Master's Thesis

Holistic Low-power IIR Filter Design for Wireless Communication Receivers using Differential Evolution

conducted at the
Signal Processing and Speech Communications Laboratory
Graz University of Technology, Austria

in co-operation with
Maxim Integrated GmbH
Lebring, Austria

by
Alexander Melzer

Supervisors:
Dipl.-Ing. Dr.techn. Manfred Mücke
Dipl.-Ing. Andreas Pedroß
Assoc.Prof. Dipl.-Ing. Dr. Klaus Witrisal

Dipl.-Ing. Bernd Janger (Maxim Integrated GmbH)
Dipl.-Ing. (FH) Alexander Resch (Maxim Integrated GmbH)

Graz, October 2, 2012

Abstract

Digital filtering has become a key building block of digital signal processing and therewith a wide range of utilization within wireless communication systems emerged. In general, filter design relies on given filter specifications either to meet frequency response or time-domain characteristics. However, in order to optimize a filter for a certain application, the overall system in which the filter is embedded needs to be considered. Both approaches are investigated in this work.
Even if there are many beneficial properties compared to analog filters, with digital filters one has to deal with the effects of quantization. This nonlinear operation makes it difficult to find an optimized set of filter coefficients for implementation in finite precision hardware. In this work a differential evolution optimization algorithm is employed to tackle this issue. It takes either the system model of an existing RF receiver for standard industrial, scientific and medical (ISM) bands or an filter specification as input. Furthermore, a toolset for an automatized design flow is proposed and implemented. It delivers quantized coefficients for infinite impulse response filters that are optimized for low-power/low-area and are plugged directly into the hardware description language model. It is shown that filter design based on a system model improves the overall receiver performance significantly.

Kurzfassung

Digitale Filtertechniken entwickelten sich zu einem wichtigen Baustein für die digitale Signalverarbeitung und daraus entstand eine Vielzahl an Anwendungen in drahtlosen Kommunikationssystemen. Im Allgemeinen beruht der Filterentwurf auf gegebenen Filterspezifikationen, entweder um Charakteristiken im Zeit- oder Frequenzbereich zu erfüllen. Um hingegen einen Filter für eine gewisse Applikation zu entwerfen, muss das Gesamtsystem, in dem dieser eingebettet ist, betrachtet werden. Beide Ansätze werden in dieser Arbeit aufgegriffen.
Auch wenn digitale Filter eine Reihe vorteilhafter Eigenschaften gegenüber analogen aufweisen, so müssen Effekte der Quantisierung berücksichtigt werden. Diese nichtlineare Operation erschwert die Optimierung von Filterkoeffizienten für die Implementierung in Hardware mit endlicher Genauigkeit. In dieser Arbeit wird ein *Differential Evolution* Optimierungsalgorithmus eingesetzt um dieses Problem zu lösen. Dabei wird entweder das Systemmodell eines existierenden RF Empfängers für die industriellen, wissenschalftlichen und medizinischen (ISM) Bänder oder eine Filterspezifikation als Optimierungsbasis verwendet. Weiters wird ein automatisierter Entwurfsablauf vorgeschlagen und implementiert. Dieser liefert quantisierte Koeffizienten welche für geringen Strom- und Flächenbedarf optimiert sind und direkt in das Hardwarebeschreibungsmodell übernommen werden. Es wird erwiesen, dass der Filterentwurf basierend auf einem Systemmodell die Leistungsfähigkeit des Empfängers maßgeblich verbessert.

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

date	(signature)

Acknowledgements

First and foremost, I would like to thank my advisors from the Signal Processing and Speech Communication Laboratory, especially Andreas Pedroß for his commitment to support me with any technical issue and the most valuable inputs he contributed to this thesis. I also wish to thank Klaus Witrisal and Manfred Mücke for supporting and guiding me into the right direction throughout this work.

Furthermore, I would like to thank Bernd Janger not only for enabling and supporting the thesis at Maxim Integrated but also I highly appreciated the flexibility granted to fit work around my studies. Thanks also to Alexander Resch and Steven Dennis for sharing their expertise with respect to the implementation details.

Finally, I wish to thank my parents for supporting me in every way throughout my whole studies.

Contents

1 Introduction

Keyless go systems have gained significant interest in the last years. Without grabbing the key from the pocket, the driver just needs to carry it in order to open the car. A further feature is that an ignition lock is no longer required, it is substituted by the start button. The engine can be started as soon as the key is located inside the car. There is huge potential for implementing additional comfort as well as security functions in *keyless go* systems.

The hardware setup consists of at least four antennas mounted at the doors, the trunk and another one inside the car to detect the key as illustrated in Figure 1.1. It is therefore possible to detect the actual location of the key, especially if it is in- or outside the car.

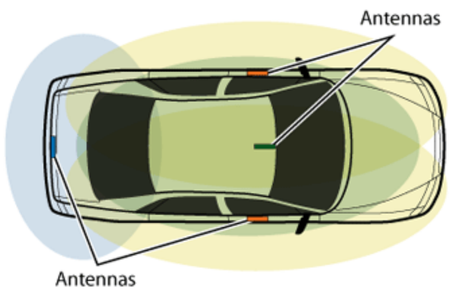


Figure 1.1: Antennas mounted on a car for a keyless go application [1]

The transceiver system consists of a key and car unit as depicted in Figure 1.2. For wireless wakeup and communication, an LF/RF link is used. The LF is ranging from 20 kHz to 125 kHz whereas for RF standard ISM bands 315, 433, 868 and 915 MHz are used (geographical location dependent). LF packets with a predefined codeword are sent out from the car antennas periodically that wakup the key unit. Over the RF link the data communication is established and a challenge response authentication takes place. A controller transfers the received data to the immobilizer which verifies the data before the access is granted finally.

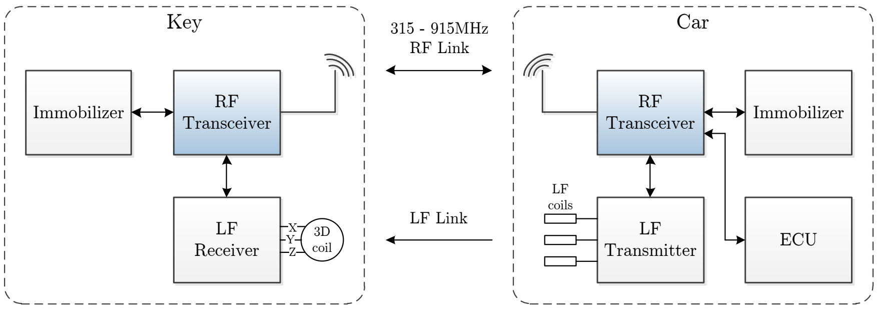


Figure 1.2: Keyless go system

Maxim Integrated offers customized *keyless go* solutions with advanced wakeup detection range for convenient keyless experience. This is achieved primarily due to a 22 kHz operating frequency. However, this thesis will mainly deal with the RF receiver implemented in such systems. It is an application specific standard product (ASIC) with an internal microprocessor that is used as data link for the authentication process. A digital filter is designed to improve the overall system performance based on the given receiver structure.

1.1 DSP Filter Design

Digital filtering is a valuable tool of digital signal processing (DSP). Compared to analog filters, which are built from passive devices such as resistors, capacitors and inductors, digital filters consist of standard logic cells. On a transistor level circuit these cells are combined to create adders and multipliers which are fundamental building blocks for any DSP[1] design. This implies that all the computations are done in discrete amplitude and time.

There are some very attractive advantages compared to analog filters such as independence on production process fluctuations, no calibration after production and easy configuration of filter coefficients during runtime. However, one of the main disadvantages is the finite number space in which the computations are performed in digital hardware. It causes quantization noise, stability issues and a limitation of the frequency range due to sampling

[1] As this work deals with filter design in ASICs, we stick to digital signal processing and the design itself (instead of implementation on a processor platform).

and the resulting periodic repetition of the frequency spectrum. Number format and arithmetic are therefore very important design decisions that need to be evaluated already at system-level design.

In a DSP design methodology a standard approach to design filters is to use specifications that are coming out from a system-level implementation. They are specified most likely by frequency response parameters such as cutoff frequency or stopband attenuation. There are several tools and algorithms available to compute exact coefficients for this design problem. However, the issue with digital filters is that quantization, a nonlinear operation, needs to be considered. Several approaches to tackle this problem emerged as digital filtering became popular. Many of these algorithms employ population based heuristic optimization algorithms as they are able to optimize several costs simultaneously for a certain set of candidate solutions.

An extended approach is to consider the target filter within the overall system. By modelling the actual RF receiver, the filter can be optimized for performance measures such as bit error probability. Although there is a lot of additional design effort, this method promises to achieve higher receiver performance as long as the system environment is modelled well.

In order to optimize the filter design for area and power of the final implementation in hardware, partial cost functions need to be employed at system-level optimization. This needs to be handled by both of the above mentioned design approaches. The costs are affected mainly by the filter structure and the arithmetic/number format in which the computations are performed.

1.2 Objectives

The main purpose of this work is to design and implement a filter in digital hardware that replaces an existing filter with the goal to optimize the RF receiver in terms of error probability. The existing filter has been designed based on a certain lowpass filter specification that came out of the system design. In order to achieve the optimal criterion, the overall system needs to be considered and the filter designed within its actual boundaries. From the implementation point of view the hardware description language (HDL) model needs to improve the existing design with respect to area and power as the filter is used in a battery supplied device.

Even more important is the reuseability as the filter should be able to be instantiated as intellectual property (IP) with different filter configurations. This demands a tool to calculate filter coefficients under certain constraints that is embedded in a fully automatized design toolflow.

As for any DSP design the effects of quantization need detailed investigations. Also, a reasonable trade-off between receiver performance and area/power of the design needs to be met. Two major number formats in DSP are compared, namely fixed-point and floating-point. Nevertheless the final implementation will be in fixed-point arithmetic. As MATLAB supports the auto-generation of very high speed integrated circuit hardware description language (VHDL) models out of its filter design and analysis tool, it is evaluated whether an automatic code generation on this basis makes sense.

The actual filter implementation is a completely standalone IP in 0.18 μm complementary metal oxide semiconductor (CMOS) technology. The coefficients can be determined by either the system model or a filter specification as illustrated in Figure 1.3. The implemented filter optimization tool takes any of these two inputs as a basis for computing the coefficients together with a filter architecture and certain constraints that define the cost functions for optimization. Anyway, the HDL design only needs the coefficients, the implementation stays the same for all these approaches. The filter optimization tool is embedded in a fully automatized design flow from filter/system specification down to register transfer level (RTL) and gate level (GL). Equivalence checks are performed automatically to a bittrue high-level model on all levels. For the hardware description there are two implementation variants demanded. Firstly, a variant where the coefficients are fixed at synthesis/compile time, and secondly a variant where coefficients are programmable during runtime. Low-power HDL techniques need to be investigated in order to optimize the costs for area and power also at implementation level.

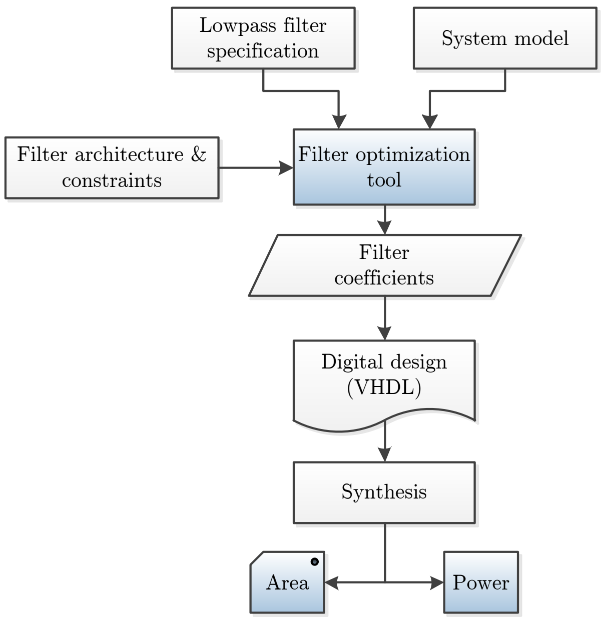


Figure 1.3: Simplified filter optimization tool flow

1.3 Related Work

Infinite impulse response filters provide much better performance with less computational/hardware requirements compared to finite impulse response filters. The disadvantage is that the error surface for the digitalized filter is nonlinear and multi-modal [2]. Furthermore, they might become unstable. Most common traditional designs of digital IIR filters rely on bilinear transformation. Based on an analog (continuous-time) reference filter transfer function the approximately equivalent digital (discrete-time) filter is designed. This holds for standard filter types, i.e. lowpass, highpass, bandpass and bandstop.

Besides this standard approach, many different design methods evolved as digital filter design became more and more valuable in the past decades. Matching a certain frequency response specification is still the major target, however there are also interesting approaches that design the filter in time-domain. Both of these methods are investigated in this work. The former is used to design a filter given a certain filter specification whereas the later approach does so by considering the whole receiver.

Especially for IIR filters designed to meet certain filter specifications such as passband ripple, cutoff frequency, stopband frequency or stopband attenuation, it is common to adjust poles and zeros in the z-plane as this automatically ensures stability [3]. It has been shown that ripples of the magnitude can be controlled by the radii of the poles, whereas the unevenness, i.e. the difference between the two extremes to the right/left of a ripple peak, is affected by the phase of poles and zeros. For hardware implementations it can be said that filters with less poles than zeros deliver promising solutions. Also, phase distortions and roundoff errors might be reduced in this case [4]. However, for a digital implementation, coefficients still need to be calculated out of the poles/zeros and quantized afterwards. Furthermore, depending on the filter structure, scaling factors that are used to limit the numerical range for finite-wordlength architectures do not come out directly from this design in the z-plane. Even if the objective to meet a certain magnitude response is the same, in contrast to [3] this approach adjusts the coefficients of the filter directly in order to avoid this issue. Of course, to verify stability of these coefficients, poles and zeros need to be computed. In fact, this is a simple operation with low computational effort.

