional sampling frequencies between these two boundaries are specified. Table 3.1 contains the complete list of downsampling ratios, along with the corresponding sampling frequencies.

3.2 Cascade Concept

Based on the downsampling factors from Table **??**, this section presents the general concept for the cascades which implement those rate changes.

As discussed in Section 1.3, implementing high downsampling ratios in a single stage is generally not a sound design choice. Consequently, the downsampling ratios from Table **??** must be decomposed into smaller factors, which can then be distributed along a chain. These factors must fulfill the following criteria:

- Filters for different stages should be re-usable across multiple downsampling ratios in order to save resources. Rate change factors which are common to multiple rate change factors are therefore preferred.
- The factors for the individual stages should be large enough to be of utility, but small enough so as not to make the resulting filter impractically narrow.

CIC filters are well-suited for large rate changes, but are not an optimal solution for smaller ones. As an example of a low-rate change CIC filter, Figure A.2 on page 77 in Appendix A.1 shows the frequency response of a CIC filter with a rate change of 2.

Based on that observation, it is reasonable to implement the lower rate change factors without CIC filters, while using CIC filters as the first element in the filter chain for the higher rate changes. This allows taking advantage of the CIC filter's high computational efficiency for large downsampling rates, while still having good frequency response behavior for the lower rate changes. Table 3.1 shows the prime factor decomposition for the overall rate changes, as well as the factors chosen for the individual filter chain stages.

Other choices are of course possible. Particularly in the case of $R = 625$, one may choose to implement a chain of $125 \rightarrow 5$ instead of $25 \rightarrow 5 \rightarrow 5$. The two implementations as they would be in the final design are compared in Figure B.1 on 83 in Appendix B.1. While the $125 \rightarrow 5$ chain would offer better stopband attunation behavior over certain frequency ranges and therefore improved SNR, the $25 \rightarrow 5 \rightarrow 5$ chain offers the advantage that if the design is ever changed and only the higher sampling rates are implemented (removing the chains for $R = 1250$ and $R = 2500$), it can re-use the elements from the higher chains. The $25 \rightarrow 5 \rightarrow 5$ chain is therefore chosen.

3.3 Filter Specifications

Continuing from Table 3.1, this section presents the concept for the cascades which are used in our design. Specifically, requirements and constraints for the filters in those cascades are specified. For this purpose, it is no longer sufficient to merely consider the filters in the mathematical sense; resource usage on the hardware must be taken into account.

The hardware places two main contrainst on the design:

- Number of available LUTs: 17 600
- Number of available DSP slices: 80

The number of LUTs is relevant for storage (filter coefficients), the CIC filters[1], the rest of the processing system, and control logic. The DSP slices can therefore be reserved for the FIR filters. Because the device has two channels, only 40 slices may be used per channel.

In order to have a realistic gauge for resource usage of the FIR filters, it is necessary to keep in mind the two factors which primarily influence resource usage:

- the sampling frequency at which the filter runs (its incoming sampling rate),
- and the number of coefficients, and therefore, adders and multipliers.

It is therefore important to know which filters in the design run at which sampling rates. For this to be possible, a concept of the six different filter chains is needed. Based on Table 3.1, Figure 3.1 depicts that concept. It includes all the required filters of the design, including the compensators for the CIC filters. Note that one of the compensators is itself used as a decimation filter in the case of the $R = 1250$ and $R = 2500$ chains.

Because the final filter in a cascade is the one which determines the overall transition band (see Section 1.3), it is desirable to have maximally steep output filters in a cascade. Therefore, the filter `5steep` should be as sharp as possible. Since that filter is not just used as the final stage in some cascades, but also as the single filter for the $R = 5$ chain, it runs at the highest sampling frequency[2]. `5steep` is therefore the most critical filter in terms of resource usage. The filter `5flat` also runs at the highest sampling frequency in the $R = 25$ chain, but because it is not the final filter in that chain, it need not be as steep.

While it is possible to estimate the needed resources of a given filter design, reliable figures are best obtained by way of experiment. The FPGA toolchain might make optimizations which are hard to take into account when performing estimates by hand. The results of these measurements are available in Appendix B.2 on page 82. Based on those numbers, a filter size of around 250 is determined to be a reasonable boundary for `5steep`; a filter of that size uses about 25 DSP slices, leaving 15 slices for other filters per channel.

[1] The CIC compiler block by Xilinx can be configured to utilize LUTs instead of DSP slices for its computations. The FIR compiler can only use DSP slices for its computations.

[2] It should be noted at this point that a filter which is configured to run at a high sampling rate can be re-used as a lower stage in a cascade in the Xilinx toolchain. The filter's behavior in that case is correct, even when being run at a lower rate than maximally possible.

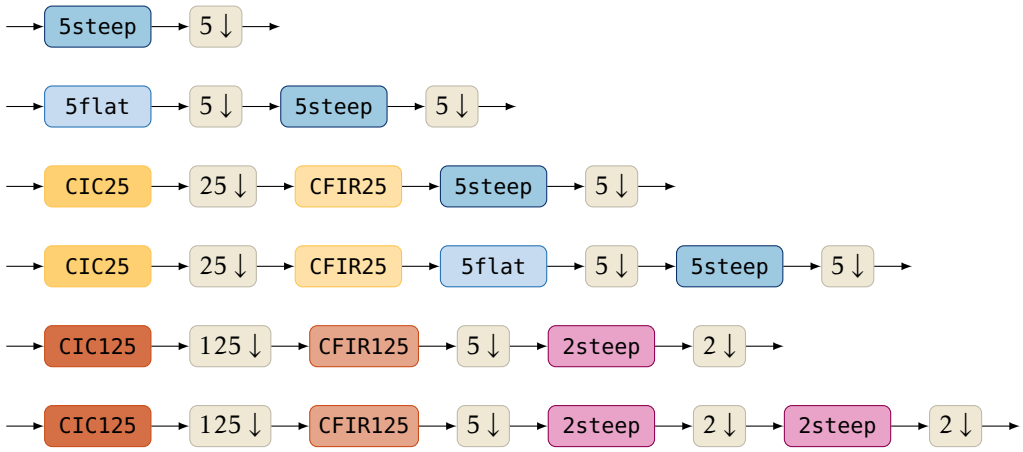


Figure 3.1: The concept for the filter chains

number of slices for the remaining filters (per channel). Of these, `5flat` is the most critical, because it must also be able to run at the full incoming sampling frequency. `2steep` runs at a much lower sampling rate and can therefore be of significant size without a notable penalty in resource usage. The final implementation of `5steep` is actually smaller, at a length of 204 coefficients and a DSP slice count of 22 per channel.

Based on these findings and the measurements of the STEMlab's stock configuration from Section 2.1.2, it is possible to define performance specifications for the filters without needing to prod in the dark, so to speak. The following paragraphs explain the considerations which lead to the final filter specifications. The results are summarized in Table 3.2.

Requirements for `5steep`: Based on the results of the moving averager used in the STEMlab's stock configuration, even a FIR filter with moderate performance characteristics should already offer significant improvements. In order to achieve notable gains over the default configuration, the following performance goals for `5steep` are specified, according to the pattern explained in Section 1.2.2:

- Passband ripple: better than 0.25 dB
- Stopband attenuation: 60 dB or better
- Transition band width: $0.05 \cdot f_s/2$ or better

The stopband attenuation criterion is also applied to all other filters. Anything else would be a waste of resources, as shown in Figure 1.22 in Section 1.3.

Requirements for `5flat`: This filter need not have a drop-off as sharp as `5steep`, as long as the end of its transition band does not overlap with the first spectral copy of `5steep` (see Figure 1.21 in Section 1.3). However, because it is in a cascade with `5steep`, a sharper requirement on its passband ripple is imposed, in order not to worsen overall passband ripple behavior of the cascade too much.

Requirements for `2steep`: In theory, the first decimator-by-two in the chain for $R = 2500$ could be designed with a wider transition band than the one which is being used as the last stage (hence the designator `5steep`). However, because these filters do not have to run

Table 3.2: The target filter specifications. These parameters are based both on the desired frequency domain behavior of the filters as well as the feasibility of implementation in terms of resource usage. For resource considerations, the results from Appendix B.2 are used as a guideline.

Filter	Passband Edge ($\times\pi$rad $\cdot$ sample^{-1})	Stopband Edge ($\times\pi$rad $\cdot$ sample^{-1})	Passband Ripple (dB)	Stopband Attenuation (dB)
`5steep`	0.2	0.225	0.2	60
`5flat`	0.2	0.3	0.05	60
`CIC25`	0.008	N/A	N/A	60
`CFIR25`	0.008	0.016	0.05	60
`CIC125`	0.0016	N/A	N/A	60
`CFIR125`	0.0016	0.0024	0.05	60
	Transition Band Width ($\times\pi$rad $\cdot$ sample^{-1})			Stopband Attenuation (dB)
`2steep`	0.004	N/A	N/A	60

at high sampling rates, the amount of DSP slices they use is very low (1 DSP slice per channel), so the same filter can be re-used without a resource penalty.

TODO: halfband

Requirements for `CIC25`: The relevant design criteria for the CIC filter are its stopband attenuation, its decimation rate, and the cutoff frequency/desired passband width (see Figure 1.17 in Section 1.2.3.2). The cutoff frequency is chosen such that it matches the frequency band which is of interest at the end of the filter chain for $R = 125$, i.e. $f_P = f_{s,high}/R = 0.008$. This means that the passband of `CIC25` and `CFIR25` combined is too wide by a factor of 5 for the $R = 625$ chain, but this is of no concern because it will be cut off by `5flat` as the last stage in that case. This allows the re-use of the two filters across both chains without changing their design parameters.

Requirements for `CIC125`: The same considerations as for the other CIC filter apply. The filter and its compensator are specified for a rate change of 125 and 5 instead of 25 and 1, respectively, and the cutoff frequency is set to match the filter chain of $R = 1250$. This makes it twice as wide as it needs to be for $R = 2500$, which is corrected by a second halfband filter.

Requirements for compensators: The compensators are specified according to the considerations laid out in Section 1.2.3.6, with the added feature of `CIC125` also being used as a decimator.

Summary: With the above considerations and the experimental results for resource usage from Appendix B.2, it is possible to formulate a complete set of specifications for the filters. They are compiled in Table 3.2. Translating the specifications from Table 3.2 into absolute frequencies results in the values from Table . The frequency responses of all filters and filter chains are listed in Appendix B.4, starting on page 84.

Table 3.3: The expected relative and absolute transition band widths of the various filter chains, based on the specifications from Table 3.2.

Chain	Relative TB Width of Final Filter ($\times\pi$rad $\cdot$ sample^{-1})	Absolute TB Width of Chain (kHz)
5	0.025	1562.5
25	0.025	312.5
125	0.025	62.5
625	0.025	12.5
1250	0.040	4.0
2500	0.040	2.0

CHAPTER 4

FPGA

CHAPTER 5

Server

Once the FPGA has recorded data that data has to be transmitted over the network. Since implementing networking in hardware is not feasible in most cases this is done via the ARM Cortex A9 core that already has a PHY. To controll all the hardware of the SoC a Embedded Ubuntu Linux is running on the ARM core which can control all hardware units as known from normal Linuxes. An application is then needed that reads the necessary data from the RAM and sends it to the network. In this section the design choices and the internal structure of the server application are explained.

5.1 Requirements

The functional requirements of the server application are:

- Read the system status and transmit it over the network.
- Receive commands over the network, translate them where needed and relay them to the FPGA IP.
- Read data from the RAM and transmit it over the network.

More requirements came in after choosing the client side model but are not part of the base requirements and will thus only be explained in the Section 5.2.

5.2 Design Choices

As the ZYNQ Logger comes with a kernel module that has to be interfaces via IOCTL calls it is recommended to write the application in C or C++. This is due to the nature of the Linux which still requires mostly C for interfacing. There is some IOCTL interfaces in Python and Rust for example but as those bear even more problems on ARM Linux as not all libraries and features are present.

As the server application is rather on a high level in the complexity sense C++ would be a good choice as it eliminates quite some caveats C has. Furthermore all the C features can be used equally in C++.

For this reason C++ was chosen as the environment to implement the server application.

Because JavaScript was chosen for the client side, WebSockets became mandatory. This is not really a design choice but rather a inherited requirement. Before the final descision on JavaScript was made, existing WebSockets libraries were tested. With a uWebSockets [**?**] a great, performant, but badly documented library was chosen. More on this in Section 5.3.2.

As a data format for settings and stats, JSON is used. This is for the simple reason that JSON has by far the biggest usage of all the available formats, the specification is simple and JavaScript can parse it natively into a JavaScript object. Here again C++ is a better choice than C as it has a lot of high level libraries that can serialize and more importantly deserialize JSON objects.

5.3 Implementation

5.3.1 Application Structure

The application is based on an asynchronous event loop. A struct holds the entire application state such that any event can access the entire data such as the socket handle, the current sample rate or the requested number of bytes.

The event loop can be observed in Figure 5.1.

5.3.2 uWebSockets

To expose a WebSocket uWebSockets [**?**] (uWS in further text) is used. It is a library with a tiny footprint that is very performant [**?**]. Of course the server application will never reach the limits of uWS as the network will give up before but it is important that this is actually the case.

uWS is based on epoll, libuv or boost::asio depending on the users choice. All of those are async (more on this topic can be read in [**?**]) libraries which makes networking very conventient.

uWS comes with callbacks that can be registered for each WebSocket event (see Section 6.2.3.1). Furthermore the user can hook into the event loop and register other events such as a reocurring timeout (timer).