Designing the filter in time-domain can be incorporated by trying to match, for instance, a reference impulse response. The design method in [5] even computes full quantized filter coefficients. After initially only unquantized coefficients are taken into account, they are substituted step by step by quantized ones inbetween optimization procedures. Besides the fact that the implementation is simple, quantization errors can be evaluated convenient in time-domain. Furthermore, computations of the impulse response can be done fully bittrue such that any numerical overflow is indicated immediately. In this work the same approach is used to optimize the overall RF receiver. Still, instead of taking the impulse response, a data sequence is generated as reference out of the system model where the target is to minimize the bit errors.

From the architecture point of view most design methods are not specific as the design is done in the z-plane [6]. However, in DSP designs the cascading of second-order sections (SOS) has emerged as the coefficient range and therefore also the numerical range for the computations is limited. Therewith, even quantization errors are reduced which is the main purpose of utilizing SOS in this work [7]. There are lattice wave filter implementations as well [8, 9], but there is the issue that in general this filter structure needs the double amount of multipliers compared to direct-form II implementations [10].

Multiplierless IIR filters are very interesting for hardware implementations. The difficulty is that the frequency response is highly sensitive to the filter coefficients [11], thus the application is limited. In [12] a method is proposed to quantize both coefficients and state variables to power-of-2 values. This leads to simple shifting operations in hardware which reduces area and power tremendously. Despite, the relatively low stopband attenuation that is achieved with this approach might not be sufficient for some applications. The multiobjective optimization proposed in this work simultaneously adopts coefficients to achieve given filter characteristics while at the same time optimizes for area and power constraints. That is why a reasonable tradeoff can be selected between costs in digital hardware and filter characteristic.

Besides the filter architecture the arithmetic in which the computations are performed in digital hardware is a major design decision. For DSP designs there are various fixed- and floating-point implementations. Both number formats are capable to use (fractional) canonic signed digit (CSD) representation [13] which is used in many digital implementations due to its beneficial aspects regarding area and power consumption.

Above mentioned filter design methods rarely incorporate the effects of finite-precision computations as they focus on the filter design itself, i.e. are capable to find filter designs that meet given filter specifications. This is a major drawback of these design methods as the behaviour of the quantized filter in hardware might be significantly different. Thus, optimization techniques need to be taken into account whenever different criteria need to be met [3]. That is why a heuristic optimization method is used in this work for designing an optimum filter.

Heuristic optimization algorithms have matured and are commonly understood as powerful tool for various optimization problems. They have been widely and successfully applied also for digital filter design. Especially the use of population based computation algorithms such as genetic algorithms (GA) [14–16], differential evolution (DE) [2, 16–19], particle swarm optimization (PSO) [20] and ant colony optimization (ACO) is widespread. A very recent and novel approach, the seeker optimization algorithm (SOA) has been developed with convincing performance [21].

As in the previous section the target is to find a global optimum for the digital (quantized) filter coefficients within reasonable time. Single-state trajectory based methods such as hill climbing (HC), simulated annealing (SA) and tabu search (TS) are sensitive to the initial starting point although they exploit a local optimum very well [21]. On the other hand, population based optimization methods in general explore the overall search space well. That is why they are advantageous in contrast to single-state methods as they incorporate a whole set of individuals at the same time.

For standard GA it is claimed that they have two major drawbacks. They lack in local search ability and premature convergence [2]. That is why [14] uses an SA in addition to the GA. It improves the local search abilities and therefore a tradeoff between exploration and exploitation is found. In contrast to that, DE has shown very well performance for filter design optimization problems. Especially remarkable is that a standard DE converges faster[2] with lower variance over different runs than other, much more complex approaches [21]. For this reason a DE is utilized in this work.

Many of the algorithms referenced above have single cost functions they are trying to optimize. For instance, [2, 14, 21] try to match a given magnitude response of the filter. This design task can be simply reformulated to a system identification problem as shown in Figure 1.4. The cost function is defined as

$$J(\mathbf{c}) = \frac{1}{N}\sum_{n=1}^{N}(d[n] - y[n])^2 \tag{1.1}$$

where d and y represent the desired and actual filter response respectively and $\mathbf{c}$ represents the filter parameters to be optimized. This however is only a single objective optimization, no further constraints are taken into account. The approach has it's entitlement whenever a fixed system design or even a filter transfer function is given and needs to be matched as good as possible.

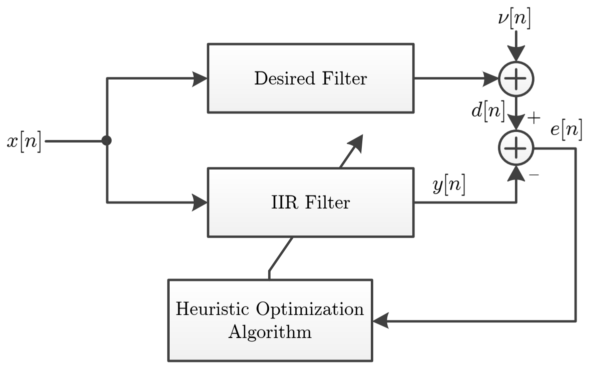


Figure 1.4: System Identification for Digital IIR Filter Design

By relaxing this fitting constraint to boundaries where the magnitude response is allowed to live in, partial costs can be taken into account. In a multiobjective optimization different cost functions of the objectives are computed and applied *simultaneously*. This technique has been successfully applied for some approaches already. In [15] they try to optimize

[2] Note that it is common to measure the convergence speed in number of generations instead of time

for the magnitude and phase response as well as the lowest filter order. In contrast to that, [17, 20] targets the magnitude response and the group-delay. This work extends the latter approach by adding dedicated cost functions for digital hardware implementation. On the one hand that is the area required by the filter (mainly for multiplication and registering the intermediate results) and on the other hand the power that is consumed (mainly affected by the chosen coefficients).

To sum up, there is a variety of algorithms to solve this optimization problem with high performance utilizing heuristics. Nevertheless these approaches consider a fixed system design, i.e. a given filter specification, and mostly a single cost function. The optimization problem extended to the overall system can be performed by using a DE again [22] as the filter coefficients are adopted once again. The multiobjective cost functions are reformulated to 1) the bit error rate of the system, 2) the area required by the filter and 3) the power consumption of the filter.

1.4 Outline

The thesis is organized as follows. Chapter 2 presents the system model and formulates the optimization problem. Chapter 3 proposes the filter optimization tool based on (arbitrary) filter specifications. Chapter 4 describes the toolflow for digital hardware design and provides analysis for different design options. Finally, Chapter 5 recaptures the results before in Chapter 6 a conclusion is drawn and an outlook for future work is presented. The appendix provides further details and insight to implemented and used tools.

2 System Model

This section presents the RF receiver in which the target IIR filter is implemented. The design and optimization of the filter based on the system model is derived, therefore a complex baseband equivalent model is investigated. The receiver performance is evaluated by comparing the existing filter implementation to the minimum-mean squared error (MMSE) solution.

2.1 The RF Receiver

In Figure 2.1 a simplified block diagram of the receiver part of the RF transceiver is illustrated. From the antenna port the signal enters a low noise amplifier (LNA). Passing through the down conversion mixers, the signal is split up into the in-phase (I) and quadrature (Q) paths. The signals are not directly downconverted to baseband but instead to a so-called intermediate frequency (IF). Therefore, the system will also be referred to as low-IF receiver. In a pair of variable gain amplifiers (VGA) the signal is amplified and bandpass filtered in order to scale the received signal to the dynamic range of the following analog to digital converters (ADC). These are sigma-delta ($\Sigma\Delta$) ADCs that provide a 1-bit datastream. In the digital domain a decimation filter reconstructs a digitalized waveform out of this oversampled bitstream. A highpass filter performs AC coupling before the downconversion to baseband including the upper/lower side band (USB/LSB) selection is done. This baseband signal is input for the target IIR filter that is mainly designed for image rejection. After decimation, a coordinate transformation computes magnitude and phase out of the real and imaginary parts of the received signal which are used for further processing like codeword correlation, bit clock recovery and finally data reception. The receiver is able to deal with amplitude shift keying (ASK) as well as frequency shift keying (FSK), however in this work only ASK is considered.

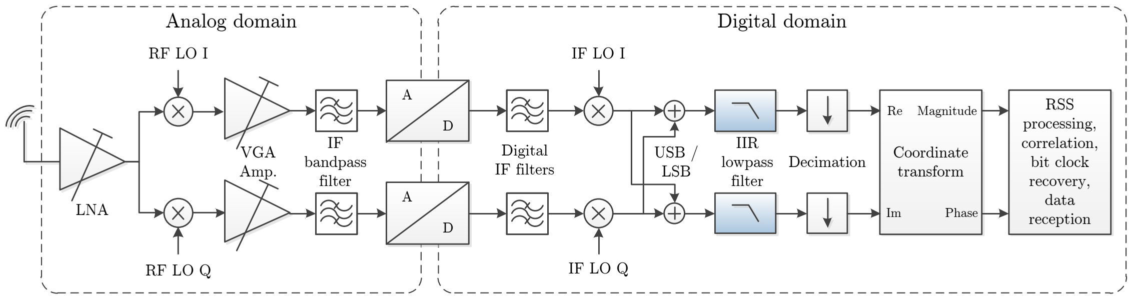


Figure 2.1: Abstract Receiver Model

2.2 Equivalent Baseband Model

In order to simulate and finally design an optimized filter for the present receiver, the whole system is modelled in a complex baseband equivalent representation. This simplifies the receiver model compared to a full passband model as any up- and downconversion to/from a carrier frequency is neglected. Therewith, simulation, and in a later context optimization, is much more convenient and has less computational effort. The passband and baseband equivalent representation is illustrated in Figure 2.2. The low- and highpass filters that are designed for the IF are used for equivalent filtering and band selection in this model. Therefore, they need to be transformed to baseband equivalents which is accomplished by shifting them by the IF carrier frequency f_{IF}.

The baseband equivalent model of the RF receiver is illustrated in Figure 2.3. Starting at the transmitter side, the desired binary data is upsampled and fed into the transmitting filter $H_T(z)$ which performs the pulse shaping. This reduces the infinite bandwidth of the δ-pulses for transmission. For simplification an additive white Gaussian noise (AWGN) channel is implemented. There are neither fading nor other distortions between transmitter and receiver considered as the receiving bandwidth is relatively small. The receiving filter $H_R(z)$ performs a band selection through a low- and highpass filter cascade. Finally the optimization filter $H_E(z)$ suppresses the introduced signal distortions before the signal is downsampled to symbol rate for detection again.

The following sections provide further details on the separate blocks of the system model from Figure 2.3.

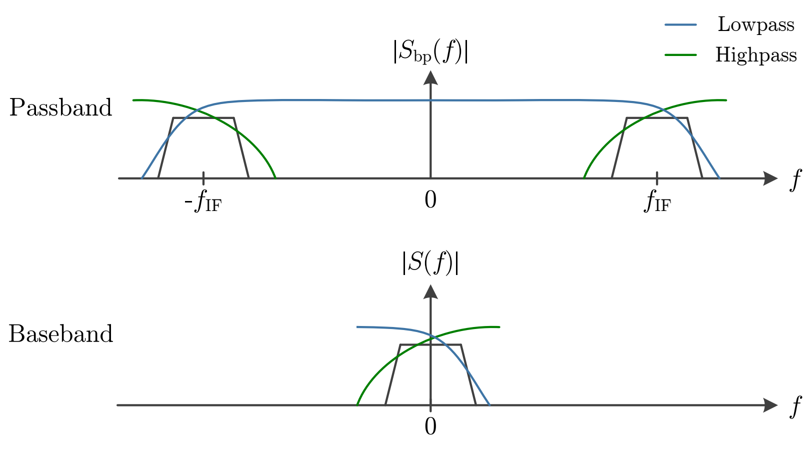


Figure 2.2: Passband and baseband spectra with filter masks

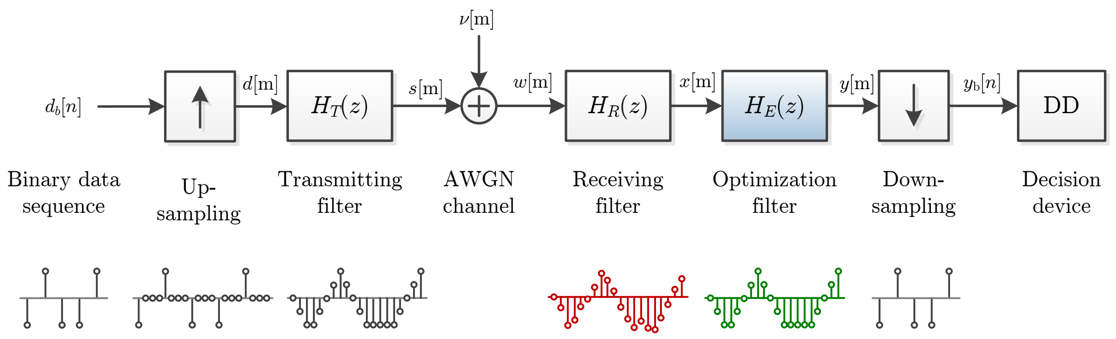


Figure 2.3: System model

2.3 Transmission Data Generation

The data for transmission is a bitstream that is provided by the microcontroller. Let's define the symbol sequence as

$$d_b[n] = \sum_{k=0}^{N_b} a_k \delta[n-k], \tag{2.1}$$

where N_b is the number of symbols to be transmitted, $a_k \in \{0,1\}$ are the symbols/bits and $\delta[n]$ is the Kronecker delta. Note that one symbol equals a single bit as we are dealing with a binary modulation method. After upsampling by a factor of M the infinite bandwidth of the signal is bandlimited by Gaussian pulse shaping. Therewith an appropriate tradeoff between the required bandwidth and the introduced inter-symbol interference (ISI) is selected. The actual transmitted sequence is obtained as

$$s[m] = d[m] * h_T[m], \tag{2.2}$$

where $*$ denotes the convolution operator and $h_T[m]$ is the Gaussian impulse response of the transmitting filter defined as [23]

$$h_T[m] = \frac{\sqrt{\pi}}{\alpha} e^{-\frac{\pi^2}{\alpha^2} m^2}, \tag{2.3}$$

where α defines the bandwidth B of the filter, with

$$\alpha = \frac{\sqrt{\ln 2}}{\sqrt{2}B}. \tag{2.4}$$

The corresponding transfer function is then written as

$$H_T(f) = e^{-\alpha^2 f^2}. \tag{2.5}$$

For the actual implementation, instead of using a filter, the pulse shaping is performed via a Gaussian lookup table (LUT) for computational simplification and hardware efficiency. The values for the LUT are obtained by (2.3). It provides a one-sided Gaussian pulse transition of length M. To apply the LUT, the binary input sequence $d_b[n]$ is oversampling by a factor M. The current data pointer within the LUT is updated according to the oversampled bitstream. If the sample is 1 the counter is increased and vice versa for 0 it is decreased. Whenever the pointer reaches the lower/upper bound of the LUT it is saturated to 0 or M respectively. This means, that if at least two equal symbols are transmitted successively, $s[m]$ constantly stays at either 0 or 1 inbetween the symbols. There is no drop as would occur with upsampled pulse shaping. The resulting transmission sequence $s[m]$ is presented in Figure 2.4 with $M = 16$. The corresponding power spectral density (PSD) is presented in Figure 2.5.

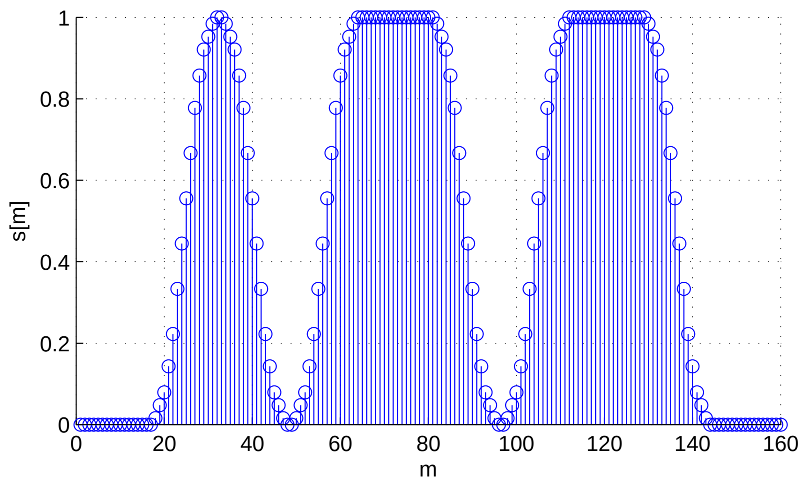


Figure 2.4: Transmission data pulse shaping

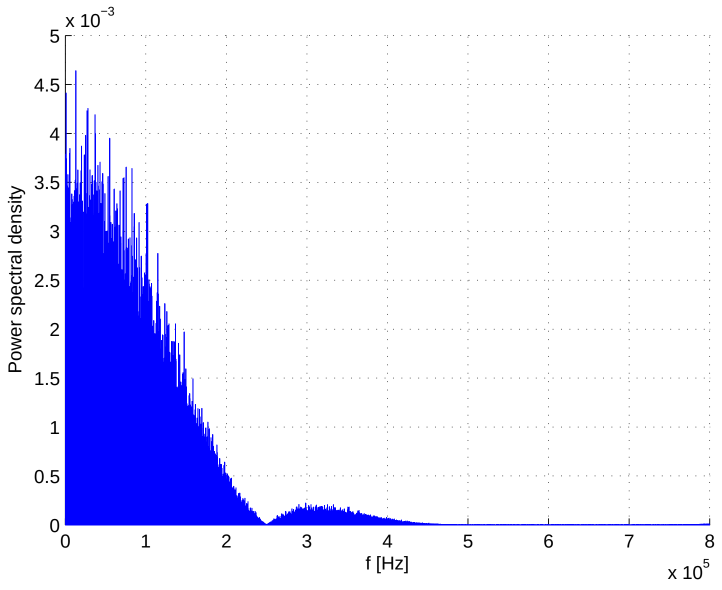


Figure 2.5: PSD of pulse shaped data samples

For performance evaluation, the average energy per bit of the sent data is calculated as

$$E_b = \frac{1}{N \cdot f_b} \sum_{m=1}^{N} s^2[m], \tag{2.6}$$

where $N = N_b \cdot M$ is the total number of samples of the data samples $d[m]$ and f_b is the data rate (bits per second). The nominal data rate is 125 kbit/s which is chosen for further analysis as well.

For ASK the baseband signal $s[m]$ is directly modulated onto the analog carrier wave, which is

$$c[m] = s[m] \cos(2\pi f_c / f_s \, m), \tag{2.7}$$

where f_c is the carrier frequency and f_s is the sampling frequency. Due to the transmitted symbols $a_k \in \{0, 1\}$ the modulation scheme is on-off-keying (OOK). In Figure 2.6 the discrete-time equivalent passband signal $c[m]$ is illustrated.

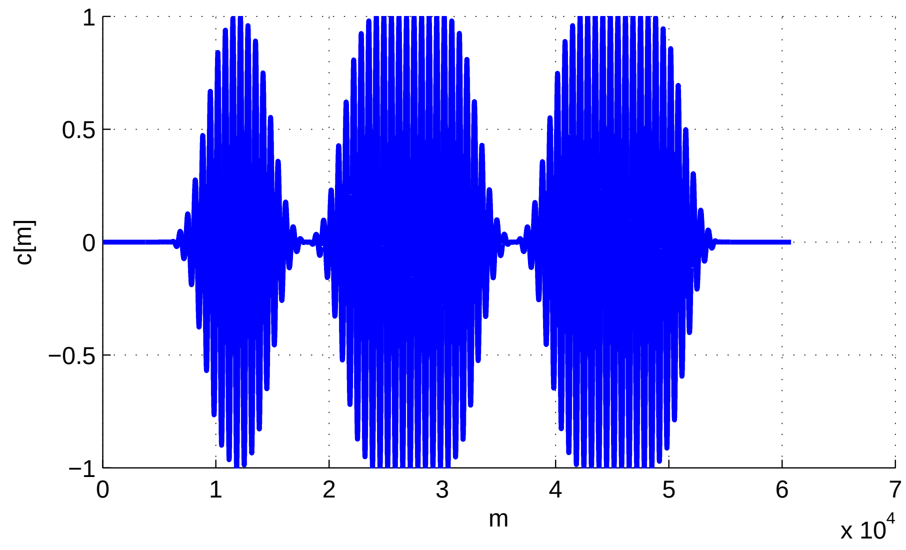


Figure 2.6: ASK modulation (OOK)

2.4 Channel

The wireless transmission channel is modelled as AWGN source. Issue at stake is the noise power that is added to the signal. White noise is defined by its autocorrelation function

$$r_{\nu\nu}(t) = \frac{N_0}{2}\,\delta(t), \tag{2.8}$$

where $N_0/2$ is the noise power and $\delta(t)$ is the Dirac delta function. The corresponding PSD is

$$P_{\nu\nu}(f) = \mathcal{F}\{r_{\nu\nu}(t)\} = \frac{N_0}{2}, \tag{2.9}$$

where $\mathcal{F}$ denotes the Fourier transform. Thus, $\nu(t)$ results in a constant power spectrum with infinite bandwidth. For mathematical simplification it is convenient to assume that the noise introduced and added at the antenna passed through an ideal bandpass filter [24]. The PSD of the noise $\nu(t)$ is presented in Figure 2.7 and defined as

$$P_{\nu\nu}(f) = \begin{cases} N_0/2 & \text{if } |f| \leq B/2 \\ 0 & \text{if } |f| > B/2 \end{cases} \tag{2.10}$$

where $N_0/2$ is the noise power density and B is the bandwidth of the ideal bandpass filter.

As the system model is discrete-time the noise is sampled, the noise power is obtained by multiplying the PSD (power per Hz) of the noise by the noise bandwidth, that is limited

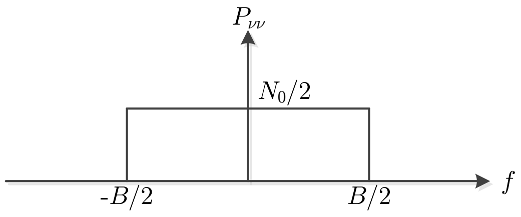


Figure 2.7: Ideal bandpass filtered white noise

by the sampling frequency f_s. Therewith, the noise variance

$$\sigma_\nu^2 = \frac{N_0}{2}\, f_s. \tag{2.11}$$

Thus, the additive noise randomly added to the signal is

$$\nu[n] = \sigma_\nu\, \gamma[n] \tag{2.12}$$

where $\gamma[n]$ is a stationary random process with normal distribution of zero mean and unit variance. Note that as we are dealing with a complex baseband model, also the noise added to the signal is complex in general. However, as the modulation scheme is OOK, the signal is modulated onto the real part only. Thus it suffices to consider only the real part of the noise.

2.5 Receiving Filter

The receiving filter models a cascade of digital filters for the IF, transformed to baseband equivalents. For nominal operation the IF is 250 kHz. The filters are located directly after the ADCs as depicted in Figure 2.1. The downconversion of the IF via mixers is performed fully in the digital domain. The overall digital IF processing is illustrated in Figure 2.8.

Again, for efficiency reasons the digital sinusoid waveform for the I/Q downconversion is generated out of a LUT.

2.5.1 CIC Decimation Filter

The cascaded-integrator-comb (CIC) decimation is very suitable for hardware implementations and a standard approach to reconstruct signals from $\Sigma\Delta$-ADCs [25]. The two objectives of the filter are given as follows:

1) Reconstruction of the bitstream from the $\Sigma\Delta$-ADC into a digitalized waveform

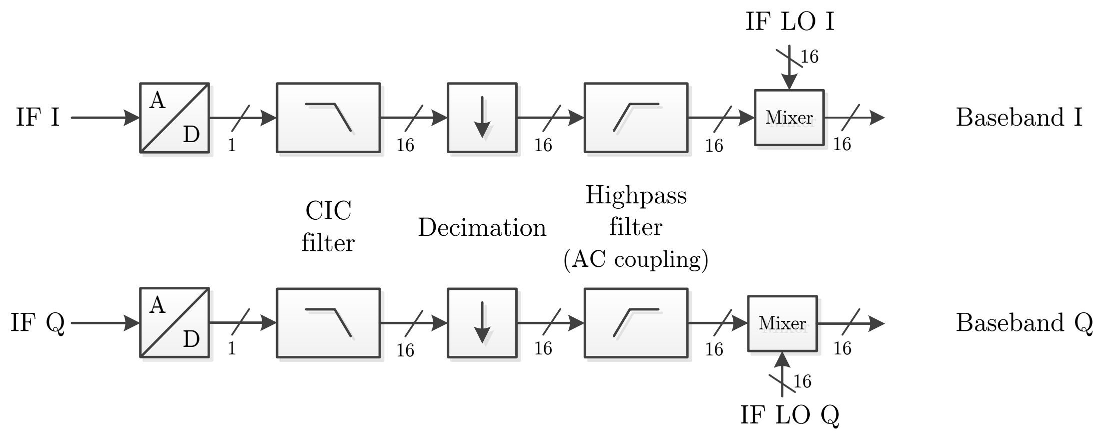


Figure 2.8: Low-IF digital filtering and mixing

(signed two-complement). Inherently this implies that the oversampling rate is reduced.

2) Suppress image of the signal at $2\,f_{\mathrm{LO}}$ introduced by the RF mixers.

The transfer function of the CIC filter is defined as

$$H_{\mathrm{CIC}}(z) = \left[\sum_{n=0}^{M-1} z^{-n}\right]^{N} , \tag{2.13}$$

where M is the decimation factor and N is the number of filter stages. The corresponding magnitude is a sinc shaped response as depicted in Figure 2.9. A very advantageous property of this filter is that it has a linear phase response and therewith a constant group delay.

Note that there is no compensation filter used to equalize the magnitude response in the passband of the filter as is generally implemented for CIC filters in combination with $\Sigma\Delta$-ADCs [25]. This can be seen as potential improvement with the cost of an additional filter. (!)

2.5.2 Highpass Filter (AC coupling)

The highpass filter is mainly used to remove DC components introduced by the $\Sigma\Delta$-ADC. Despite, it is used for the baseband equivalent model as kind of band selection filter together with the CIC decimation filter. It is shifted by the IF to form the complex baseband equivalent filter.