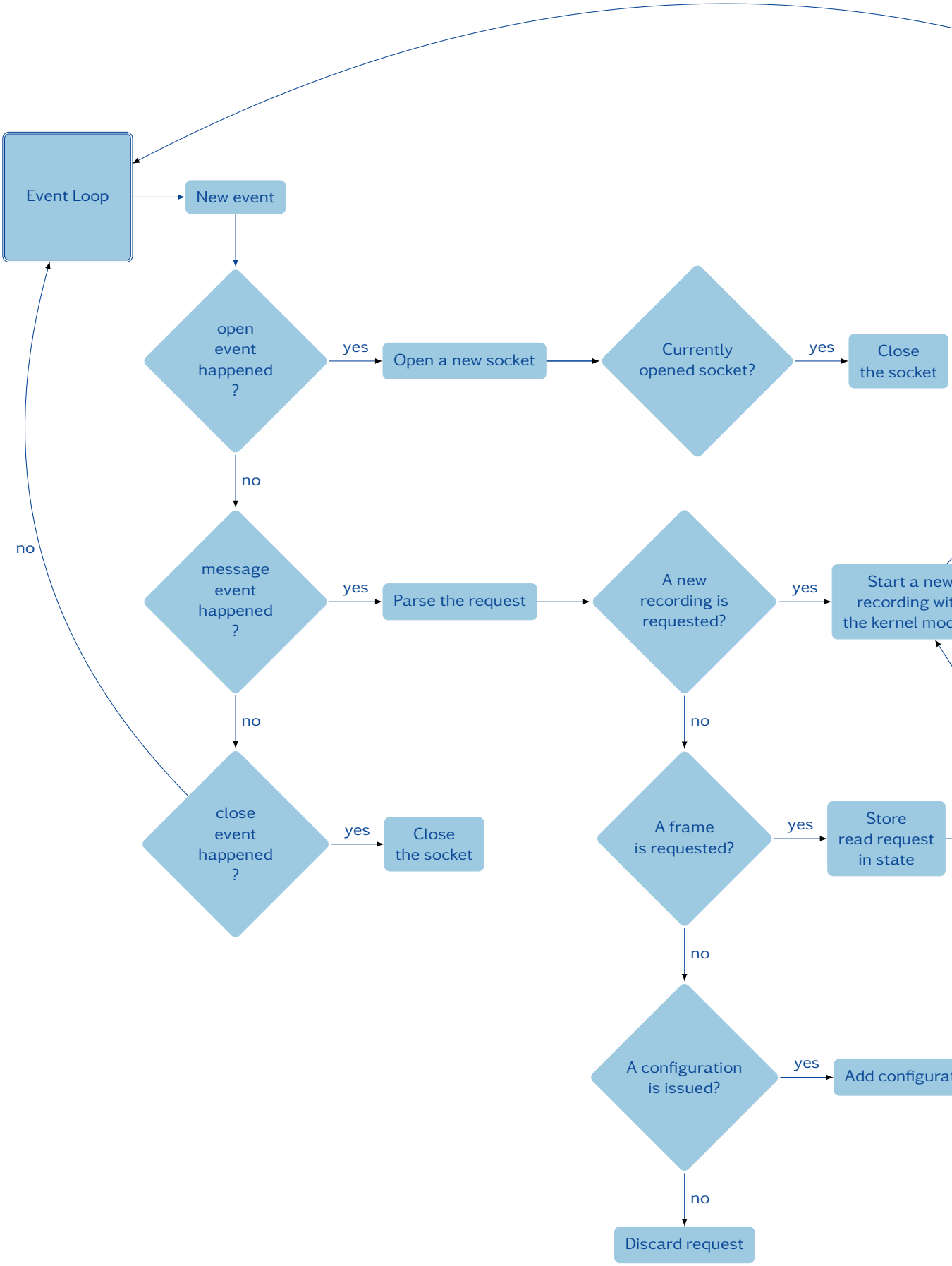


Figure 5.1: The servers event structure.

CHAPTER 6

Graphical Front End

To view measured data a oscilloscope application (scope in further text) with a graphical user interface (GUI in further text) and mathematical capabilities like calculating the spectral density or the SNR of the signal was created. It can receive the recorded samples over the network and display them on a canvas. Furthermore it manages triggers and does a lot of math to get more specific metrics of a signal. In this section the requirements for this piece of software, the design choices and the implementation details as well as the results are discussed.

6.1 Requirements

The requirements for the scope were given by the scoping application "Spektrum Analyzer" TODO: cite?? written in Java by Prof. Gut and his students. The task description required the new scope to have the same features as the old one plus as many more as possible.

The requirements were as follows

- Receive data in configurable size over the network.
- Display received data in time as well as fequency space.
- Calculate the RMS power density in the signal.
- Calculate the THD ratio of the signal.

6.2 Design Choices

There is a wealth of programming languages to choose from but only a few suit a task best. And there are as many libraries helping with graphics and networking as well for most of those languages. In the following sections explain why JavaScript and web technologies are used to implement a GUI and mathematical functions in this project.

A select few popular programming languages have been evaluated. They have been given weights for certain attributes of the respective language to determine the best fitting one. The results are visible in the comparison matrix 6.1.

All the attributes with their impact and meaning for this project are explained in the following.

Open Standard

Since this is a university based project meant for educational purposes too, it is very important to make all source code available under public license. Thus it is important to have a company and paid model independant solution. Many languages are managed by a council or similar and open to public commits and thus deemed an open standard. Some are managed by a company and not classiefied as a open standard.

Networking

To ensure a fast and lossless data transfer, it was very important to have the choice between good networking protocols as well as convenient libraries to ease the use of those standards. Networking is not a trivial thing and standards can be quite engineering- and feature-heavy. Thus it is important to have ready-to-use libraries that abstract the network. For more information on evaluated networking solutions, read Section 6.2.1.

Graphics

An oscilloscope is quite requiring when it comes to graphics, since a image-stream that is fluent for the human eye has to be provided in a high resolution. This fact made it indispensable to use a library that interfaces OpenGL. Since a interface is not easy to design from scratch only using rectangles and circles, a GUI toolkit that makes the design process of the GUI easy is indispensable.

Widespread

It is important for the project to use a widespread solution since that way it is easy to obtain information and ask more savy users about certain pitfalls.

User-Friendly

Some solutions are more user-friendly when it comes to toolkits and usage. Since both team members come form a Linux background, it was strongly preferred to use a language that does not quasi-require a huge IDE or requires a lot of uneasy maintenance.

Easy To Use(r)

Since not all users want to fight with installers and package managers, the deployment options as well as general stability of the environment for the binaries were a strong point in the descision process.

Familiarity With The Language

The best toolkits do not matter if none of the involved programmers have ever used it and will struggle with even the basics for a major part of the project duration. Thus it was unavoidable to have some personal preferences for some languages.

After weighting in all the different aspects JavaScript was chosen as the language to implement the oscilloscope. JavaScript is a scripting language that can be interpreted by the browser. It is known for it's high versatility and widespread use in the web community. A few years ago, JavaScript would not have been a viable choice for graphics and networking at all. But with the recent addition, and more important, great increase in stability and performance of WebGL and WebSockets JavaScript has become a very potent

	Rust	C++	Java	Python	JavaScript
Open Standard	6	6	1	6	6
Networking	6	6	6	6	4
Graphics	2	5	5	5	6
Widespread	3	6	6	5	6
User-Friendly	5	5	5	5	6
Easy To Use(r)	3	4	5	6	6
Familiarity With The Language	3	3	4	6	6
Total	28	35	32	39	40

Table 6.1: Weights of certain aspects of possible programming languages.

solution available to everyone. With JavaScript deploying the application to the enduser is as simple as making it accessible via a website that runs on the STEMlab board. Thus it is very convenient for the user to work with the board, considering the assumption that every user has a webbrowser that is able to run the application. A downside of JavaScript is the huge runtime the browser needs to execute the application and thus resulting in a lot of memory ressources used. Since those are easily available nowadays, this issue was considered non-relevant. Another downside of JavaScript is that it leaves very little room when it comes to networking choices. For streaming data there is only WebSockets that performs well. Since WebSockets is a very capable and convenient solution, this issue did not change much in the descision process.

6.2.1 Networking

To ensure a fluent stream of data, very little overhead for the transmitted data is key.

Normally for streamed data *where packets can be lost*, **UDP** is the best choice since it has no overhead for guaranteeing completeness and in-order for all packets sent. UDP sends packets but does not guarantee that none are lost.

For guaranteed transmission and sequentiality of the data, one is advised to use **TCP**. This comes at the cost of some more overhead. This overhead is most of the time completely negligible as a custom solution that performs better than TCP is very hard to write. What TCP does more that is important for this project is congestion control. It ensures that no more packets are sent if old ones are missing. It prevents the network from collapsing because an UDP sender sends all packets it can and thus using the entire bandwidth even if the receiver cannot even process the data at this point. This means that TCP also helps when the bandwith is small by waiting for the current package and not already sending further packages and thus providing sort of automatic bandwith adaption. There is the possibility of using raw TCP sockets or one of TCP's subprotocols. Raw sockets require the user to implement their own protocol entirely to handle data transmission on an application layer whilst using subprotocols already provide a standard way to do so. Two of those subprotocols are HTTP and WebSockets. HTTP comes with great overhead and is meant for single transactions only. It does not fit the project's needs. WebSockets on the contrary are meant for data streaming and fit the project's needs precisely. Since JavaScript cannot use raw TCP sockets and needs WebSockets to espablish a raw-ish TCP socket, this

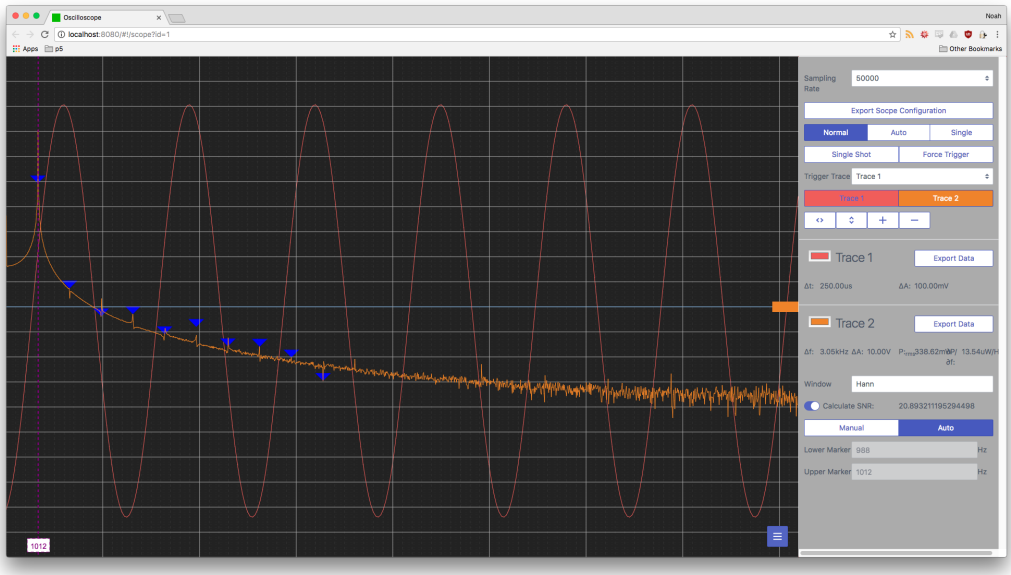


Figure 6.1: The scope application in it's current state, displaying time and FFT data.

is the chosen solution, as it has not downsides relevant for this project. The functionality of WebSockets is explained a little bit more in the subsection 6.2.3.1.

6.2.2 Product

The oscilloscope application is a web application that can be directly loaded from the server running on the STEMlab. It is capable of the following things as of date:

- Receive data over the network for two channels (this is only limitted by the physical channels of the STEMlab).
- Manage triggering including setting a trigger type, level and a number of samples that have to be recorded before and after the trigger was issued.
- Calculate and display the power density spectrum.
- Calculate the SNR both automatically detecting the signal and manually being told where the signal is.
- Calculate the THD for a given base harmonic. TODO: !
- Export data to an array-string.
- Export and load the scope configuration to and from JSON strings.

How the application is structured and certain features are implemented is explained in the following sections.

6.2.3 Application Structure

The entire application consists of a single state tree. This makes it very easy to import and export settings on one hand and on the other it concentrates all the states in one place which gives a way better overview than in the case all the states are scattered in various objects in the application. Listing 6.1 shows an extract of the tree structure.

As every JavaScript application this application runs asynchronous too. The entire eventloop structure can be seen in Figure **??**.

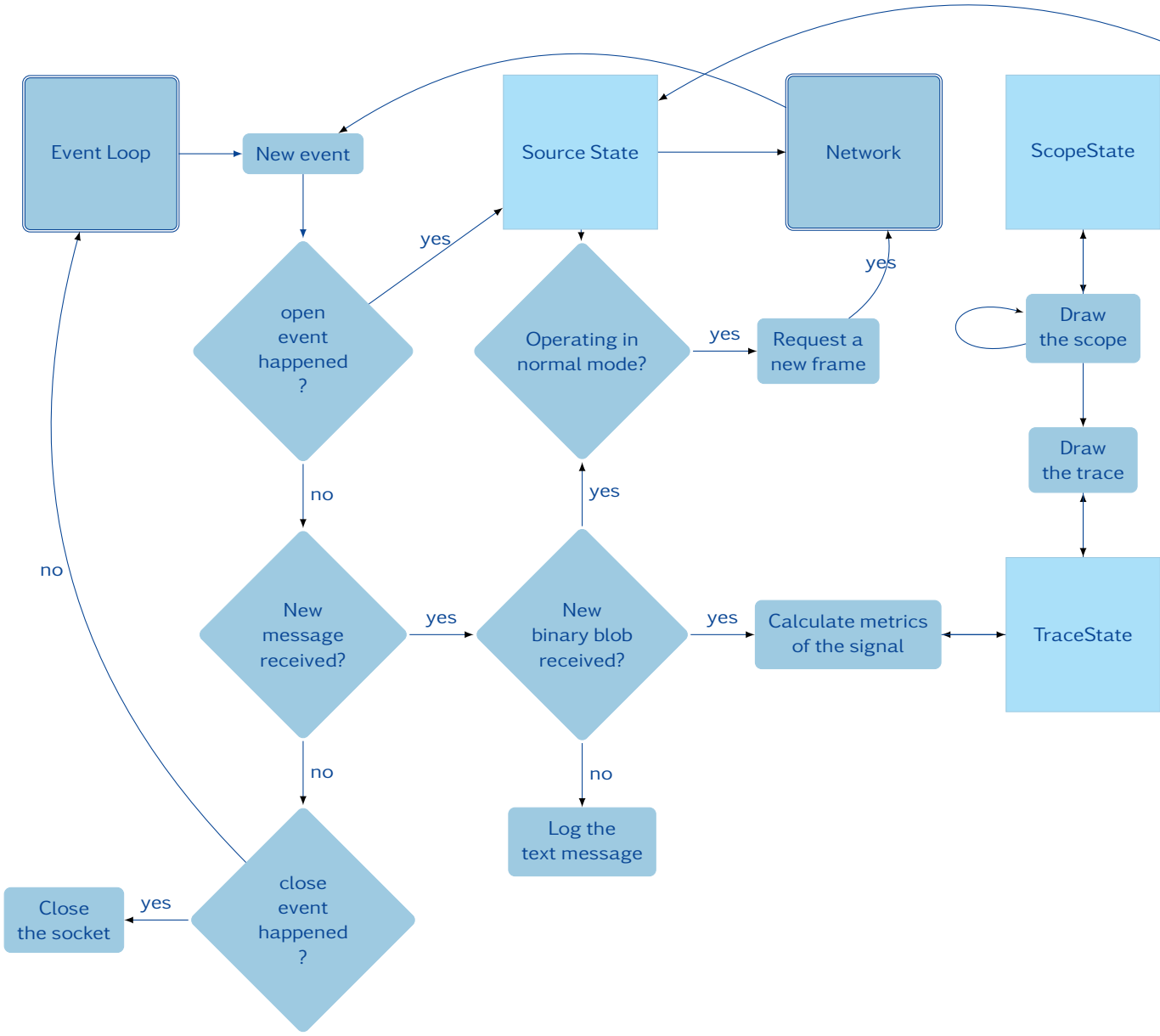


Figure 6.2: The scope event structure.

Listing 6.1: The state tree of the scope application

```
var appState = {
    scopes: [{
        ui: {
            prefPane: {
                open: true,
                width: 400,
            }
        },
        source: {
            id: 2,
            name: 'Source ' + 1,
            location: 'ws://localhost:50090',
            frameSize: 4096,
            samplingRate: 5000000,
            bits: 16,
            vpp: 2.1, // Volts per bit
            trigger: {
                type: 'risingEdge',
                level: 32768,
                channel: 1,
                hysteresis: 30,
                slope: 0
            },
            triggerTrace: 0,
            triggerPosition: 1 / 8,
            numberOfChannels: 2,
            mode: 'normal',
            activeTrace: 0,
            traces: [
                {
                    id: 4,
                    offset: { x: 0, y: 0 },
                    windowFunction: 'hann',
                    halfSpectrum: true,
                    SNRmode: 'auto',
                    info: {}, // Populated during runtime with math
                    name: 'Trace ' + 2,
                    channelID: 1,
                    type: 'FFTrace',
                    color: '#E8830C',
                    scaling: { x: 1, y: 1 },
                    markers: [
                        {
                            id: 'SNRfirst',
                            type: 'vertical',
                            x: 0,
                            dashed: true,
                            color: 'purple',
                            active: true,
                        }
                    ]
                }
            ],
        }
    }]
};
```

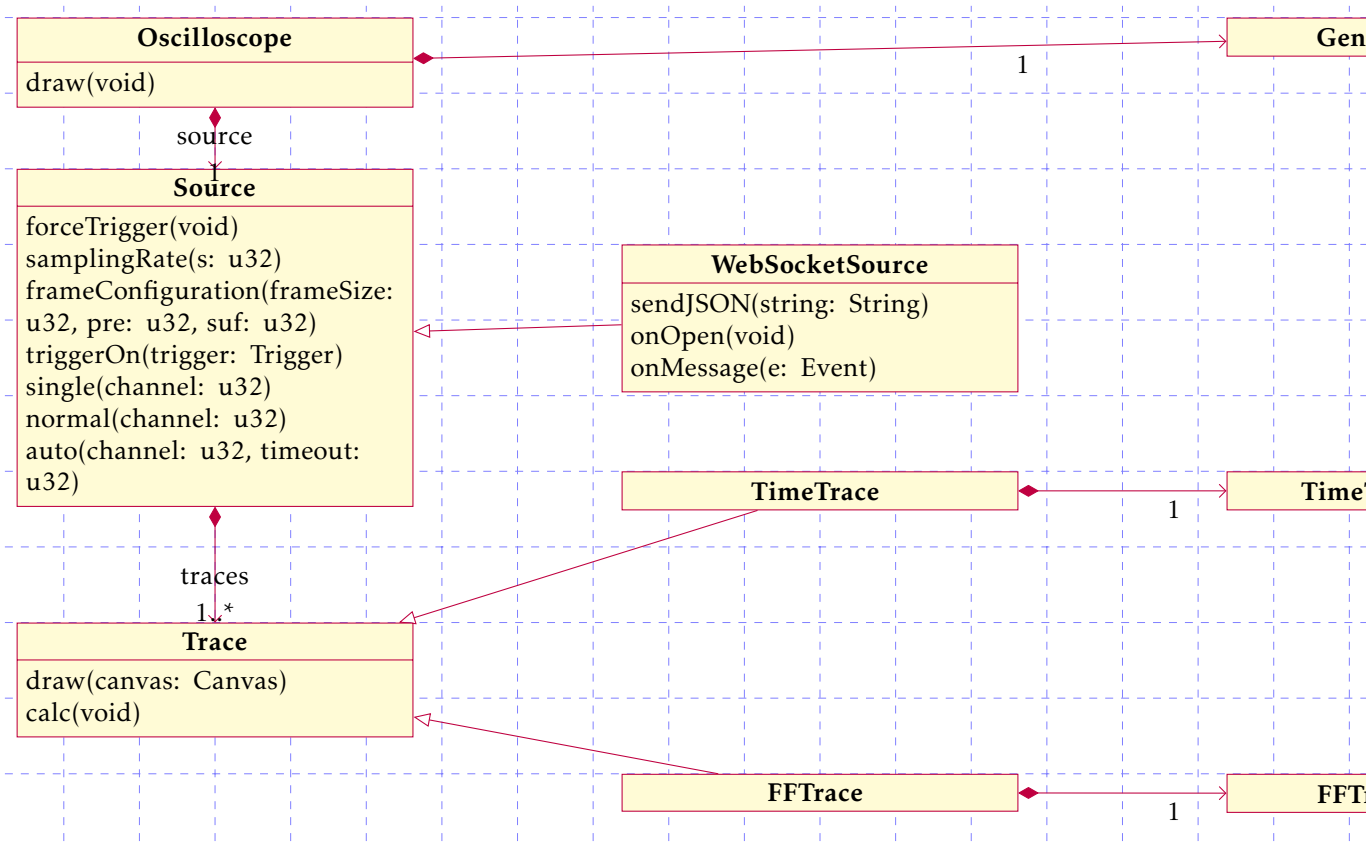


Figure 6.3: The structure of the scope application with all it's important relations.

All of the values that can be controlled through the GUI - and many more - are also controllable directly through the state tree. On application start the entire statetree is loaded and references to extracts of it are handed to the controller objects.

The structure of the application is hierarchical and the most important relations are depicted in Figure 6.3.

In the following some of the more important prototypes and functions are elaborated on.

Oscilloscope

The Oscilloscope prototype is the top level controller which contains exactly one source. It is responsible for handling all mouse event and react accordingly, such as moving the trigger level or zooming and paning. The Oscilloscope draw call is responsible for drawing general info on the canvas that is not part of a specific trace. It is also the caller for the Trace draw call. The oscilloscope manages general information and is responsible for rescaling the canvas and initiating the draw call chain.

Source

The Source controller manages all calls from and to the server. It contains a lot of helper calls to set a trigger or issue a new frame that are called by the Oscilloscope controller or GUI elements. It also contains the important callbacks to send and receive data from the server which are explained in Section 6.2.3.1. The source stores all received frames on itself which then, later in the call chain, will be copied and worked on by the trace controller. Each frame received will always be overridden by the next one, ensuring no memory leaks and current data. Once the Source controller received new data it starts the *calc()* call for each trace to have them update the trace math with the new data. In the next section the concept of WebSockes is explained very briefly.

6.2.3.1 WebSockets

WebSockets' final RFC 6455[**?**] was released in December 2011 and is thus still quite young. It is meant to compensate the lack of raw UDP and TCP sockets in JavaScript which is due to security threats that are not further elaborated here. WebSockets is located in the Application Layer of the OSI model[1]. Instead of directly opening a raw WebSocket, the handshake is done via HTTP(S). This brings the benefit of communicating through the same ports as the browser (80 or 443) which enables the protocol to go through most firewalls. Furthermore it greatly simplifies the handshake for the user (here being the programmer). The client sends an upgrade request to the server which then opens a WebSocket connection. This allows for a very conventient way to use TCP Sockets without any entirely new standards. The section "1.5 Design Philosophy" in RFC 6455[**?**] explains it very well:

"Basically it is intended to be as close to just exposing raw TCP to script as possible given the constraints of the Web.

The only exception is that WebSockets adds framing to make it packet rather than stream based and to differentiate between binary and text data. This differentiation is very useful for this project. Instructions to the server are issued via the text channel whilst data is sent back through the binary channel, allowing for very convenient interfacing with close to no effort."

So in short: WebSockets are close-to-raw TCP sockets whose handle is shared through HTTP(S). JavaScript provides a interface that makes it really easy to shove data back and forth. As nearly anything in JavaScript this is done using callbacks. There is callbacks that handle connections, messages and errors. The code snippet in 6.2 gives some insight how WebSockets in JavaScript are used. All the details can be read in the Mozilla documentation [**?**].

[1] For those not familiar with the OSI model Wikipedia **??** provides a good overview.

Listing 6.2: JavaScript "Using WebSockets"

```
// Open a new socket
this.socket = new WebSocket('ws://localhost');
// Make sure the binary data transmitted
// is interpreted as an ArrayBuffer
// More on ArrayBuffers and Blobs in:
// - https://developer.mozilla.org/en/docs/Web/API/Blob
// - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/ArrayBuffer
this.socket.binaryType = 'arraybuffer';

// Define all the callback handlers
connection.onopen = function () {
    // The connection was established; send some regards.
    connection.send('Hello World!');
};

connection.onerror = function (error) {
    // An error has occurred; print it to the console.
    console.log('WebSocket Error: ' + error);
};

connection.onmessage = function (e) {
    if (typeof e.data == 'string') {
        // If a text type message was received, print it out.
        console.log('Text message received: ' + e.data);
    } else {
        // A binary type message was received.
        // Interpret the values as 16 bit uints.
        var arr = new Uint16Array(e.data);
        // Plot the data.
        plot(arr);
    }
};
```

Trace

The Trace prototype is in charge of displaying the recorded and calculated data in the scope application. The trace controller can be managing an FFT or a normal Time trace. This could be extensible to general math traces e.g. subtracting two differential pair traces for noise cancelling in the future. Each derived trace prototype calculates some metrics and a grid to quantize the signal.

Whilst the Timetrace prototype just returns untouched data, the FFTrace prototype calculates metrics. Namely those are:

- The spectral power density
- The SNR
- The THD TODO: !

The graphics portion of the scope application is the most important part for a nice to use interface. Since the scope application should plot data fast and conveniently as well as display some numbers and provide controls to manipulate the view, it is important to have a good library, that enables the coder to do all those things. It is absolutely key to render the graphics on the GPU. Since the application should be cross platform and open source, libraries using OpenGL is a good choice. Since JavaScript exposes WebGL this is a perfect match.

With the choice of JavaScript & HTML there comes a great wealth of libraries that enable the user to easily write GUI applications. Prototyping is fast and with CSS and a lot of different frameworks the GUI is also nice-looking.

A few frameworks were evaluated to build the controls of the scope, with mithril.js finally being chosen for it's simplicity and flexibility. Mithril is a framework with an exceptionally low footprint and high DOM recalculation. Those two facts are key to a good WebUI, since the User does not want to load lots of data and also does not want to experience any lag when refreshing the UI. More on mithril.js in section 6.2.3.4.

To plot the data some graphing libs could have been used. Those would namely be plotly.js or chart.js. Whilst they bring in a lot of built in functionality like logarithmic plots or automatic axis labeling, they also have a quite heavy overhead. Practical experience and tests have shown that both of them are not meant and performant enough to plot high amounts of data in real time. Thus it was decided to use WebGL draw calls to draw onto a HTML canvas. A HTML canvas is an environment that is exposed directly from the GPU to the user such that they can use GPU rendering inside the browser. More on WebGL and it's functioning in the next Section 6.2.3.2

6.2.3.2 WebGL

The application uses the canvas DOM element which provides a direct interface to WebGL. The user can render vertices to the canvas and even apply shaders or in the case of this application simple 2D geometry calls suffice since it is essentially only necessary to draw lines. Via the canvas one can retreive a 2D Rendering Context on which simple geometry can be drawn. In JavaScript this can be done using the code in 6.3 which shows how a single red line can be drawn on the canvas.