Finally, these two filters are cascaded to form the receiving filter $H_R(z)$. They fully

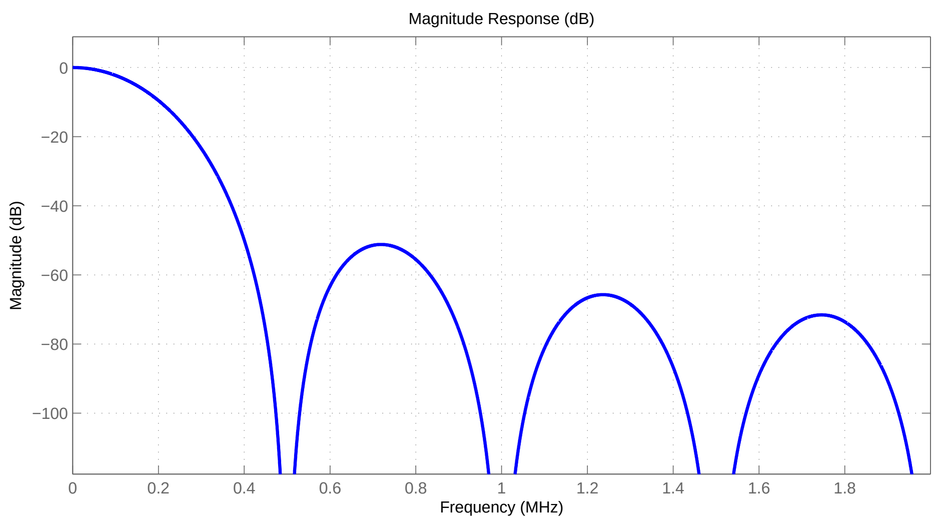


Figure 2.9: Magnitude response of the CIC decimation filter

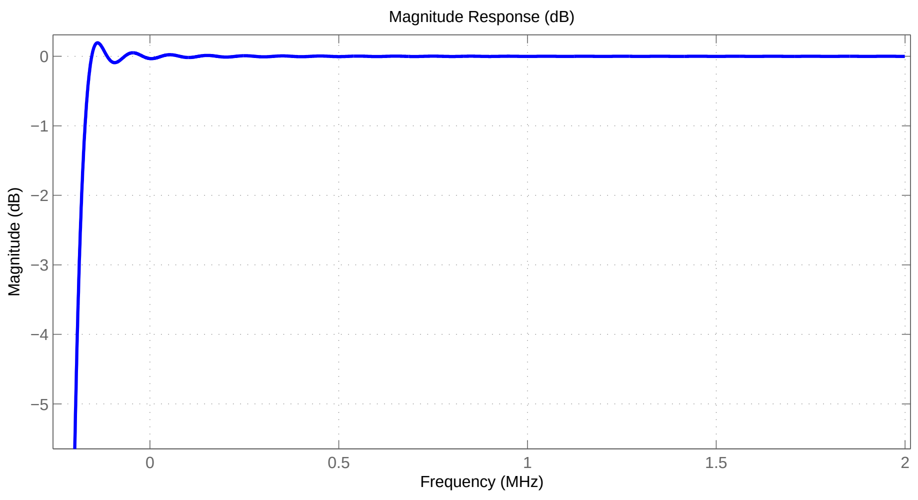


Figure 2.10: Magnitude response of the highpass filter

represent baseband equivalent filters as they are transferred from the IF to the complex baseband equivalent as shown in Figure 2.2.

2.6 Optimization Filter

The optimization IIR filter $H_E(z)$ is placed directly after the receiving filter $H_R(z)$ and the following downconversion to baseband. Low-IF receivers are advantageous compared to direct-downconversion receivers as DC offset issues, $1/f$ noise and second-order nonlinearities are circumvented [26]. The issue with low-IF receivers is though that through the downconversion an image of the signal occurs at the double IF. Thus, the main purpose of the existing IIR filter is image rejection. This is accomplished simply by lowpass filtering as shown in Figure 2.11. The second objective of this filter is to reduce the bandwidth as downsampling is performed right after the filter, but this is given by the image rejection inherently.

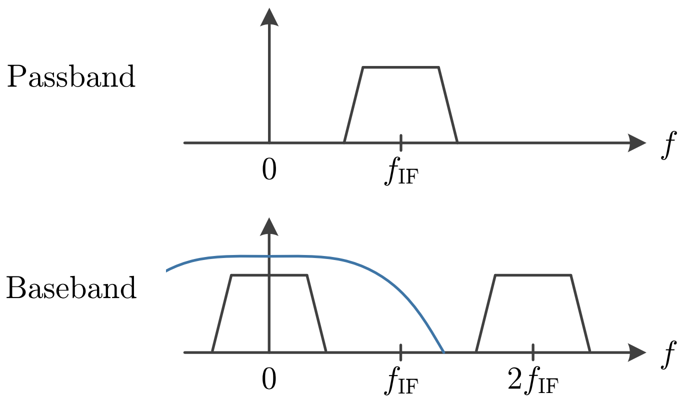


Figure 2.11: Image rejection

2.6.1 Deriving a Filter Specification

From the analysis done so far we can define a certain filter specification that has been used to design the existing filter. It will be reused for filter design in Chapter 3. The required bandwidth of the data is 250 kHz as has been shown in Figure 2.5. As this filter is the first in the baseband, the cutoff frequency of the lowpass filter is chosen to be 150 kHz to have a reasonable tradeoff between noise suppression and damping of the data signal. The stopband frequency is defined at 350 kHz to allow a smooth transition. As no distortions to the passband signal are desired, the magnitude response ripple should be as low as possible (around 0 dB). In order to keep the ISI low, special care should be taken of the group delay. In order to suppress the image as much as possible, a high stopband attenuation is desired at -80 dB. The sampling frequency is 4 MHz. The filter specification is depicted in Figure 2.12.

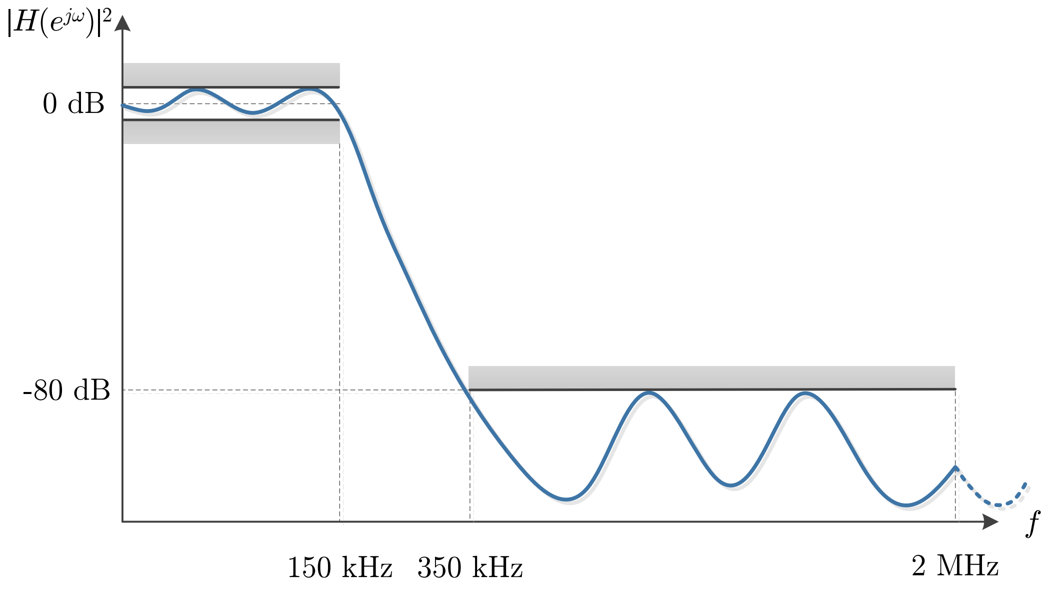


Figure 2.12: Filter specification for the existing filter design

2.6.2 Defining the Optimization Problem

Based on the blocks of the transceiver model described so far, the optimization problem to maximize the sensitivity of the receiver is defined. Under the constraints that the system model is fixed, an optimization filter is implemented that ensures optimum detection as deduced in Figure 2.13.

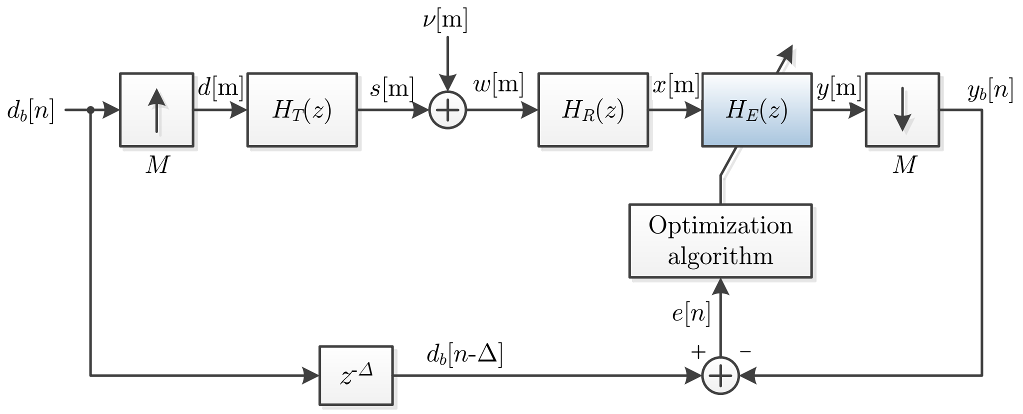


Figure 2.13: System optimization

2.7 Data Reception

A usual transmission packet for the RF transceiver consists of a preamble (PR), a packet synchronization (PS) word and the data field (DF) as depicted in Figure 2.14. The preamble consists of an alternating 1010 sequence, typically 8 or 16 bits long. It is mainly used for data slicing threshold generation. More important is the packet synchronization which correlates the known, predefined reference word.

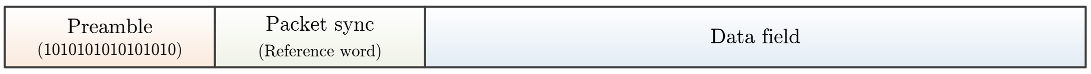


Figure 2.14: Typical transmission packet consisting of PR, PS and DF

If the samples of the correlator shift register match with the expected reference word, the correlation sum reaches its maximum. This crosscorrelation is defined as

$$c_{yp}[m] = \frac{1}{MN_b} \sum_{n=0}^{MN_b} y[n]p[n], \tag{2.14}$$

where M is the oversampling factor, N_b is the number of bits used for the PS, $y[m]$ are the received samples and $p[m]$ is the oversampled PS word. This correlation peak provides a basis for sampling. Thus, right after the occurrence of a PS, the bit clock generation is triggered that delivers sampling strobes for the rest of the DF as illustrated in Figure 2.15. It is therefore ensured, that the sampling of a bit value takes place in the middle of the oversampled incoming data bitstream.

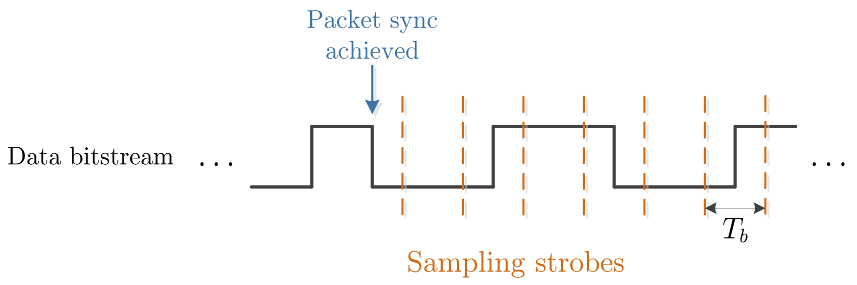


Figure 2.15: Sampling strobes generation after packet sync

2.8 MMSE Reference Filter

Standard linear equalizers that aim to flatten the frequency spectrum to remove ISI are the zero-forcing (ZF) and minimum-mean squared error (MMSE) equalizer. The ZF equalizer tries to compensate any distortions inbetween transmitter and receiver. However, it neglects the introduced channel noise. In contrast to that, the MMSE equalizer considers the overall receiver structure including the noise. This optimization procedure is visualized in Figure 2.13. The optimization algorithm can be arbitrary. For MMSE equalizers the least-mean squares (LMS) algorithm is often utilized to seek the filter parameters iteratively.

The MMSE solution will be used to compare the existing receiver performance to an optimum reference solution. Furthermore, it will also be compared to the solution found by the DE approach later on. Refer to Section A.1 for a detailed derivation of the MMSE solution for the receiver.

2.9 Performance Evaluation

In order to evaluate the receiver performance the bit error rate (BER) is computed for two different filter types. Firstly, there is a simple Butterworth lowpass filter which is used in the current version of the RF receiver. Secondly, there is the MMSE equalizer. As a reference, the theoretical BER as deduced in Section A.2 is plotted.

Figure 2.16 shows that the receiver performance is improved by the MMSE equalizer. At an E_b/N_0 ratio of 10^{-3}, which is defined as the sensitivity limit, the improvement is approximately 1.2 dB. The theoretical BER is not reached due to the encountered system design issue discussed in Section 2.5. The simulation is done for 10^6 bits.

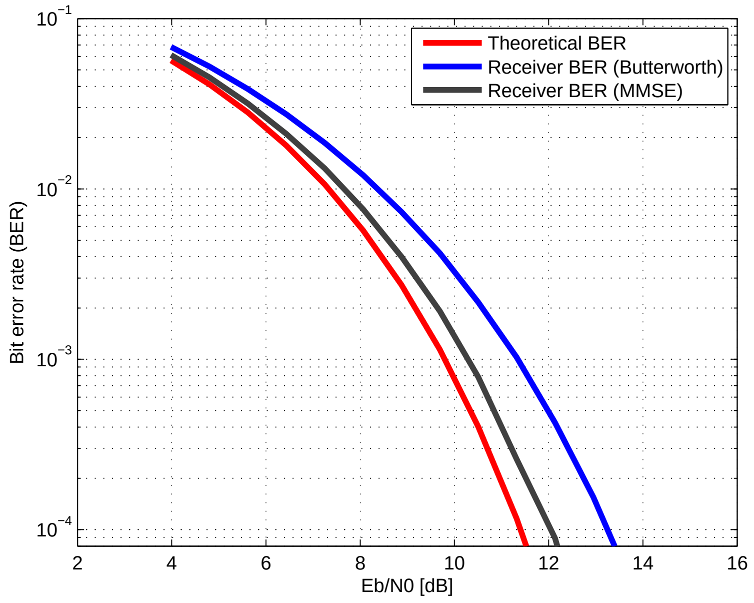


Figure 2.16: Comparison of existing Butterworth filter and MMSE equalizer

3 Quantized Filter Optimization

This section deals with the optimization of the filter with an arbitrary (lowpass) filter specification as well as for the dedicated RF receiver presented in Section 2. The filter specification may be defined by the system design and already provides certain constraints about the filter criteria such as passband and stopband behaviour of the frequency response. A DE algorithm will be employed to optimize the filter coefficients. The algorithm incorporates several partial cost functions that one might want to optimize.

3.1 Digital Filter Design

Most common traditional designs of IIR filters rely on analog-to-digital transformation based on filter specifications. Established filter design methods are Butterworth, Elliptic, Chebyshev type I and II filters. Based on the analog filter transfer function it is transformed into a digital filter using the bilinear transformation. MATLABs filter design and analysis tool (FDAtool) computes solutions for standard filters on this basis. This powerful tool even has the possibility to extract quantized coefficients (fixed-point) that are stable and normalized to unity gain, nevertheless it does not optimize the filter for a certain application [3, 10, 27].

Before starting with optimization of the filter itself, an introduction to IIR filter architectures is provided.

3.1.1 IIR Filter Architectures

In general an IIR filter can be represented by the difference equation

$$y[n] = \sum_{k=0}^{M} b_k \; x[n-k] - \sum_{k=1}^{N} a_k \; y[n-k], \tag{3.1}$$

where $x[n]$ and $y[n]$ are the in- and output of the filter respectively, M is the number of previous input and N the number of previous output samples taken into account for the filtering operation. The samples are weighted by b_k and a_k respectively. Equation (3.1) consists of a feedforward (FIR) and a feedback path. The latter is responsible for the infinite impulse response of the filter. The corresponding transfer function $H(z)$ is given as

$$H(z) = \frac{\sum_{k=0}^{M} b_k z^{-k}}{1 - \sum_{k=1}^{N} a_k z^{-k}}. \tag{3.2}$$

The filter architecture itself can be represented in different ways [10]. When writing the transfer function as in (3.2), it is said to be in direct form. By factorizing the polynomials by their poles d_k and zeros c_k, the transfer function $H(z)$ can be rewritten in cascade form, which is

$$H(z) = s \; \frac{\sum_{k=1}^{M} (1 - c_k z^{-1})}{\sum_{k=1}^{N} (1 - d_k z^{-1})} \tag{3.3}$$

There is full freedom of pairing the poles/zeros to subsystems in the form of (3.3). Thus, by combining pairs of real factors and complex conjugate pairs into second-order factors, the above equation can be rewritten as product (cascade) of sub-filter stages, so called second-order sections (SOS) or biquads. Therefore, the transfer function $H(z)$ can be given as

$$H(z) = \prod_{k=1}^{N_s} s_k \; \frac{b_{0k} + b_{1k} z^{-1} + b_{2k} z^{-2}}{1 - a_{1k} z^{-1} + a_{2k} z^{-2}}, \tag{3.4}$$

where N_s is the number of sections. Each SOS has a filter order of two. Additionally a scale value s_k is introduced for each biquad. Note that there is also full freedom in ordering the sections itself.

A graphical representation of (3.4) in direct-form II is given in Figure 3.1. This structure is rather advantageous for implementation in DSP designs [7]. Starting at the entry of the biquad, the scale value is a useful factor for limiting the numerical range of finite-precision arithmetic, thus overflows can be suppressed in an elegant way. Furthermore this scale value can be used to weight the SOS itself. As each SOS is a filter of order two, quantization errors are only influencing one section, the influence to the overall filter transfer function is reduced. Nevertheless the errors introduced at early stages will propagate through the following filter stages. This will be discussed in detail in Chapter 3.3. Finally, the number of delay elements is decreased (compared to direct-form I sections). It is important to note that the magnitude response is constrained by the filter order [10, 19].

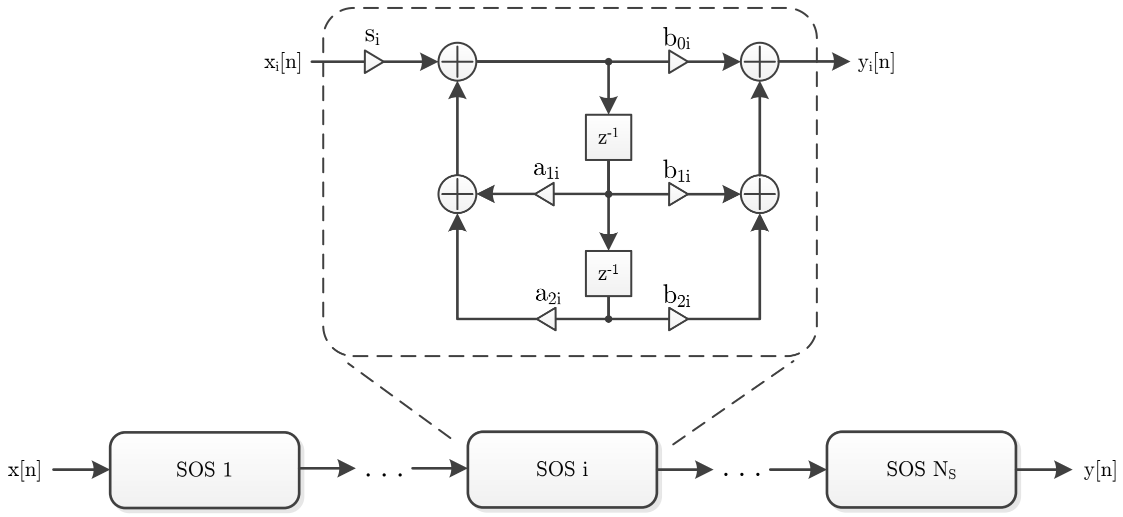


Figure 3.1: Cascade of second-order sections in direct-form II

3.1.2 Lowpass Filter Types

There are several lowpass filter types available based on the previously derived cascaded form [10, 27]. In this work the Butterworth and elliptic structure are mainly used. Within its SOS there are only two configurable coefficients, namely a_1 and a_2. The feedforward coefficients are fixed to $b_0 = 1$, $b_1 = 2$ and $b_2 = 1$. This filter type has a smooth magnitude response with low passband ripple and relatively low group delay ripple as depicted in Figure 3.2. However, sharp transition bands can't be reached using this filter architecture. In contrast to that the elliptic filter has three configurable parameters a_1, a_2 and b_1. This architecture allows for sharper transition bands with high stopband attenuation at the cost of passband ripple in the magnitude response and deterioration of the group delay as illustrated in Figure 3.3. Of course, Butterworth filters consume less area and power as one multiplier is saved since the coefficient multiplication by b_1 can be implemented as simple shift. This needs to be considered during the filter design.

By looking at corresponding pole/zero plot in Figure 3.4 it is observed that the poles of the elliptic filter get very close to the unit circle. Depending on the quantization step size the filter could become unstable as the pole is shifted outside the unit circle.

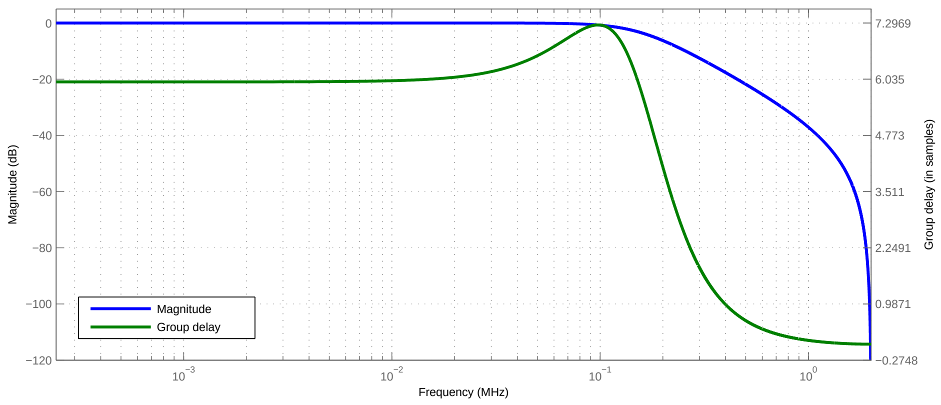


Figure 3.2: Magnitude and group delay of a fourth order butterworth filter with cutoff frequency 150 kHz and 4 MHz sampling frequency

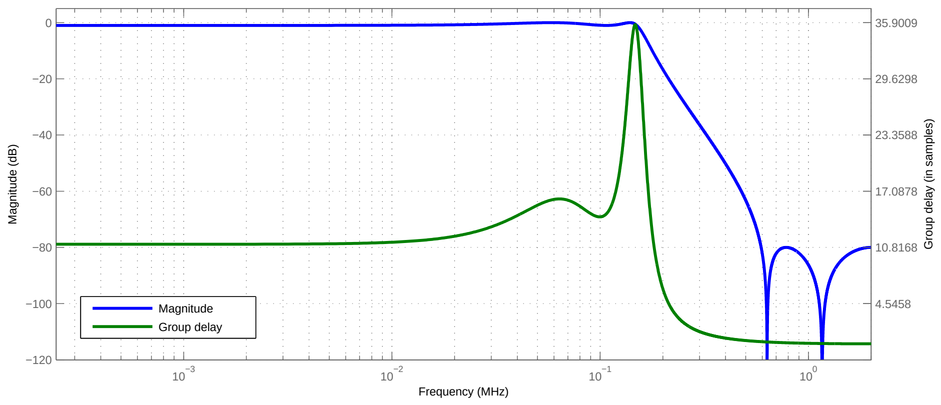
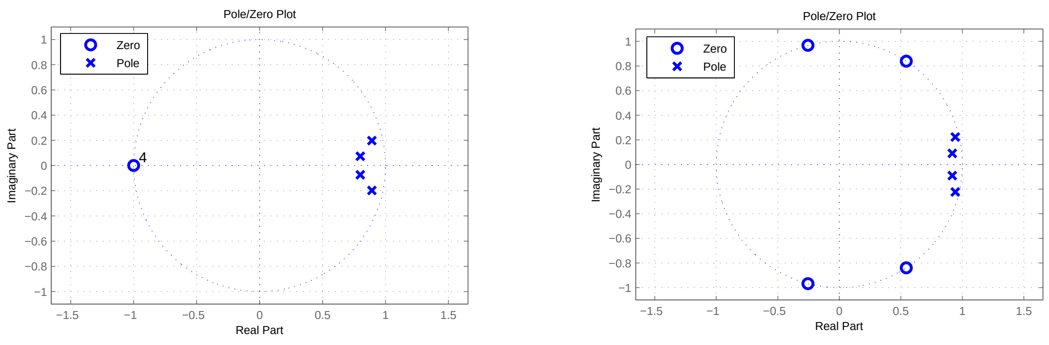


Figure 3.3: Magnitude and group delay of a fourth order elliptic filter with cutoff frequency 150 kHz and 4 MHz sampling frequency



(a) Butterworth (b) Elliptic

Figure 3.4: Poles/zeros of the Butterworth and elliptic filters from Figure 3.2 and 3.3 respectively

3.1.3 Effects of coefficient quantization

As quantization plays an important role in this chapter the effects are demonstrated here very shortly. Figure 3.6 shows the passband magnitude response of an elliptic filter with order 6. The coefficiants and scales are quantized to 10 bits. The outcoming filter response derivates significantly from the double-precision solution. Especially remarkable is, that the magnitude is no longer normalized to unity. This could cause overflows in systems where the arithmetic is assuming this constraint is hold. The group delay is less affected by the quantization process as presented in Figure 3.6.

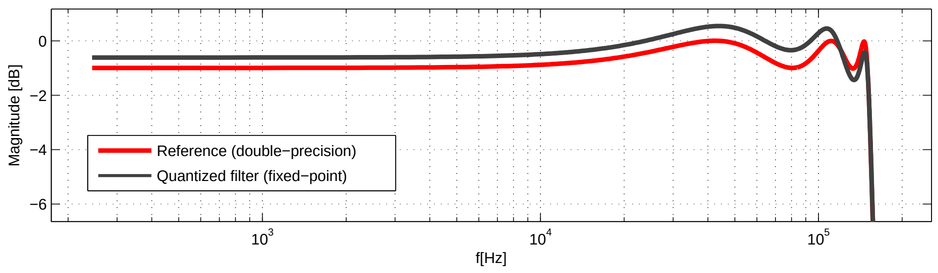


Figure 3.5: Effects of quantization on passband magnitude response of an elliptic lowpass filter

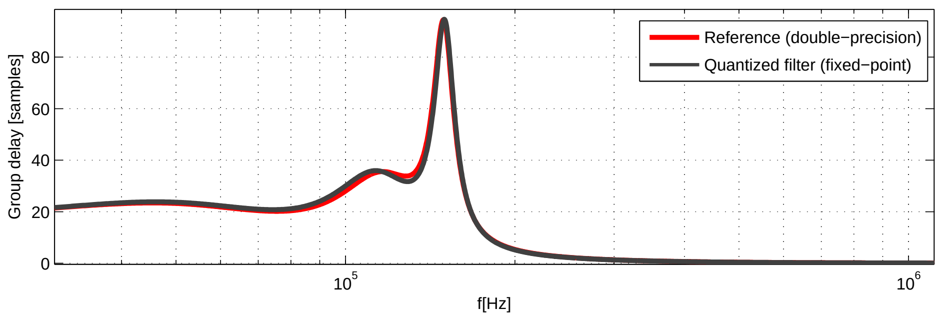


Figure 3.6: Effects of quantization on group delay of an elliptic lowpass filter

3.2 Heuristic Optimization Methods

As the effects of coefficient quantization and roundoff noise aren't negligible in general, optimization techniques need to be employed [3]. Quantization is a nonlinear process, thus optimization of a quantized filter is not a trivial task. A general optimization approach would always be some sort of gradient search which would need a cost[3] function that is fully

[3] Note that for heuristics the cost is often referred to as fitness although the meaning is completely the same.

differentiable. In fact this no longer holds when coefficient quantization is incorporated into the minimization [28]. Whenever exhaustive/random search is impractical, heuristic methods are applied in order to improve the speed of convergence to find an optimum, or at least a satisfactory solution.