Listing 6.3: JavaScript "Getting a 2D Rendering Context from a Canvas and Drawing on it"

```
// Get the canvas element from the dom
var canvas = document.getElementById('canvas-id');
// Get the 2d context of the canvas
var context = canvas.getContext('2d');

// Set brush color to red
context.strokeStyle = '#FF0000';

// Start a new path and move the cursor
// from start to end of the line to be drawn
context.beginPath();
context.moveTo(x, y);
context.lineTo(x + 100, y + 100);

// Finally actually draw the line on the canvas and end the path
context.stroke();
```

There is also the possibility to draw rectangles, circles and much more. All of those elements can be styled easily via properties of the context environment. All the functionality is documented on the mozilla network [**?**].

Now something can be drawn on a canvas once. If this should be done to create an actually moving image, those draws to the canvas have to happen over and over again. There is various possibilities to do that in JavaScript but only one is actualy performant and recommended. Instead of just drawing to the canvas over and over again, it would be ideal to only do that before a new frame is pulled from the framebuffer by the display. JavaScript provides a interface to register a callback that is called before a new frame is released. This callback will be called with the same frequency as the display referesh rate, which nowadays usually is 60 Hz. To make sure that callback will always be executed it has to be registered again after a callback was issued. The sample 6.4 shows how this is done.

Listing 6.4: JavaScript "Usage of the requestAnimationFrame callback"

```
// The register function is not named the same way in every browser
// Make sure this is the case
window.requestAnimationFrame = window.requestAnimationFrame
                            || window.webkitRequestAnimationFrame;

// Our callback we call for every frame drawn
export const draw = function() {
    // Draw anything needed

    // End draw

    // Register the callback again
    requestAnimationFrame(function(){
        // Execute our callback
        // We cannot hand this directly to the register function
        // since it is not yet known inside it's own definition
        draw();
    });
};

// Initially call the draw function
draw();
```

This callback will not affect the rest of the DOM. Like that the JavaScript runtime will handle the redraws of the dom performantly whilst the callback will render a fluent graph of the data onto just one of the DOM elements.

6.2.3.3 PrefPanes

For each trace an instance of a PrefPane component corresponding to a trace is created. A PrefPane component is a mithril component that creates a vnode that exposes all the necessary controls for it's corresponding trace in the GUI. It is also responsible for displaying calculated data for a trace such as the SNR for a FFTrace. There is also a general PrefPane that exposes general controls such as switching modes, the trigger trace or the active trace. The PrefPanes are built with mithril.js which is introduced in the next section.

6.2.3.4 mithril.js

The official mithril webpage describes mithril.js the following way: "Mithril is a modern client-side Javascript framework for building Single Page Applications. It's small (< 8kb gzip), fast and provides routing and XHR utilities out of the box.[**?**]" Mithril, like a lot of other frameworks such as React, Angular.js or Vue.js uses a virtual DOM. This means that it does not modify the DOM the browser outlines but rather maintains it's own DOM. When a new render call is issued, the virtual DOM calculates all the deltas that stem from new content and applies them to the real DOM. Like this mithril.js performantly calculates the DOM based on a descriptive model. The developer does not have to manually modify an object's state but rather has to describe it.

A redraw generally happens when an event is triggered by any input element but can also be issued manually. A virtual DOM consists of many vnodes and can be mounted on

any node of the browser's DOM as example 6.5 shows.

Listing 6.5: JavaScript "Basic creation and usage of mithril components"

```
// A mithril component is a simple object that has at minimum a
// view() function that returns a vnode.
var HelloWorld = {
    view: function() {
        // Return the toplevel <div> vnode
        return m('', [
            // Create a <h1> vnode with the attribute class="title"
            m('h1', { class: 'title' }, 'A very interesting title!'),
            // Create a <p> vnode
            m('p', 'Hello World!'),
        ])
    }
}
// Get the root div and mount the HelloWorld component
var root = document.getElementById('root');
m.mount(root, HelloWorld)
```

A component can be mounted on any DOM node and becomes a vnode in the virtual DOM. The developer can create new Components by simply creating an object that holds at least a *view()* function that instantiates new vnodes. The new Component can then be instantiated via the *m()* or *m.mount()* command. As this section should only give a base overview on mithril and is not meant to be a manual, further information on mithrils features and usage can be obtained on it's webpage [**?**].

6.2.4 Power Calculation

The scope application calculates the power in the spectrum.

This is done using the FFT algorithm developped by Cooley & Tukey. The FFT code that was used is provided by Kevin Kwok [**?**]. It is a very compact and fast version written in JavaScript and free of charge. It's functionality has been proven by matching it against Matlabs own FFT as seen in Figure 6.4. There is minor numerical differences but that boils down to implementation choices and precision.

For an input signal x we obtain the spectrum with

$$Y[i] = \frac{1}{N} FFT\{x[i]w[i]\} \tag{6.1}$$

The one-sided power spectrum is not far away and we find it with

$$P_{yy}[0] = \frac{Y[0] \cdot Y[0]^*}{NG} and P_{yy}[i] = 2 \cdot \frac{Y[i] \cdot Y[i]^*}{NG} for 0 < i <= N/2 \tag{6.2}$$

The power between two frequency can be found by integrationg all the frequencies or in this case summing all the bins

$$P_{1,2} = \int_{f1}^{f2} P'_{yy}(f)df \approx \sum_{i_1}^{i_2} P_{yy}[i] \tag{6.3}$$

The noise gain (NG) can be extracted from the Table 6.2.

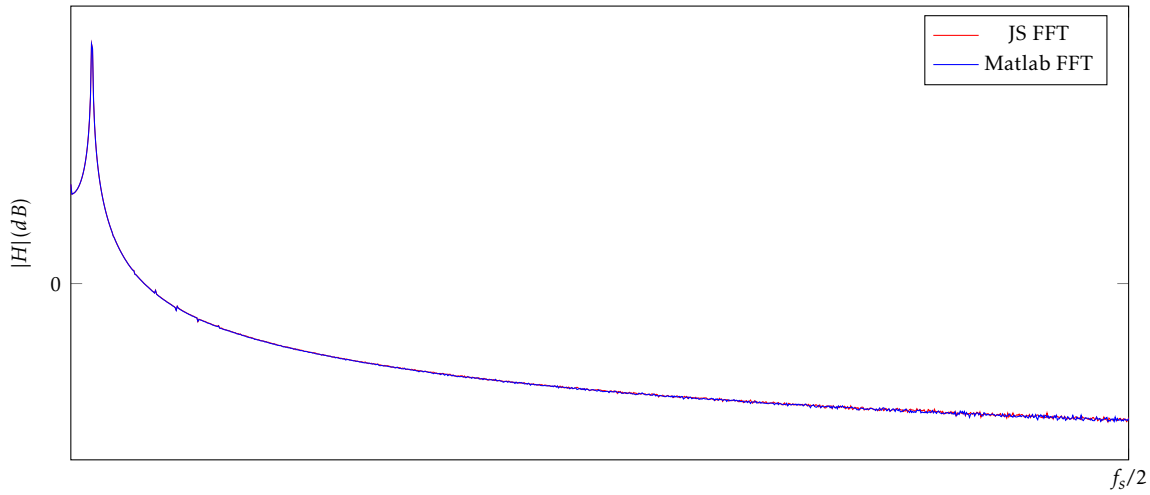


Figure 6.4: The used JS FFT compared to the FFT of Matlab.

Window	CG	NG
Rectangular	1	1
Hamming	0.54	0.3974
Hanning	0.5	0.375
Blackman-Harris	0.3587	0.258
Flat Top	0.2156	0.1752

Table 6.2: Correction factors for the different window types used in the scope application as seen in [**?**].

The exact reasoning behind the math can be read in **??**.

6.2.5 SNR Autodetection

The SNR Autodetection can be done in a static approach based on Section 4.4.3 of [**?**] and especially Table 4.3. The basic idea is to find the peak in the spectrum which normally is the signal and regard it and the $\frac{n}{2}$ spectral lines before and after as the actual signal. The number n is determined using Table 4.3 in [**?**]. The rest of the spectrum is regarded as noise except for the DC offset and it's next $\frac{n}{2}$ spectral lines.

The SNR can be calculated with

$$SNR = 10log_{10}(\frac{P_s}{P_n}) \tag{6.4}$$

During tests only an average SNR of ~33 dB could be measured in the given signal with the Matlab built in function *snr()* consistently finding an SNR of 78dB and more for the same signal. Accounting for the harmonics cancellation the Matlab function has, our SNR would still only be ~43 dB with 10 harmonics cancelled.

A second approach to auto-determining the SNR, is to iteratively increase the number of lines beneath the peak that are still considered signal and do this until the SNR does not

change drastically anymore. In our implementation the change is considered irrelevant when below 0.05 dB which leads us to the following algorithm:

Algorithm 1 An algorithm to iteratively determine the SNR of a spectrum.

$l \leftarrow 1$
$max_i \leftarrow 0$
$SNR_p \leftarrow 0$
$SNR_n \leftarrow 0$
while $|SNR_p - nextSNR| > 0.05$ **do**
  $SNR_p \leftarrow SNR_n$
  $Ps \leftarrow power(spectrum[max_i - l : max_i + l + 1])$
  $Pn \leftarrow power([l : max_i - l] + power(spectrum[max_i + l + 1 :]$
  $SNR_n \leftarrow 10 \cdot log_{10}\left(\frac{P_s}{P_n}\right)$
  $l \leftarrow l + 1$
end while
$SNR \leftarrow SNR_p$

The algorithm in 1 can be repeated for any number of harmonics to cancel them out.

6.2.6 THD Calculation

TODO: how do we calculate THD

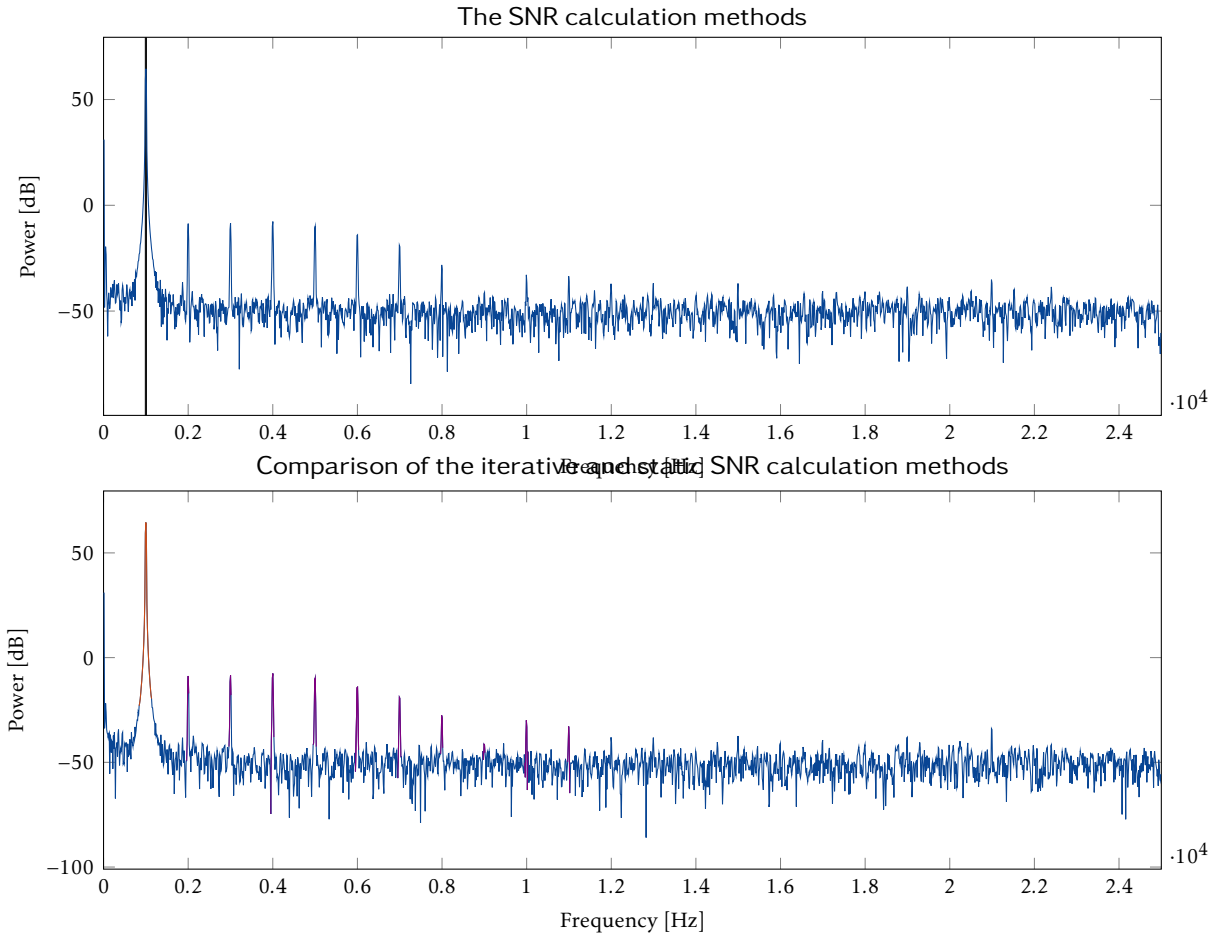


Figure 6.5: The iterative and static SNR detection compared.

CHAPTER 7

Verification

CHAPTER 8

Conclusions

Part II

Developer Guide

Documentation for a person who wishes to utilize our system in their work and/or improve upon it?

Make sure to distinguish between *Implementation* and this part. Lines seem a bit blurry to me (R.F.) at the moment (August 14, 2017).

CHAPTER 9

Project Structure

repo structure: TLDs and what is contained in them.
"Where do I have to go if I want to do X?"

CHAPTER 10

FPGA Toolchain

build box, vivado, linux image for pita, logger, references to vivado documentation

CHAPTER 11

Filter Toolchain

matlab stuff: What is where, how do I specify a new filter, how do I extract the filters into Vivado, where can plot data be found

CHAPTER 12

Server

instruction set, add new instructions, kernel module

CHAPTER 13

Scope

what is where, yarn, . . .