This section provides a short introduction and overview about evolutionary meta-heuristic optimization methods and examples for existing implementations are given based on [29], [30]. Additional investigations are provided for multiobjective optimization [31].

Evolutionary heuristics try to improve candidate solutions in an iterative way. A certain cost function needs to be defined that creates a hyperplane in which a minima or maxima shall be found. The target is to find a global optimum, thus the aim is to cover the whole search space. Metaheuristics are generally applied to find solutions for problems where the optimum might not be obvious to find and the path to arrive there is not straightforward. An structured overview of algorithms that will be discussed shortly in this section is given in Figure 3.7.

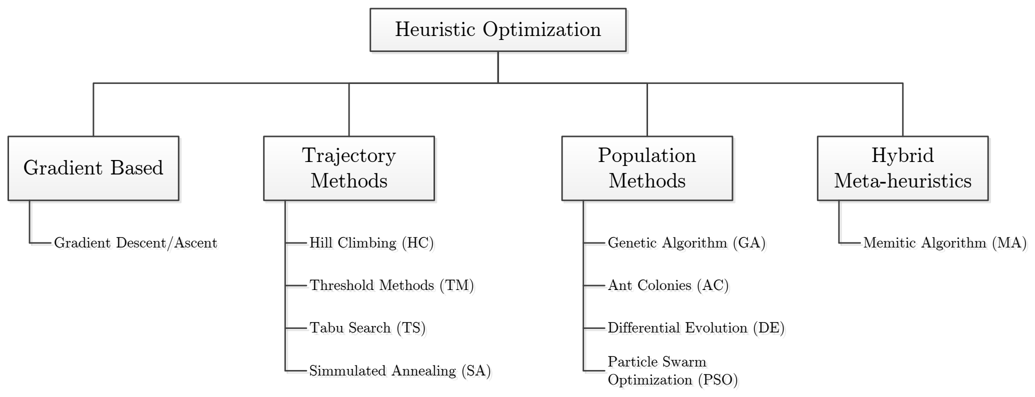


Figure 3.7: Overview of Heuristic Optimization Methods

3.2.1 Evolutionary Heuristics

Gradient Based

A traditional mathematical method to find optima of a function is the gradient ascent/descent method. It does so by computing the derivative of the function and therefore moves up/down the hill iteratively. The update is done with a configurable step size α that needs to be chosen appropriate. If the slope becomes zero, the algorithm has found at least a local optimum. The initial starting point is chosen randomly. For a multidimensional function $f(\boldsymbol{x}), \boldsymbol{x} = x_1, ..., x_N$, where N represents the dimensionality, the gradient $\nabla f(\boldsymbol{x})$ is computed to obtain the update vector. A pseudocode of the gradient descent algorithm is provided in Listing 3.1.

```
$\boldsymbol{x}$ ← random vector

loop
  $\boldsymbol{x}$ ← $\boldsymbol{x}$ + $\alpha$ $\nabla f(\boldsymbol{x})$
until Optimum solution found or max. iterations reached

return $\boldsymbol{x}$
```

Listing 3.1: Gradient descent algorithm

The algorithm is constraint as it can only deal with continuous and differentiable functions. Also, the step size α needs to be readjusted for different optimization problems. In order to overcome this issue Newton's method can be employed. It additionally computes the second derivative of $f(\boldsymbol{x})$, thus it converges faster. However, the major problem with these algorithms is that they get stuck in local minima or maxima.

Trajectory Methods

The constraint that the cost function needs to be differentiable for gradient based optimization is a major drawback. For many application such a function can't be provided. It might even be, that the function to be optimized is completely unknown. Trajectory methods try to test such systems with candidate solution generated randomly around the current solution and evaluate the output. An abstract pseudocode is provided in Listing 3.2.

```
$\boldsymbol{a}$ ← random candidate

loop
  $\boldsymbol{b}$ ← Manipulate($\boldsymbol{a}$) -- Generate new candidate solution based on $\boldsymbol{a}$

  if Fitness($\boldsymbol{b}$) > Fitness($\boldsymbol{a}$)
    $\boldsymbol{a}$ ← $\boldsymbol{b}$
until Optimum solution found or max. iterations reached

return $\boldsymbol{a}$
```

Listing 3.2: Trajectory optimization algorithm

- **Hill Climbing** The hill climbing technique is related to gradient based methods, however neither the direction nor the slope of the gradient itself is computed. Instead, this algorithm randomly creates individuals spread out into random directions. Iteratively, a local hill or valley is found. As the gradient method, this algorithm gets stuck at local optima, thus several starting points are needed to find different minima/maxima of the cost function.

- **Tabu Search (TS)** Tabu search has the prerequisite that the search space is finite and therefore discrete. Visited solutions are stored in a memory (*tabu-list*), thus *tabu*-solution will never be re-visited. This exploits the search space, however the strong prerequisite limits the usage to certain applications.
- **Simulated Annealing (SA)** This method is considered as a refinement of a local search which is based on the annealing process of solids for solving combinatorial problems. In contrast to standard hill climbing it accepts moves into the *wrong* direction with given probability. This probability depends on the current *temperature* T that decreases exponentially, thus the innovation is large at the beginning, but small in the end to find a refined optimal solution.

Population based methods

This set of algorithms improve a certain population of individuals at the same time. That is what makes them advantageous in terms of exploitation of the whole search space. Of course, this increases the computational costs of the overall optimization process. This class of algorithm is built up in the same way, a basic framework is provided in Listing 3.3.

```
popsize ← population size
$P$ ← random initial population

best ← none
loop
  for 1 to popsize do
    for each $P_i$ in $P$
      EvaluateFitness($P_i$)
      if best = none or Fitness($P_i$) > Fitness(best)
        best ← $P_i$
      end
    end

    $Q$ ← []
    for 1 to popsize
      -- Select parents randomly
      [$P_1$, $P_2$] ← Selection($P$)
      -- Create child out of it's parents
      $C$ ← Crossover($P_1$, $P_2$)
      -- Mutate the child
      $Q_i$ ← Mutation($C$)
    end
    $P$ ← $Q$
  end
until Optimum solution found or max. iterations reached

return best
```

Listing 3.3: Population based optimization

- **Genetic Algorithm (GA; Holland, 1975)**
  The GA imitates the sexual reproduction during evolution from a population and

is considered as the prototype for population based heuristics. New individuals are evolving using *crossover* where random properties of the parents are used to create the childs (*mutation*). These properties are also called *individuals* or *chromosomes*. Depending on the random selection of these properties the child might have an improved performance, thus higher probability to survive within the population. Only the fittest individuals are kept inside the population after each iteration, they will become parents for the next one. Obviously, the average fitness is improved step by step while at the same time the exploration is high due to the random *mutation*. Generated childs that do not improve or provide any new features are neglected anyway. The variance of new *mutated chromosomes* is large in the beginning if the population is spread out over the whole search space (3.8a). Once the individuals settle around potential solutions the variance will decrease and the mutation will produce refined individuals within this area (3.8b). Nevertheless it is claimed that the GA lacks in local search ability [2]. At this point it is important to note that the population size needs to be chosen with care. Carrying a high number of individuals results in high exploration of the search space, but convergence speed will decrease as the exploitation happens after the individuals accumulate around optimum solutions. The algorithm either runs for a certain number of iterations or stops after an individual with a fitness less than a certain threshold has been found.

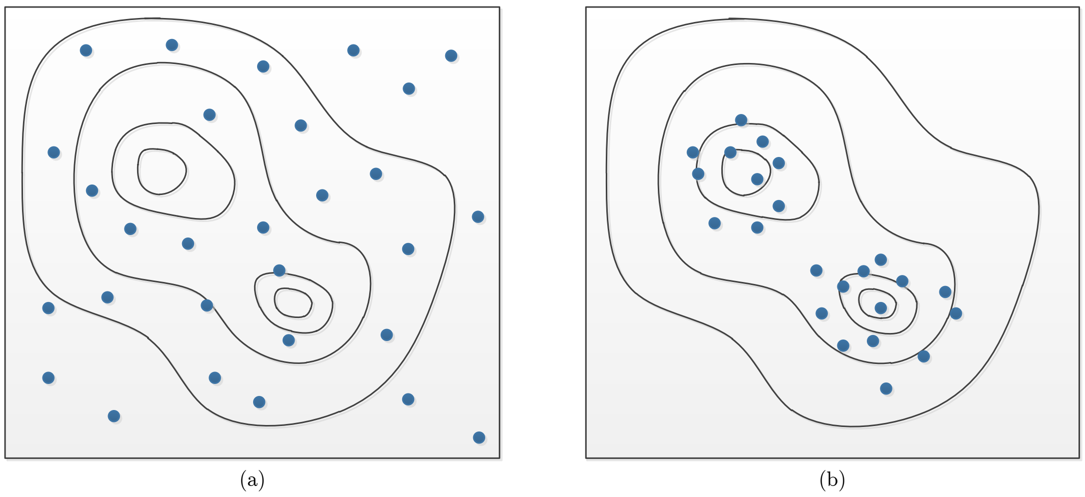

(a) (b)

Figure 3.8: Evolvement of Genetic Algorithm over generations

- **Ant Colony Optimization (ACO; Colorni et al., 1992)**
  This method imitates the way ants explore their environment to find food. Initially the ants do a random search. Once a food source has been found, the ant leaves a trace of pheromones (*fitness*) on it's way back to the nest that delivers a potential path for the other ants. The intensity of pheromones provides a weight for the food quality, quantity and the travel distance from the nest. Once the food sources are exhausted, the process starts all over again.

For implementation the search area is divided into a discrete set, an objective function represents the amount/quality of food and the pheromone path is modelled via an adaptive memory. The objective function is evaluated iteratively and the best path and its weights updated accordingly.

- **Differential Evolution (DE; Storn and Price, 1997)**
  The DE evolution is closely related to the GA, the main difference is that the update (*mutation*) is done based on a differential vector addition. The basic principle behind this vector addition/subtraction is rather simple and illustrated in Figure 3.9. In order to update individual A, two other randomly chosen individuals B and C are incorporated to compute the update. The vector difference between B and C is computed and added to A. This provides the *mutated* child A'.

  It has been shown that this algorithm delivers better results with faster convergence compared to the GA especially for filter design [2, 21].

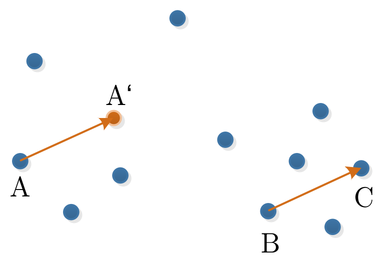


Figure 3.9: Vector addition/subtraction in differential evolution

- **Particle Swarm Optimization (PSO; Eberhart and Kennedy, 1995)**
  In PSO the population consists of so called *particles* which are iteratively updated with a *velocity*. The vector along which the update is performed is computed out of a vector between the positions of the current particle and

  1) the best particle found within the current population respectively

  2) the best particle found so far.

  These two vectors are then randomly blended to define the direction in which the update with the velocity estimate is performed.

Hybrid meta-heuristics

A hybrid meta-heuristic is defined by assembling two different heuristic approaches in order to gain further performance and/or precision. Hence a better tradeoff between exploration of the overall search space and exploitation of the solution at local minima can be found. For instance, a population based method might explore the search space well while a trajectory method climbs up the hill pretty fast in the local surrounding. For filter design this technique has been successfully applied [14]. As one can imagine there are numerous possible combinations of methods that need to be fully customized for a dedicated optimization problem.

3.2.2 Multiobjective Optimization

In order to fully use the power of heuristic algorithms, multiple objective cost functions can be defined. The optimization problem is then reformulated to minimizing/maximizing the fitness functions $F_n(\mathbf{x})$ for $n = 1, 2, ..., N$, where N is the number of objectives and $\mathbf{x}$ is a vector containing the parameter variables needed for computation of the fitness function. Multiple objectives can be applied to any of the above described heuristic optimization methods, it's just about computing the total fitness of an individual. For instance, this can be performed by a naive weighting approach, i.e.

$$F_{\text{total}}(\mathbf{x}) = \sum_{n=1}^{N} \alpha_n F_n(\mathbf{x}) \tag{3.5}$$

with α_n being a normalized weighting factor for the partial cost functions F_n.

Another approach is to compute the *Pareto dominance and Pareto front.* Individual A Pareto dominates individual B if

$$F_{\text{An}}(\mathbf{x}) \geq F_{\text{Bn}}(\mathbf{x}) \quad \forall\ n = 1, 2, ..., N \tag{3.6}$$

where N is the number of objectives. $F_{\text{An}}(\mathbf{x})$ and $F_{\text{Bn}}(\mathbf{x})$ are the fitness functions of the n-th objective of individual A and B respectively given the parameter set $\mathbf{x}$. However, for the whole population there will be individuals that do not Pareto dominante others but are still of interest as they are better in some objectives. Some of these will even be the best for a single or very few objectives. The hyperplane out of these chromosomes defines the Pareto front. By non-dominated sorting, the fitness space is classified into different Pareto ranks and the selection done based on the rank of the individuals. Although this is an interesting measure there is an additional computational effort to obtain the Pareto ranks. It grows exponentially with the number of objectives N.

3.2.3 Design Decisions

For the optimization tool implemented in this work a multiobjective DE approach is chosen. Among other heuristics it has proven to have very high performance for filter design [2, 21]. In general, the design is independent on the filter structure, however in this work an SOS approach is used due to advantages described in Section 3.1. One chromosome represents a whole cascade of SOS. There are six parameters per SOS to tweak. The cascaded structure is presented in Figure 3.1, where N_s is the number of sections (overall filter order $2N_s$). The corresponding chromosome that is used in the DE can be represented via an SOS matrix **c** given as

$$\boldsymbol{c} = \begin{bmatrix} b_{01} & b_{11} & b_{21} & a_{01} & a_{11} & a_{21} \\ b_{02} & b_{12} & b_{22} & a_{02} & a_{12} & a_{22} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{0N_s} & b_{1N_s} & b_{2N_s} & a_{0N_s} & a_{1N_s} & a_{2N_s} \end{bmatrix} \tag{3.7}$$

and a scale vector **s** defined as

$$\boldsymbol{s} = [s_1, s_2, ..., s_{N_s}]^{\mathrm{T}}. \tag{3.8}$$

3.3 Statistical Filter Quantization Model

In this section a statistical model for representing round-off noise due to quantization in digital filters is derived. Starting with a simple one pole filter the quantization model for a cascade of second-order sections is deduced. Main concepts are taken from [10, 32].

3.3.1 Representation of Quantization Noise

As an introduction it is shown that quantization noise can be represented by substituting a quantizer by the addition of a single noise source. The concept is the addition of pseudo quantization noise (PQN) as proposed in [33].

Lets define $\nu[n]$ as a random variable (RV), uniformly distributed between $\pm q/2$, with q representing the smallest quantization interval of a quantizer Q. The noise RV has zero mean and variance $\sigma_n^2 = q^2/12$, the corresponding probability density function (PDF) is $f_\nu(\xi)$ as shown in Figure 3.10(a). The signal $x[n]$ is fed through a quantizer Q as depicted in Figure 3.10(b), the quantizer output is $y_Q[n]$. Now, utilizing the PQN model [33] the random noise variable $\nu[n]$ is added to $x[n]$ and hence substituted by the nonlinear quantization process as depicted in Figure 3.10(c). Assuming that $x[n]$ and $\nu[n]$ are statistically independent, the addition

$$y_\nu[n] = x[n] + \nu[n] \tag{3.9}$$

results in a convolution of the PDFs, given as

$$f_{y_\nu}(\xi) = f_x(\xi) * f_\nu(\xi). \tag{3.10}$$

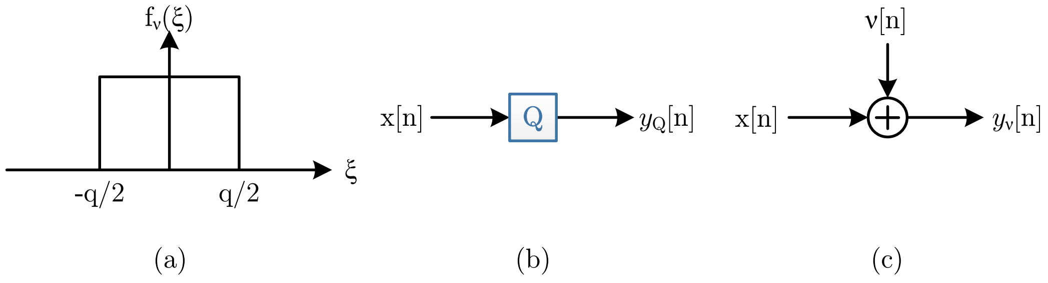


Figure 3.10: PDF of independent quantization noise (a), signal passing through quantizer (b) and quantization represented as independent additive noise (c)

Note that there is a fundamental difference of the output $y_Q[n]$ and $y_\nu[n]$ from Figure 3.10(b) and 3.10(c) respectively as the quantization and addition of the RV is a completely different operation. $f_{y_Q}(\xi)$ consists of discrete dirac impulses, uniformly spaced with q, whereas $f_{y_n}(\xi)$ is a continuous function in general. This is illustrated in Figure 3.11.

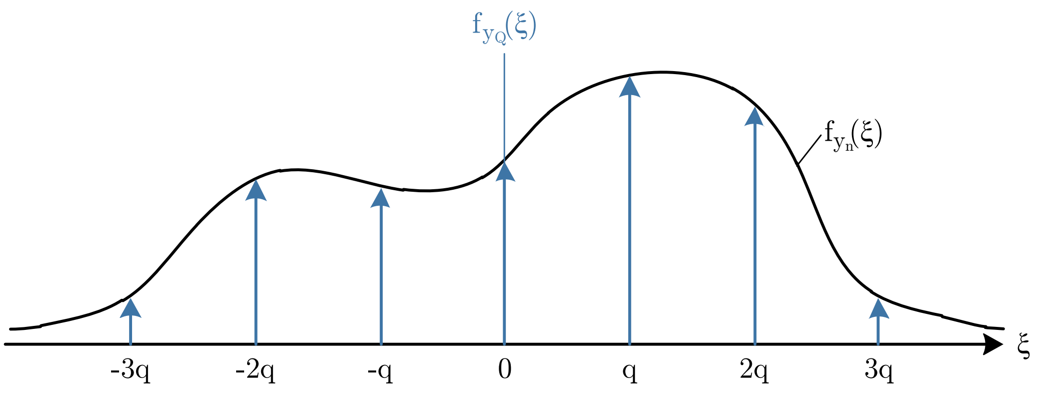


Figure 3.11: PDF of the quantizer output $y_Q[n]$ and $y_\nu[n]$

To constitute the actual difference between the models an impulse train $c(\xi)$ is defined which models the uniformly spaced Dirac impulses used for sampling the PDF. Then,

$$f_{y_Q} = c(\xi) \underbrace{[f_x(\xi) * f_\nu(\xi)]}_{f_{y_\nu}(\xi)}. \tag{3.11}$$

For further analysis the characteristic function is utilized. It is defined by

$$\Phi_x(\mu) = \int_{-\infty}^{\infty} f_x(\xi)\ e^{j\xi\mu}\ d\xi \tag{3.12}$$

which is the Fourier transform of a PDF, where μ can be considered as formal frequency parameter. Applying a Fourier transform to (3.11) therefore results in

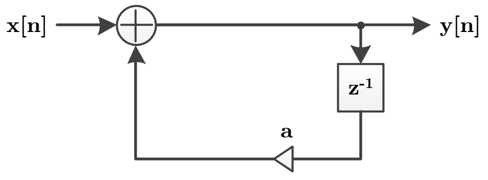
$$\Phi_{y_Q} = C(\mu) * [\Phi_x(\mu) \cdot \Phi_n(\mu)]\,. \tag{3.13}$$

This shows that there is a direct correspondence between sampling and quantization using this model. As the PDF is sampled, there are bandlimitation conditions of the input signal x that need to be satisfied such that the PQN model can be applied for analysis of quantization noise. This is covered by the quantization theorem QT I [33]. As the convolution in (3.13) results in a periodic repetition every $2\pi/q$ it is given as

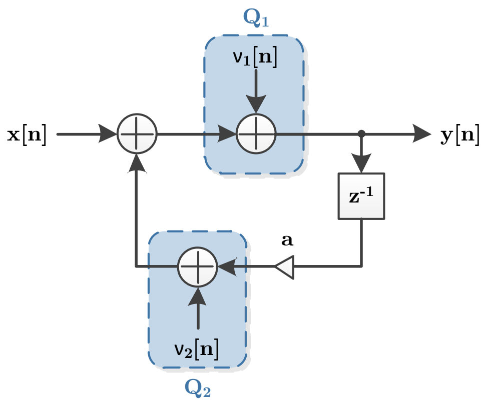
$$\Phi_x(\mu) = 0, \quad |\mu| > \frac{\pi}{q}. \tag{3.14}$$

3.3.2 Simple One-pole Filter

The simple one-pole filter that is used for an introductory example for fixed-point quantization is shown in Figure 3.12(a). The input and output of the system are $x[n]$ and $y[n]$ respectively. There is simply one multiplication and one addition performed. The quantization noise introduced by these two operations is represented by Q_1 and Q_2 (Figure 3.12(b)). Both are implemented as PQN noise sources $\nu_1[n]$ and $\nu_2[n]$.



(a) One-pole filter



(b) One-pole filter (PQN model)

Figure 3.12: One-pole filter structures

Time-domain analysis

The transfer function of the system in Figure 3.12(a) is

$$H(z) = \frac{1}{1 - a\ z^{-1}}. \tag{3.15}$$

As the signal, and hence also the introduced quantization error passes recursively through the filter, the impulse response of the filter is of major interest. For this system this is a simple geometric series $h[n] = a^{n-1}$. We wish to compute the overall round-off noise power introduced by the system. It is given by the sum of squares of the impulse response, which is

$$\sum_{n=1}^{\infty} |h[n]|^2 = \sum_{n=1}^{\infty} a^{2(n-1)} = \frac{1}{1-a^2} \qquad \forall\ |a| < 1. \tag{3.16}$$

To get the overall contribution of Q_1 to the system transfer function we can simply multiply the sum of squares of the impulse response with the mean square quantization error to obtain the noise variance given as

$$\sigma_{\nu_1}^2 = \frac{q^2}{12} \cdot \frac{1}{1-a^2}. \tag{3.17}$$

As long as the quantization noise is assumed to be uncorrelated, the quantization noise introduced by Q_2 is

$$\sigma_{\nu_2}^2 = \frac{q^2}{12} \cdot \frac{1}{1-a^2}. \tag{3.18}$$

The total noise power introduced by the quantizers sums up to

$$\sigma_{\nu}^2 = \sigma_{\nu_1}^2 + \sigma_{\nu_2}^2 = 2\ \frac{q^2}{12\ (1-a^2)}. \tag{3.19}$$

Frequency-domain analysis

For a filter design problem, the impact of quantization on the transfer function is issue at stake. Based on the previous calculations it is straightforward to compute the noise power introduced. The quantization noise variance is attenuated by the squared magnitude response of the filter, hence written as

$$P_{\nu\nu}(\omega) = 2\ \frac{q^2}{12}\ |H(e^{j\omega})|^2. \tag{3.20}$$

Stability

The one-pole filter is stable as long as $|a| < 1$. In order to guarantee that the filter is stable even on a finite arithmetic unit, the pole that the coefficient a creates needs to stay within the unit circle even after the coefficient quantization process. As a second stability criterion the numerical range of the state variable needs to be considered. The integer portion of the number representation must be chosen large enough such that all possible multiplication results stay within the lower/upper bound.

3.3.3 Second-Order Section

The theory derived in the previous section will now be extended to a full filter section. For one SOS there are six multiplications performed. The corresponding direct-form II model including quantizers is presented in Figure 3.13. Each multiplication results in quantization noise at the output. For the adders there are no quantization errors introduced as the inputs to each of them are already quantized. Of course there is also the possibility that the summation of the feedforward or feedback path causes an overflow, however this is assumed to be covered by a suitable scale value. In Figure 3.13 the fixed-point number representation is indicated for different Qm.n number formats. This notation represents the number of integer (m) and fractional (n) portions of the number representation. After each multiplication the bitwidth of the number doubles and therewith the double amount of integer and fractional portions, i.e. Q2m.2n, is required without further knowledge.

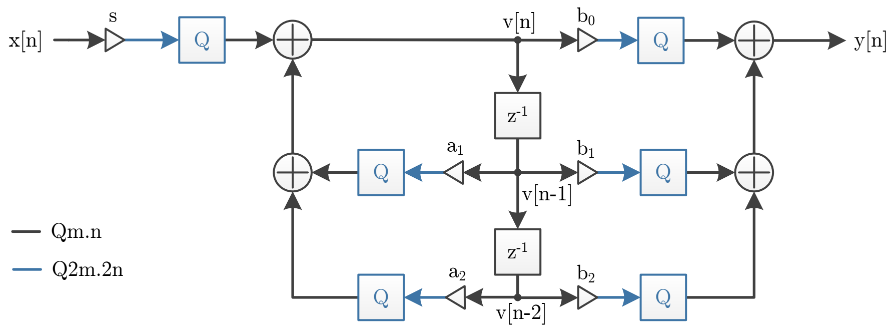


Figure 3.13: Second-order section with quantizers and corresponding number representations

In order to compute the noise variance the scale factor, the coefficients of the feedback path and the coefficients of the feedforward path are considered separately. The noise introduced by the feedforward coefficients is directly added to the output signal without attenuation. The quantization noise from the feedback path is multiplied with its impulse response as shown exemplary for the one-pole filter. Analogous, the quantization error from the scale multiplication is multiplied with the overall impulse response of the SOS. The resulting noise variance is

$$\sigma_\nu^2 = \underbrace{\frac{q^2}{12}\sum_{n=0}^{\infty}|h[n]|^2}_{\text{from scale factor}} + \underbrace{2\cdot\frac{q^2}{12}\sum_{n=0}^{\infty}|h[n]|^2}_{\text{from feedback path}} + \underbrace{3\cdot\frac{q^2}{12}}_{\text{from feedforward path}}, \tag{3.21}$$

where $h[n]$ is the impulse response of the overall SOS. Note that σ_ν^2 replaces all the separate noise introduced by the quantizers by a single addition after the SOS.

The corresponding noise PSD can be similarly derived as

$$P_{\nu\nu}(\omega) = \underbrace{\frac{q^2}{12}|H(e^{j\omega})|^2}_{\text{from scale factor}} + \underbrace{2\cdot\frac{q^2}{12}|H(e^{j\omega})|^2}_{\text{from feedback path}} + \underbrace{3\cdot\frac{q^2}{12}}_{\text{from feedforward path}}, \tag{3.22}$$

where $|H(e^{j\omega})|$ is the magnitude response of the overall SOS.

Floating-point

As defined in [32], the noise power of one floating-point quantizer is given as

$$E\{\sigma_\nu^2\} = 0.180\cdot 2^{-2p}E\{y^2[n]\}, \tag{3.23}$$

where p is the number of bits used for the mantissa. There is some additional computational effort needed to compute the expectation value $E\{y^2[n]\}$. A certain input sequence needs to be defined before the variance $\sigma_\nu[n]$ might be computed as the number range matters for floating-point computations. Besides that the noise power is propagated through the filter in the same manner, thus also (3.21) and (3.22) hold for the floating-point case.