Part III

User Guide

Documentation for the end user. Primarily concerned with the front-end.

CHAPTER 14

Setup

how to get scope to run on computer, how to connect to pita, ...

CHAPTER 15

Operation

gui explanation

Appendices

APPENDIX A

Theoretical Background

A.1 Internal Behavior of a CIC Filter

This section presents an example for a very simple CIC filter to better understand its internal workings. For verification, the filter is also implemented in a Simulink model and simulated.

In the interest of simplicity, we choose a filter with a decimation rate $R = 2$, a differential delay $M = 1$ and $N = 1$ stages. The corresponding topology is shown in Figure A.1; Figure A.2 shows the magnitude frequency response of the filter. As can be clearly seen, this filter would be of very limited use in practice. However, for the purposes of this example, we will feed a DC signal (a constant) into the filter, so the only thing of importance is the filter's DC gain (which is 6 dB, or 2).

Lastly, we will restrict numerical accuracy to three bits in two's complement; the entire range of representable values can be found in Figure A.3. This will limit the number of steps which need to be calculated to gain the desired insight into the filter's mathematical mechanics.

The state of the filter can be calculated by the formulae given in Equations A.1 and A.2:

$$N = 1 \quad M = 1 \quad R = 2$$

$$OUT_{INT}[n] = IN_{COMB}[n] = IN[n] + OUT[n-1] \tag{A.1}$$

$$\begin{aligned} OUT_{COMB}[n] &= IN_{COMB}[n] - IN_{COMB}[n - R \cdot M] \\ &= OUT_{INT}[n] - OUT_{INT}[n-2] \end{aligned} \tag{A.2}$$

The input of the filter shall be a constant of 1, starting at time zero. Once this input is applied to the system, the integrator stage will begin to accumulate the constant. Given an unlimited number of digits (bits), the integrator would in theory reach infinity if it kept

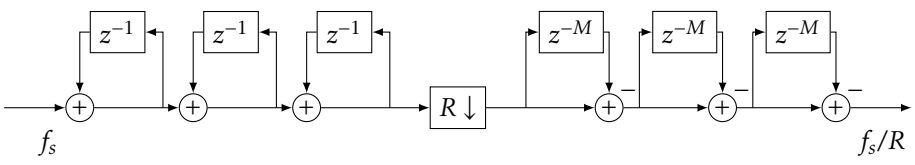


Figure A.1: Topology of the CIC filter for this example

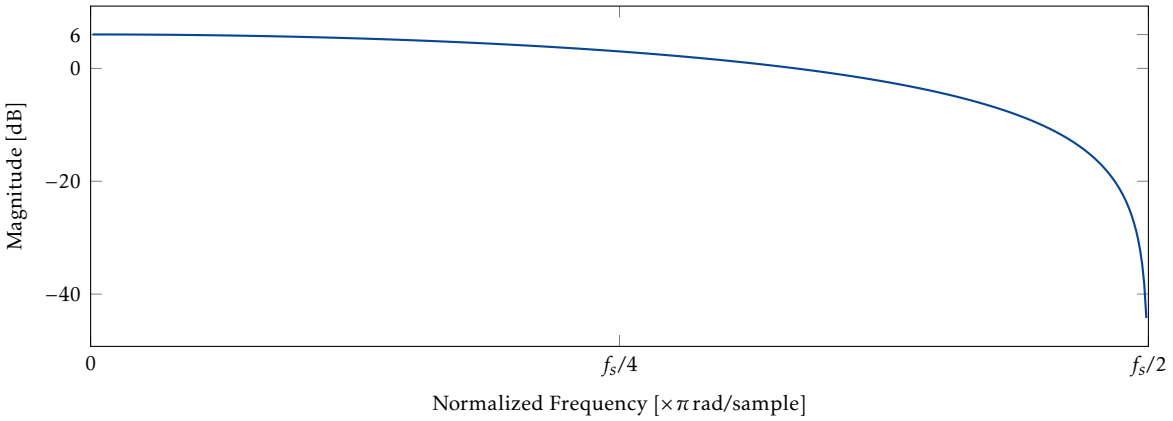


Figure A.2: Frequency response of a CIC filter with $N = 1$, $M = 1$, $R = 2$. Note the DC gain of 2.

running forever. In practice, however, it wraps around once it has reached its maximum representable value (`011` = 3) and begins counting from its lower numerical limit (`100` = −4) again. This cycle keeps repeating as long as the filter is running.

The ingenuinty of the CIC filter lies in exploiting the fact that this wraparound is irrelevant to the comb stage. Whether the comb stage calculates the difference between an integrator's value whose precision is unbounded or whether it calculates the difference between two values which have potentially been wrapped is without consequence.

As a demonstration of this effect, we shall examine the computation step of cycle $n = 4$ from Table A.1 (which contains the entire filter's state for 15 steps). At this point, the state of the filter's output is as follows (represented in three-bit two's complement and decimal):

$$\begin{aligned} OUT_{COMB}[4]_\mathrm{b} &= OUT_{INT}[4] - OUT_{INT}[2] \\ &= 101_\mathrm{b} - 011_\mathrm{b} \\ &= 010_\mathrm{b} \end{aligned} \tag{A.3}$$

$$\begin{aligned} OUT_{COMB}[4]_\mathrm{d} &= -3_\mathrm{d} - 3_\mathrm{d} \\ &= -6_\mathrm{d} \end{aligned} \tag{A.4}$$

Obviously, the decimal and binary results do not match. This is where the wraparound comes into play, for which we shall look at Figure A.3. The figure presents a circular arrangement for all numbers in two's complement with three digits precision. In that arrangement, addition of a positive number corresponds to moving clockwise through the circle, while subtraction of a positive number means moving counterclockwise. Doing this for the calculation of Equation A.4, we find that moving by three in the counterclockwise direction lands us at 2, exactly as Equation A.3 demands. The computation has *wrapped around* its boundary (−4).

What is left to verify is that the result of Equation A.3 is indeed the correct result, i.e. if an unbounded integrator and comb would have yielded the same outcome. And indeed, they would have:

$$OUT_{COMB}[6] = 7 - 5 = 2 \tag{A.5}$$

As a last step to confirm our results, the CIC filter of this exercise is simulated with Simulink. Its block design is given in Figure A.4. All blocks are set to two's complement with three digits of precision (no fractional bits).

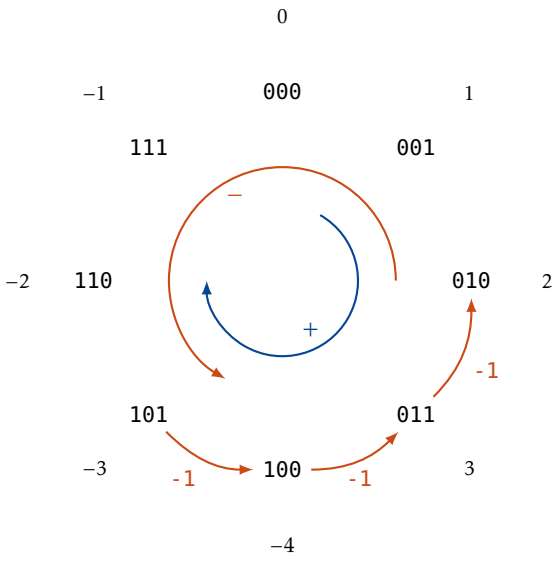


Figure A.3: Subtracting 3 from −3 in two's complement with three digits precicion, represented on a circle. Addition of a positive number corresponds to moving clockwise, subtraction of a positive number corresponds to moving counterclockwise.

Table A.1: Binary and decimal values for the different filter elements during various stages of the filtering process. As expected due to the filter's DC gain of 2, its output is indeed 2. The two right columns contain the calculations as they would occur if the filter's components had unbounded precision. It can be seen that the wraparound effect of the two's complement representation does indeed not change the filter's output.

Cycle	IN	OUT_{INT} IN_{COMB}	OUT	OUT_{INT} (unbounded)	OUT (bounded integrator)	OUT (unbounded integrator)
-2	000	000	000	0		
-1	000	000	000	0		
0	001	001	001	1	$1-0=1$	$1-0=1$
1	001	010		2		
2	001	011	010	3	$3-1=2$	$3-1=2$
3	001	100		4		
4	001	101	010	5	$-3-3=-6=2_{\text{wr}}$	$5-3=2$
5	001	110		6		
6	001	111	010	7	$-1-(-3)=2$	$7-5=2$
7	001	000		8		
8	001	001	010	9	$1-(-1)=2$	$9-7=2$
9	001	010		10		
10	001	011	010	11	$3-1=2$	$11-9=2$
11	001	100		12		
12	001	101	010	13	$-3-3=-6=2_{\text{wr}}$	$13-11=2$
13	001	110		14		
14	001	111	010	15	$-1-(-3)=2$	$15-13=2$

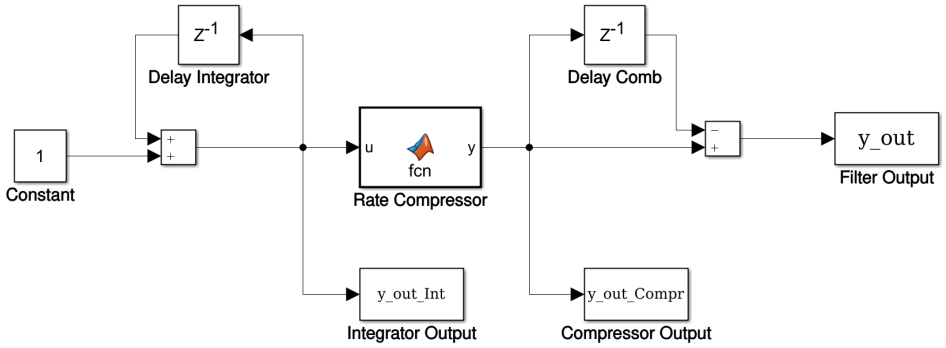


Figure A.4: Simulink model for the filter in Figure A.1. The Rate Compressor is a simple Matlab function which returns every second value of its input vector.

Table A.2: The same CIC filter as before stimulated with an input of 2. A comparison between the right two columns shows that the output of the bounded filter and its unbounded counterpart no longer match starting with the output at $n = 2$; the filter produces a false output.

Cycle	IN	OUT_{INT} IN_{COMB}	OUT	OUT_{INT} (unbounded)	OUT (bounded integrator)	OUT (unbounded integrator)
-2	000	000	000	0		
-1	000	000	000	0		
0	010	010	010	2	$2-0=2$	$2-0=2$
1	010	100		4		
2	010	110	100	6	$-2-2=-4$	$6-2=4$
3	010	000		8		
4	010	010	100	10	$2-(-2)=-4$	$10-6=4$

The simulation results are given in Figure A.5. As can be seen, the filter states are identical to our manually calculated example. The effect of the integrator's output wrapping around the numerical boundaries is also nicely visible.

TODO: Give file path

Lastly, it is shown what happens when the expected output of the filter exceeds the numerical range available. This is accomplished by feeding a constant of 2 into the filter; the expected output is therefore 4.

Because 4 is not within the range of a three-digit two's complement number system, the filter wraps around and produces an incorrect output, -4. The first few calculation steps for this are presented in Table A.2; running the above simulation with the modified input also comfirms this result.

A.2 CIC Filter Tables

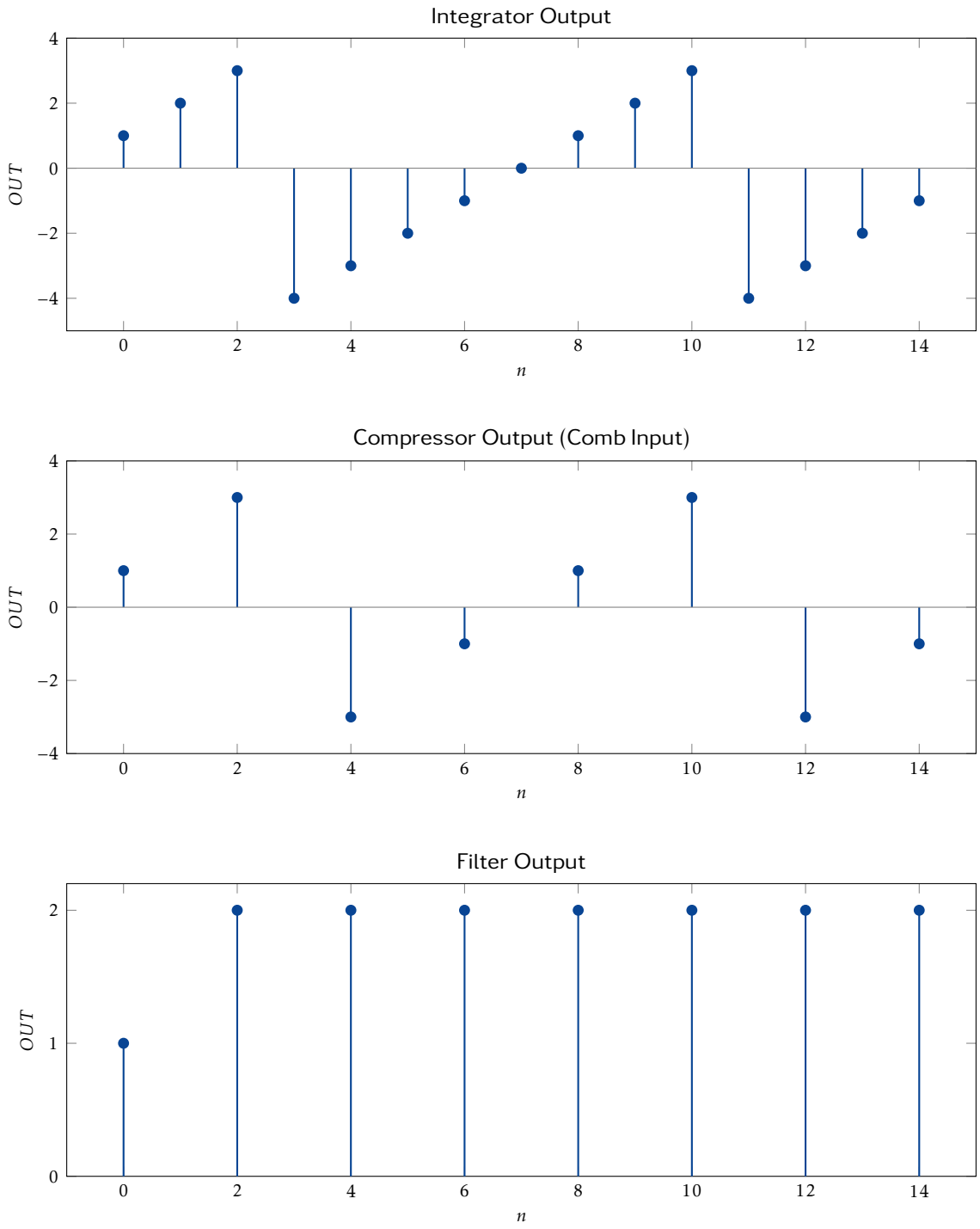


Figure A.5: Simulation results from Simulink for the filter in Figure A.4.

Table A.3: Passband attenuation for CIC filters as a function of the bandwidth-differential delay product. Taken directly from [**?**].

Relative Bandwidth-Differential Delay Product (Mf_c)	Passband attenuation at f_c in dB as a Function of Number of Stages (N)					
	1	2	3	4	5	6
1/128	0.00	0.00	0.00	0.00	0.00	0.01
1/64	0.00	0.01	0.01	0.01	0.02	0.02
1/32	0.01	0.03	0.04	0.06	0.07	0.08
1/16	0.06	0.11	0.17	0.22	0.28	0.34
1/8	0.22	0.45	0.67	0.90	1.12	1.35
1/4	0.91	1.82	2.74	3.65	4.56	5.47

Table A.4: Passband aliasing attenuation for CIC filters as a function of the bandwidth and the differential delay. Taken directly from [**?**].

Differential Delay (M)	Relative Bandwidth (f_c)	Aliasing/Imaging Attenuation at $f_{s,low}$ in dB as a Function of Number of Stages (N)					
		1	2	3	4	5	6
1	1/128	42.1	84.2	126.2	168.3	210.4	252.5
1	1/64	36.0	72.0	108.0	144.0	180.0	215.9
1	1/32	29.8	59.7	89.5	119.4	149.2	179.0
1	1/16	23.6	47.2	70.7	94.3	117.9	141.5
1	1/8	17.1	34.3	51.4	68.5	85.6	102.8
1	1/4	10.5	20.9	31.4	41.8	52.3	62.7
2	1/256	48.1	96.3	144.4	192.5	240.7	288.8
2	1/128	42.1	84.2	126.2	168.3	210.4	252.5
2	1/64	36.0	72.0	108.0	144.0	180.0	216.0
2	1/32	29.9	59.8	89.6	119.5	149.4	179.3
2	1/16	23.7	47.5	71.2	95.0	118.7	179.3
2	1/8	17.8	35.6	53.4	71.3	89.1	106.9

APPENDIX B

Filter Design

This chapter contains some additional information about the filter design process.

B.1 Decimation of 625: Variants

B.2 Resource Usage for FIR Filters on the FPGA

This section contains the experimental results of how many DSP slices a given FIR filter needs when implemented with Xilinx's FIR Compiler. These figures form the basis to determine reasonable bounds for the FIR filter `5steep` (see Figure 3.1 on page 38). Figure B.2 depicts the results of the measurements, while Table B.1 contains the configuration parameters which were used for the FIR compiler core.

As can be seen in the plot, DSP slice usage rises roughly linearly at these high sampling rates. When using only a single filter, a filter of roughly 760 coefficients is the maximum possible size. Because the STEMlab has two channels, and because other filters are required as well, an upper limit for `5steep` of 250 coefficients is set based on these results. A smaller filter is also acceptable as long as it fulfills the general requirements.

TODO: source FIR compiler documentation

B.3 Halfband Filters

Table B.1: The parameters used to configure the FIR compiler core for the usage measurements from Figure B.2

Parameter	Value
Clock Frequency	125 MHz
Decimation Rate	5
Input Data Width	24 bit
Input Fractional Bits	7
Output Data Width	32 bit
Coefficient Fractional Bits	17

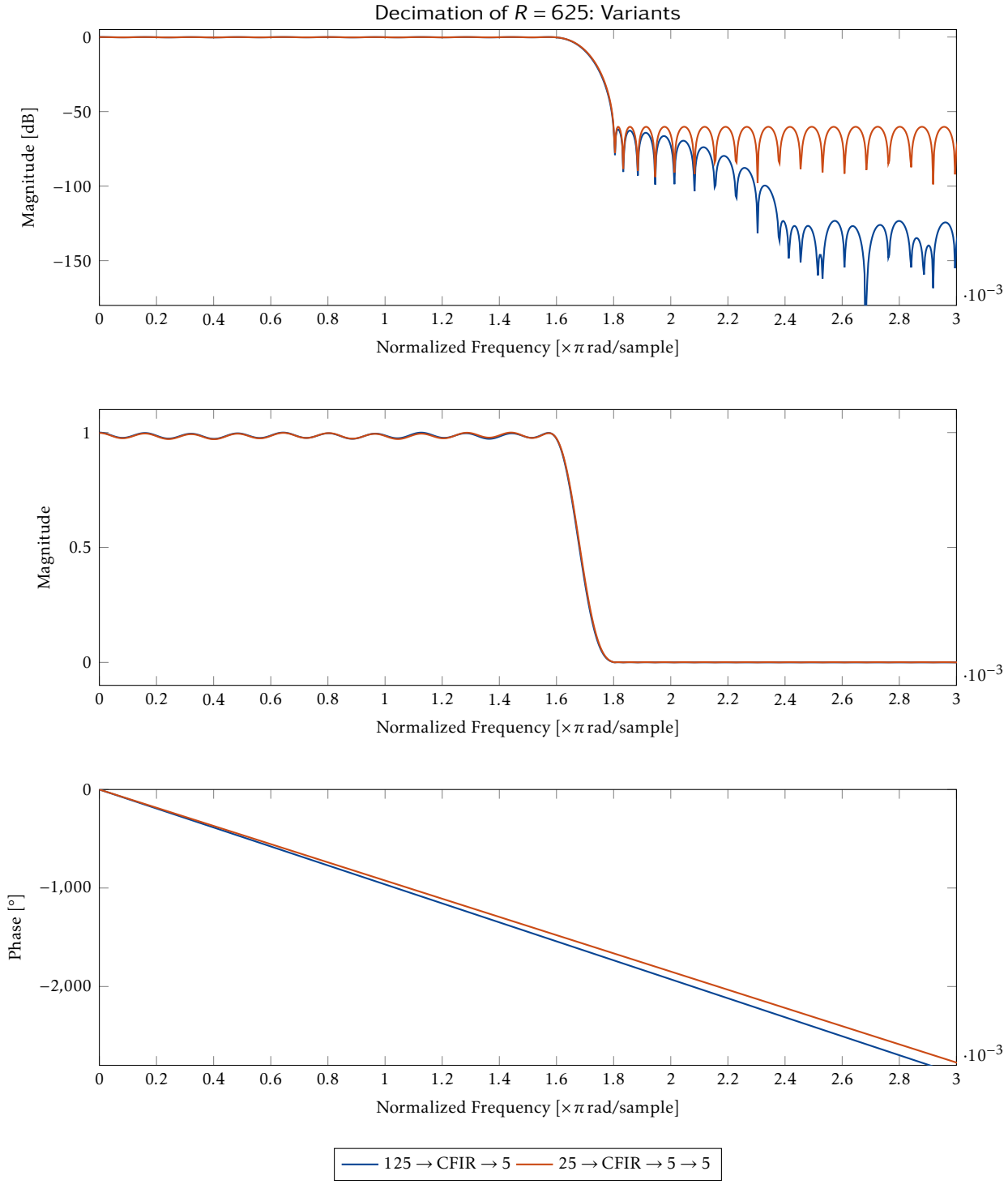


Figure B.1: Semilog (top) and linear plot (middle) for two variants for implementing a decimation chain for a rate change factor of $R = 625$. Both choices show almost the same behavior with regards to magnitude. The bottom plot shows the phase response of the two filters; here, too, the behavior is very similar. *Note:* These plots show the frequency responses of the entire filter cascade for the two respective variants.

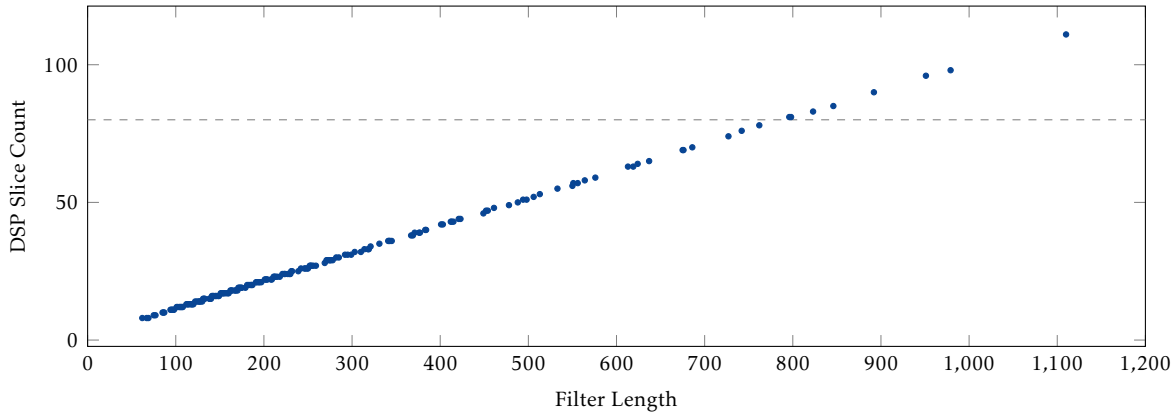
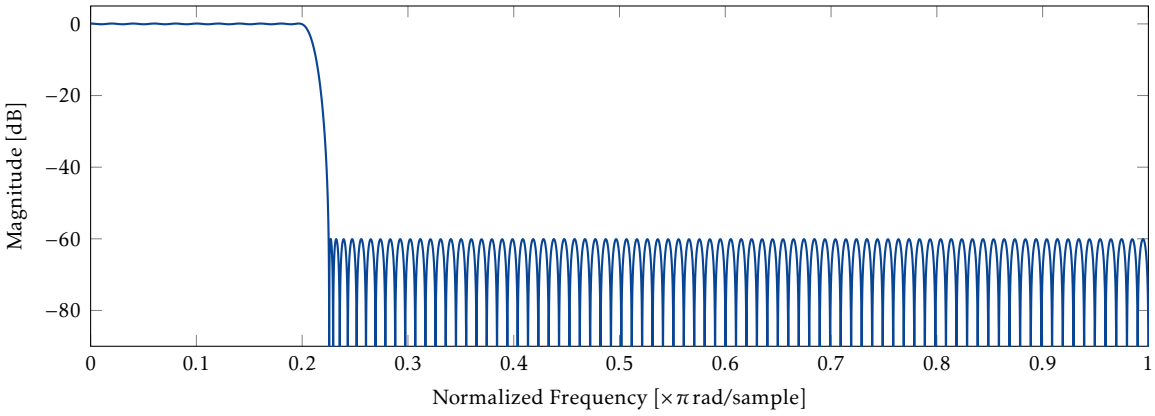


Figure B.2: Usage report figures for DSP slices using the Xilinx FIR compiler block. The configuration of the filter in terms of bith widths is identical to the actual configuration used in the final implementation. Unlike the implementation, however, only a single channel was configured.

B.4 Filter Frequency Responses

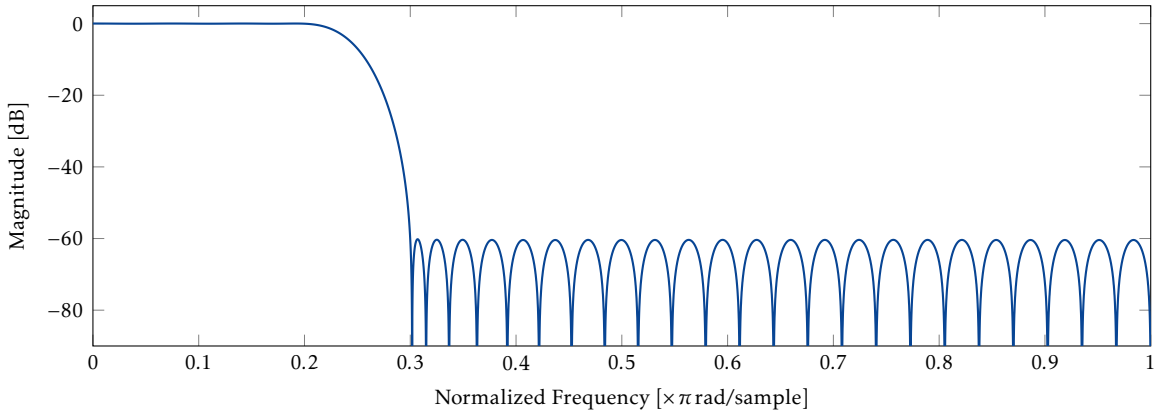
This section contains the frequency responses of the filters and the cascades as specified in Section 3.

B.4.1 `5steep`

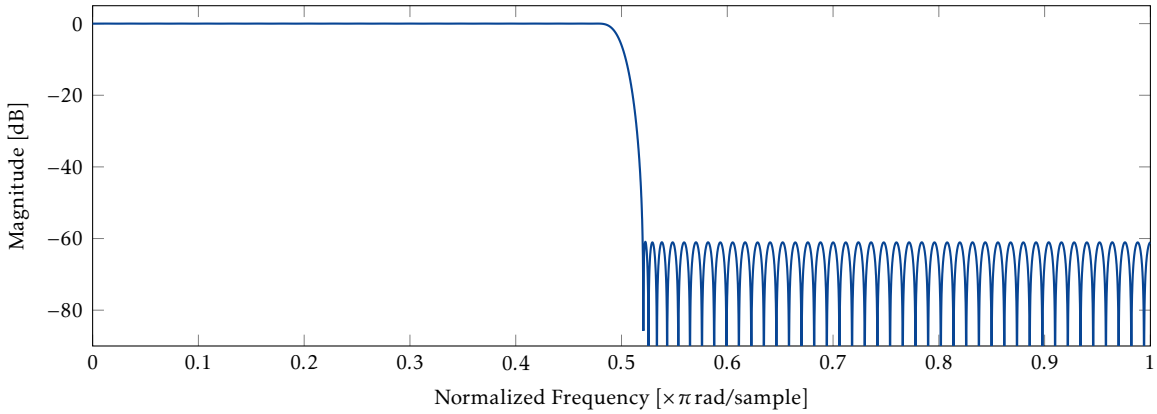


B.4.2 5flat

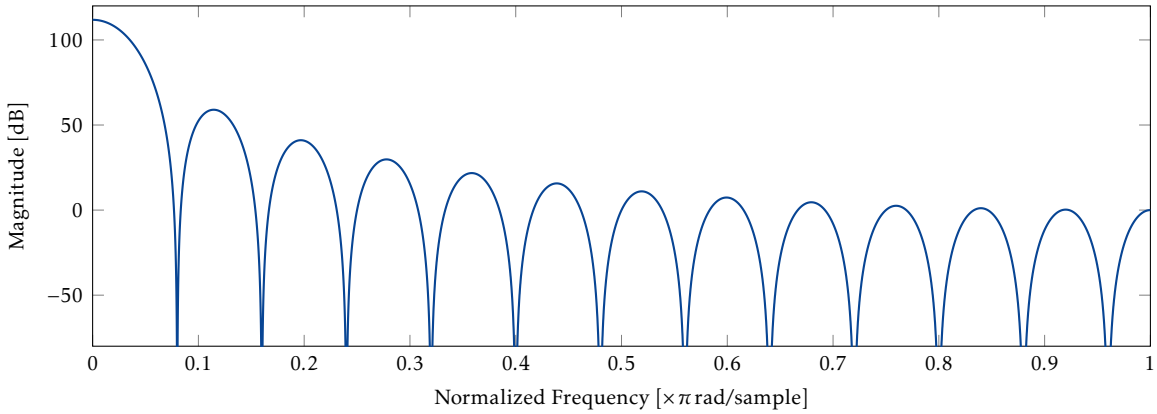
Filter Characteristics	
Filter Length	62
Passband Edge	0.2
3 dB Point	0.23491
6 dB Point	0.24708
Stopband Edge	0.3
Passband Ripple (dB)	0.047745
Stopband Attenuation (dB)	60.1991
Transition Width	0.1
Number of DSP Slices	22



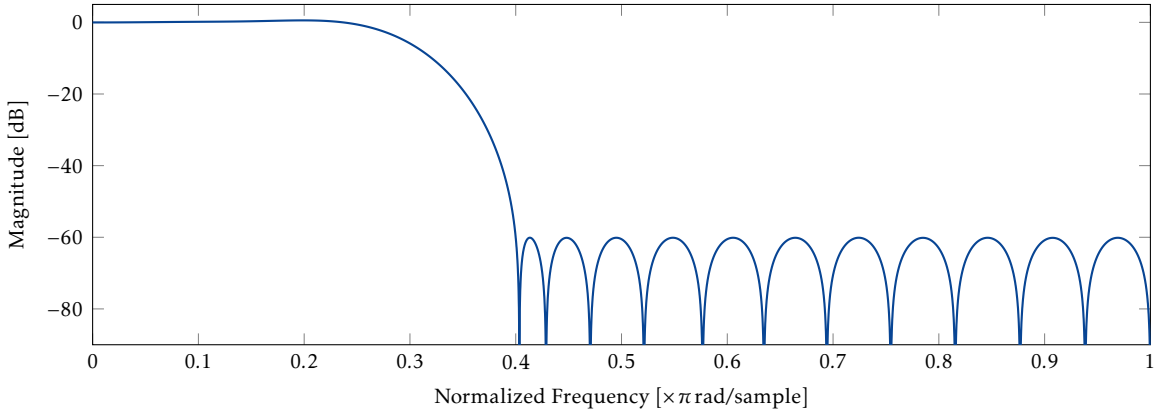
B.4.3 2steep



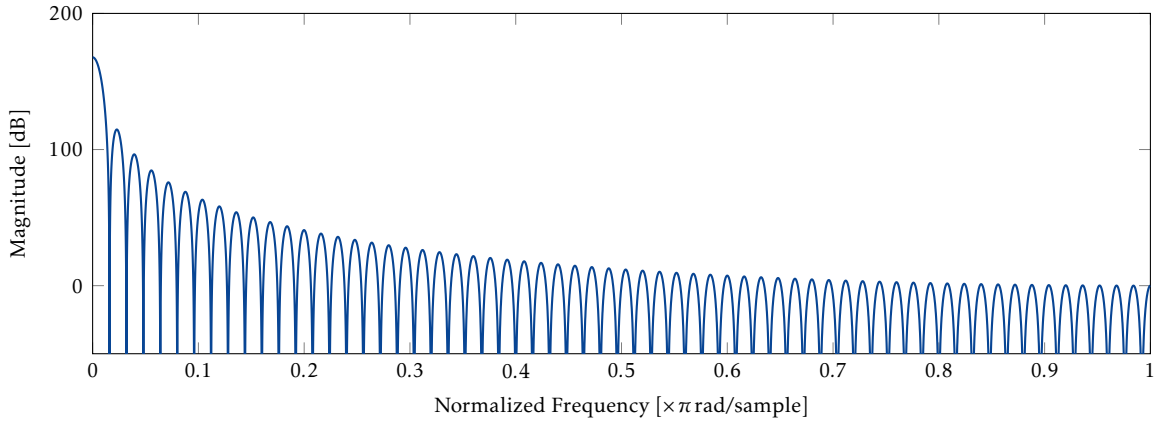
B.4.4 CIC25



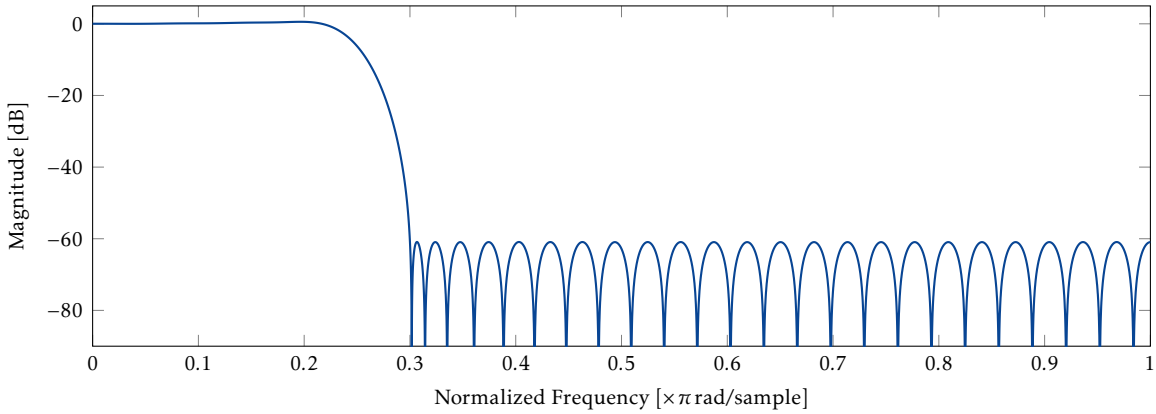
B.4.5 CFIR25



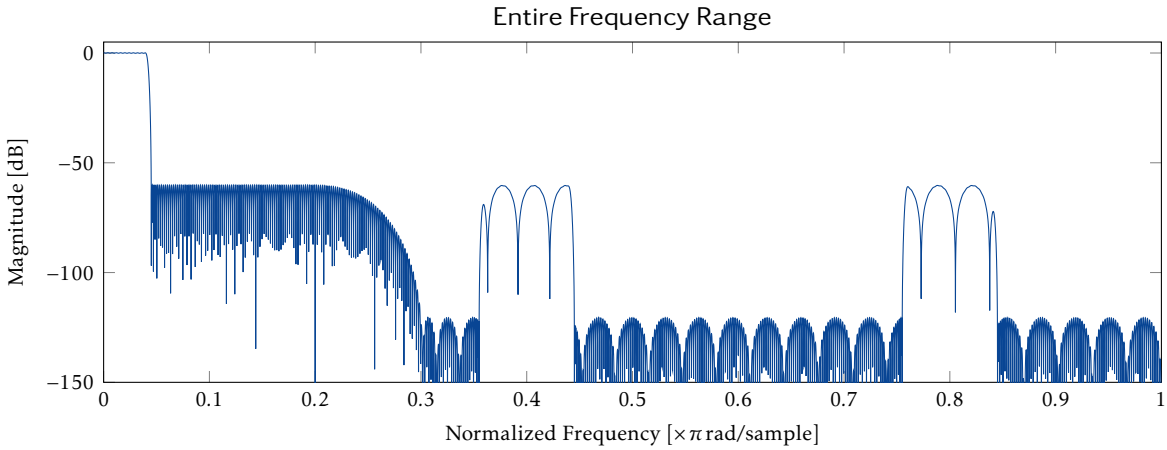
B.4.6 CIC125

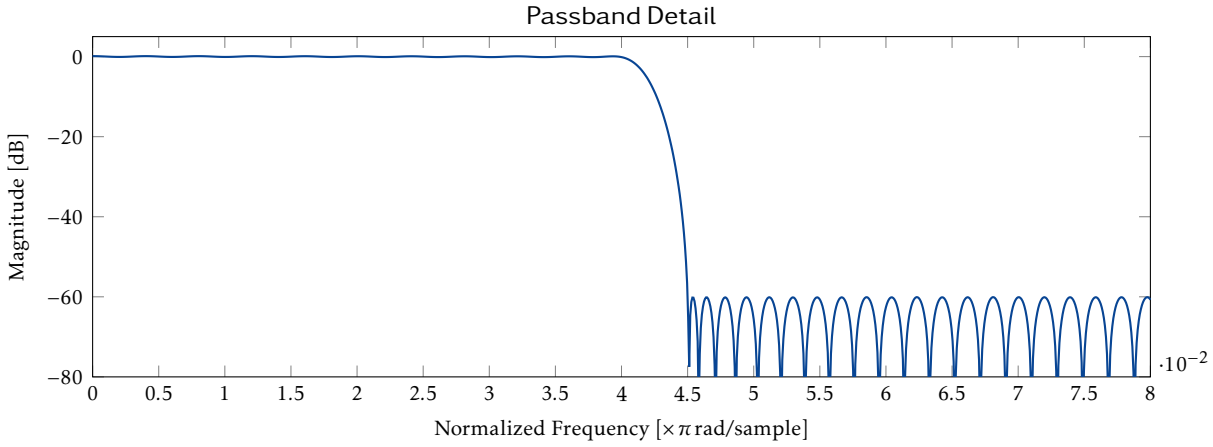


B.4.7 CFIR125

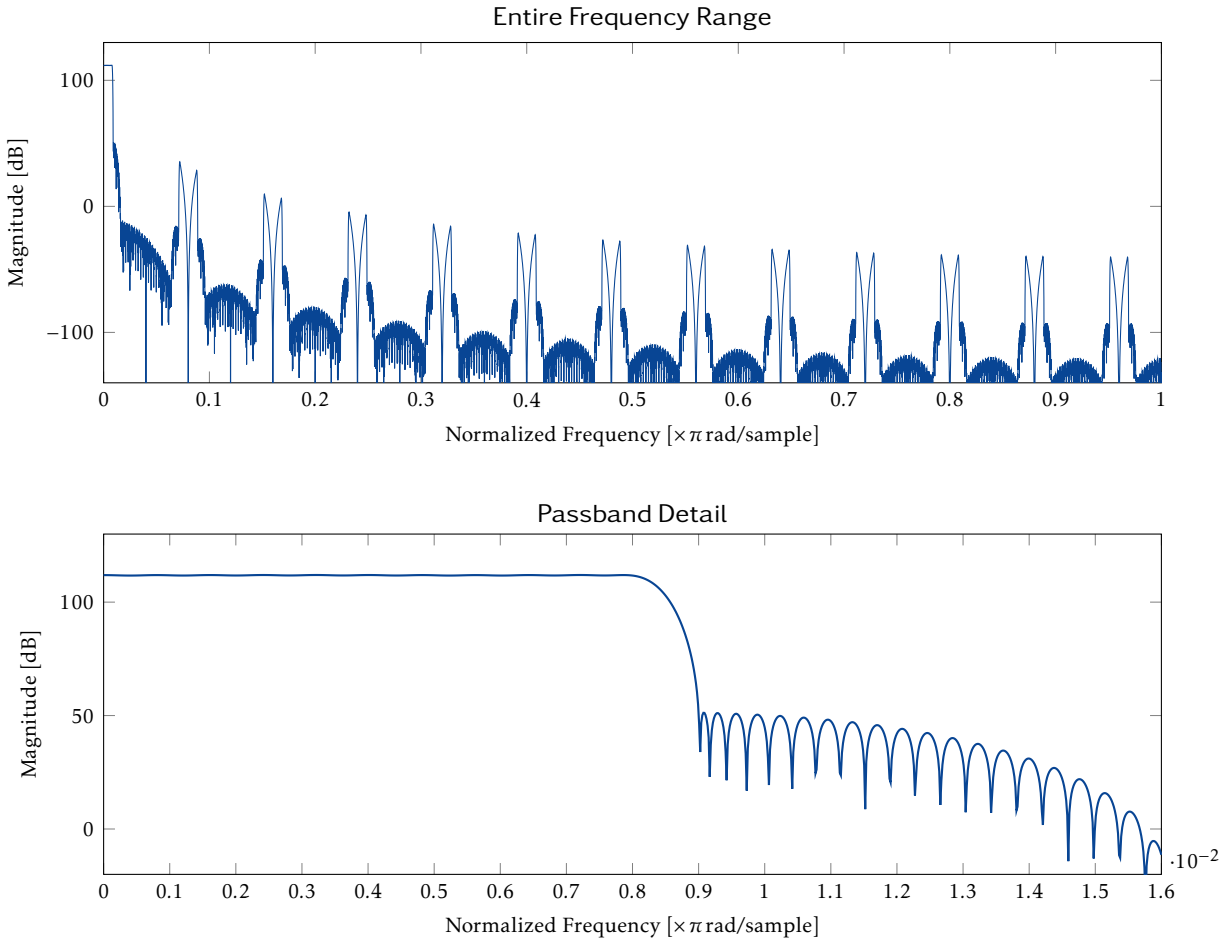


B.4.8 Chain for R=25

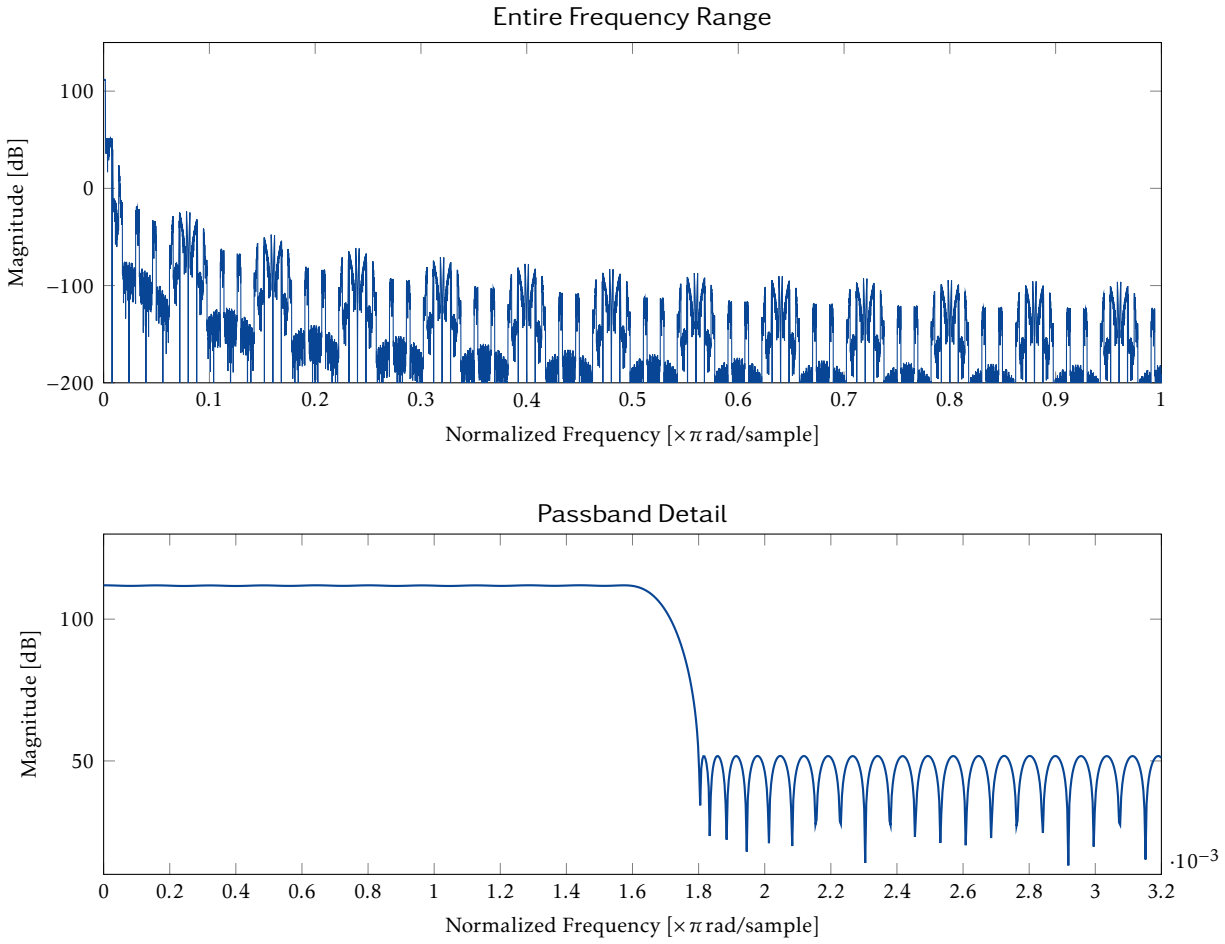




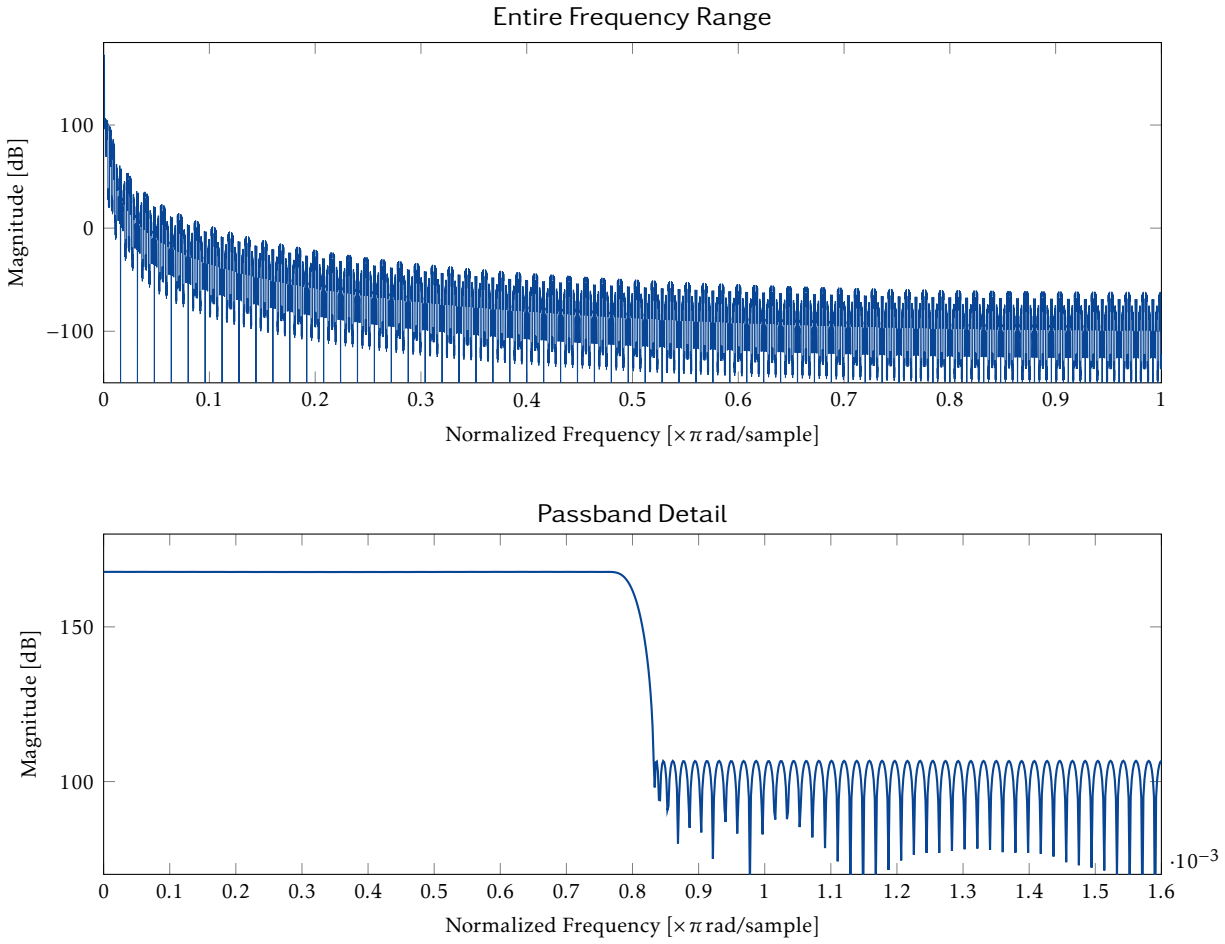
B.4.9 Chain for R=125



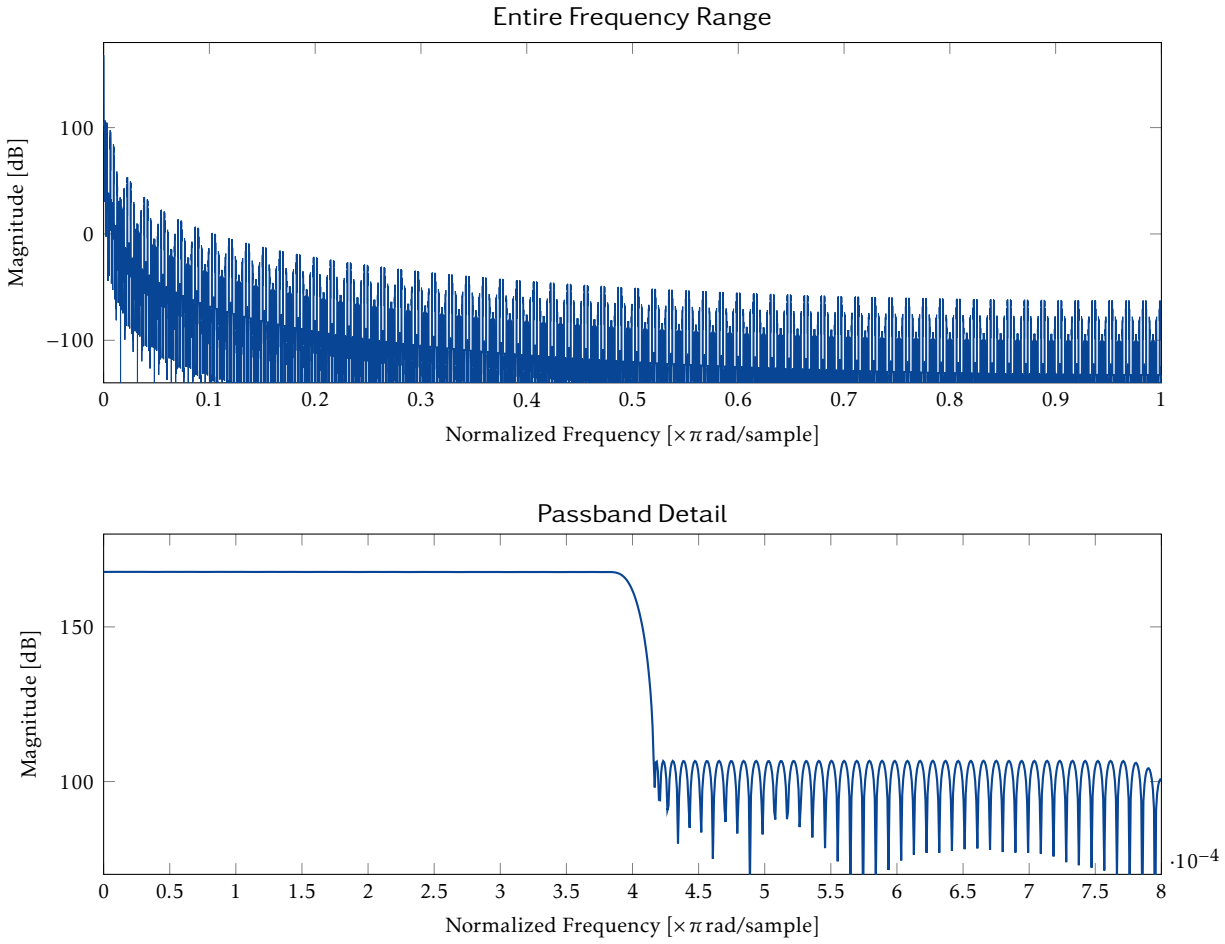
B.4.10 Chain for R=625



B.4.11 Chain for R=1250



B.4.12 Chain for R=2500



APPENDIX C

Licenses

C.1 MIT License (Source: [?])

```
Copyright <YEAR> <COPYRIGHT HOLDER>

Permission is  hereby granted, free of  charge, to any person  obtaining a
copy of this software and associated documentation files (the "Software"),
to deal in the Software  without restriction, including without limitation
the rights to  use, copy, modify, merge,  publish, distribute, sublicense,
and/or sell  copies of  the Software,  and to permit  persons to  whom the
Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT  NOT LIMITED TO THE  WARRANTIES OF MERCHANTABILITY,
FITNESS FOR  A PARTICULAR PURPOSE  AND NONINFRINGEMENT. IN NO  EVENT SHALL
THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER  IN AN ACTION  OF CONTRACT, TORT OR  OTHERWISE, ARISING
FROM,  OUT OF  OR IN  CONNECTION WITH  THE SOFTWARE  OR THE  USE OR  OTHER
DEALINGS IN THE SOFTWARE.
```