3.3.4 Cascade of Second-Order Sections

The model from the previous chapter can be developed even further by cascading the second-order sections to obtain a quantization noise model for the overall filter structure. The overall noise power for the cascaded SOS can be calculated out of the single noise PSDs as

$$P_{\nu\nu}(\omega) = \sum_{n=1}^{N_s} P_{\nu\nu,n}(\omega) \prod_{i=n+1}^{N_s} s_i^2|H_i(e^{j\omega})|^2, \tag{3.24}$$

where N_s is the number of sections and $P_{\nu\nu,n}(\omega)$ are the noise PSDs for $n = 1, ..., N_s$. Important to note is that the quantization noise introduced at prior sections is suppressed by the scale values of the following sections. This is shown by computing the noise PSD for each single SOS, examplary done for a sixth order elliptic filter with a bitwidth of 10 for the internal state variables. Figure 3.14, 3.15 and 3.16 show the magnitude response of the SOS and the corresponding feedback path as well as the round-off noise PSD of each section. Figure 3.17 presents the resulting overall magnitude response and the round-off noise introduced. Note that the quantization noise is almost fully dominated by the last section.

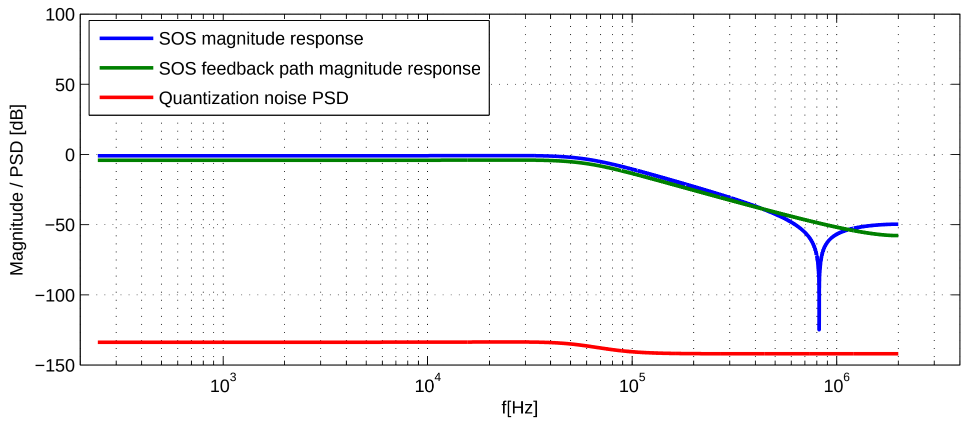


Figure 3.14: Magnitude responses and noise PSD of SOS1

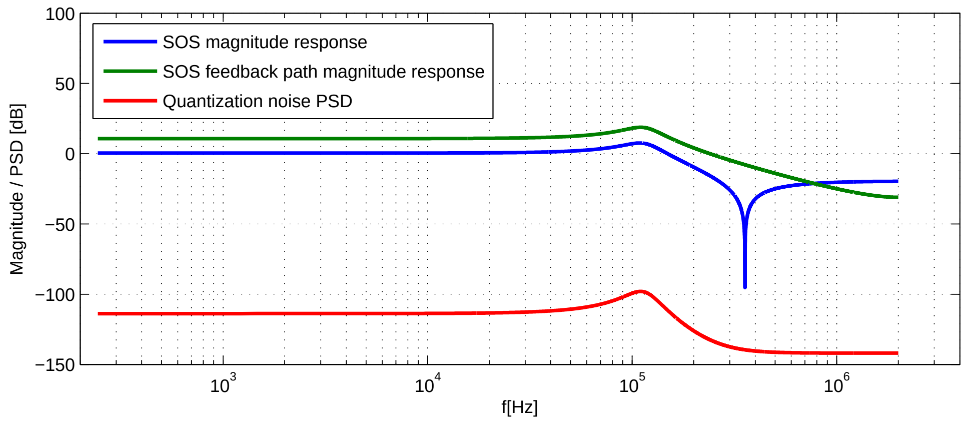


Figure 3.15: Magnitude responses and noise PSD of SOS2

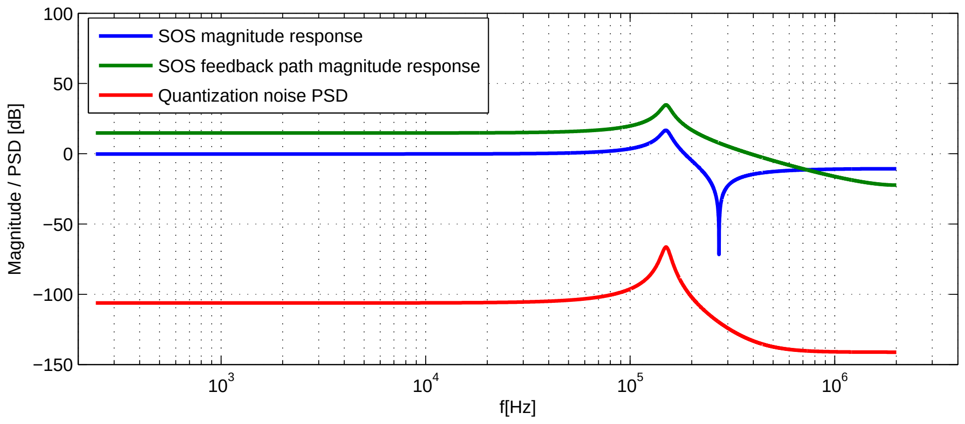


Figure 3.16: Magnitude responses and noise PSD of SOS3

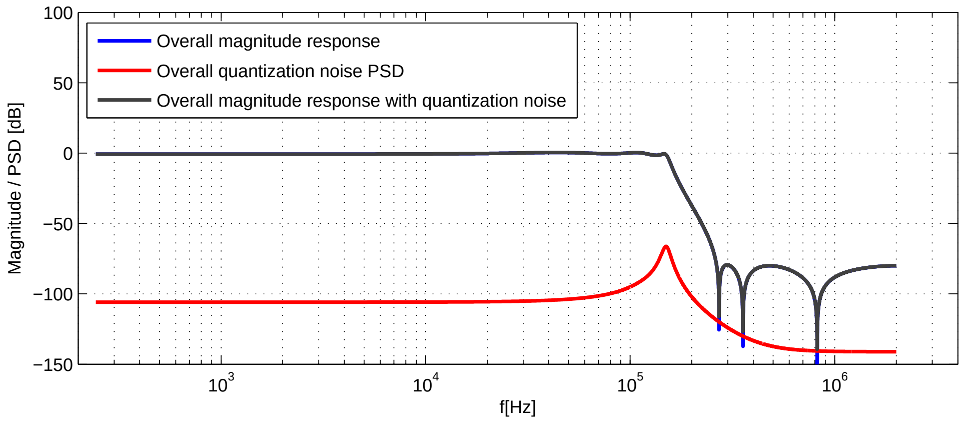


Figure 3.17: Effects of quantization

3.4 Optimization Parameters

The implemented optimization tool works based on a DE algorithm with multiobjective optimization as defined in Section 3.2. There are several objectives computed that will be discussed in this section.

3.4.1 Magnitude response

The target magnitude response is defined via the pass- and stopband attenuation as visualized in Figure 3.18. Its transition band is of low interest, since the pass- and stopband are limiting the slope anyway. In addition to that a weight for these two frequency ranges

is taken into account for customization purposes.
The fitness of the frequency response is calculated from two integrals over the range of pass- and stopband respectively. For the passband, any deviation from 0 dB worsens the fitness. For the stopband, this happens whenever the magnitude is above the upper stopband attenuation $A_{\text{stop,u}}$. The cost function for the magnitude response is defined as

$$F_{\text{mag}} = \alpha_{\text{pass}} \int_{\omega=0}^{\omega_c} |H(e^{j\omega})|^2_{\text{dB}}\; d\omega + \alpha_{\text{stop}} \int_{\omega=\omega_{\text{stop}}}^{\pi f_s} \sigma[\Delta A_{\text{stop}}(e^{j\omega})] \cdot \Delta A_{\text{stop}}(e^{j\omega})\; d\omega, \quad (3.25)$$

where α_{pass} and α_{stop} are the pass- and stopband weights, ω_c is the cutoff frequency, ω_{stop} is the stopband frequency and f_s is the sampling frequency. Furthermore,

$$\Delta A_{\text{stop}}(e^{j\omega}) = |H(e^{j\omega})|^2_{\text{dB}} - A_{\text{stop,u}}$$

and $\sigma(x)$ is the unit step function defined as

$$\sigma(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geqslant 0 \end{cases} .$$

Note that in practice the integrals of 3.25 are computed over sums with discrete values of ω.

3.4.2 Group delay

Whenever a filter is designed, the distortions of the filter in the passband should be minimized. As ripple of the magnitude is already covered, there is a further demand for a constant group delay in order to remove ISI.

For the group delay of a lowpass filter only the passband is of interest as signals in the stopband are significantly damped. The first derivative of the phase gives the group delay whereas the second derivative provides the slope and therefore directly the desired fitness constraint, given as

$$F_{\text{gd}} = \int_{\omega=0}^{\omega_c} \frac{\partial^2 \phi(e^{j\omega})}{\partial^2 \omega}\; d\omega, \quad (3.26)$$

with $\phi(e^{j\omega})$ being the phase response given as

$$\phi(e^{j\omega}) = \arg(H(e^{j\omega})). \quad (3.27)$$

Analogous to the magnitude response, the integral for the group delay is computed over a discrete ω.

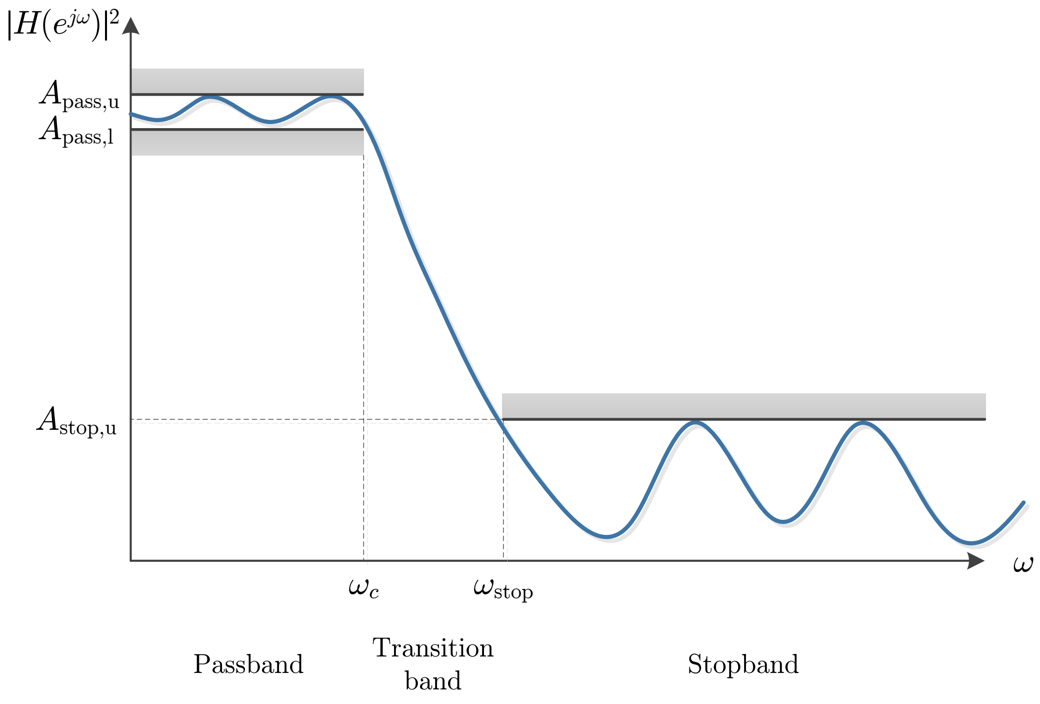


Figure 3.18: Arbitrary lowpass filter specification

3.4.3 Area constraint

As the implementation of the filter is done in digital hardware, the area of the IP inside the chip is directly related to

- the number of sections,
- the number of multipliers and adders and
- the bitwidths of scales, coefficients and internal filter states (size of multipliers).

All this constraints are reformulated to partial cost functions to contribute to the overall fitness of the individual. For this purpose, the optimization algorithm needs the bitwidths which are given by the numerical range of the filter input and the desired resolution.

3.4.4 Scales and coefficient power contribution

Any set bit within the scale or SOS coefficient multiplier values will increase the overall power consumption of the circuit. This objective therefore determines the number of all the set bits within a population and sums up all of them to get a single fitness value.

3.5 Algorithm Details

For the DE optimization employed in this work, the optimization for a filter specification and the system model are distinguished. Nevertheless they use the same optimization core and partially share cost functions.

3.5.1 Filter Specification

1) **Initialization**
   Generation of initial population of individuals (coefficient sets). One individual consists of a vector of scale values and a matrix of filter coefficients (SOS matrix). By default the coefficients are uniformly distributed around zero. Optionally there is the possibility to generate the initial population based on the double-precision floating-point solutions derived from the MATLAB Filter Design and Analysis Tool. The individuals are then distributed around this solution.

2) **Stability and SOS ordering**
   Stability needs to be checked as the initial as well as mutated chromosomes might deliver unstable solutions. If there are unstable solutions found, the poles are mirrored into the unit circle, i.e. $p = 1/|p|\ e^{j\cdot\arg(p)}$ where p is a pole of the filter individual that is located outside the unit circle.
   Ordering of sections is done according to [34] of all generated individuals. This not only results in an overall filtering improvement, but also reduces the quantization noise.

3) **Compute fitness**
   Obtain the characteristics for each individual that are needed in order to calculate the fitness. For the frequency response, add the quantization noise PSD as derived in Section 3.3.

4) **Evaluate fitness**
   Evaluate the multiobjective fitness functions described in Section 3.4. The overall fitness is merged by weighting of the partial cost functions for magnitude response, group delay, area and power contribution for each individual as discussed in Section 3.2.2. The weighting parameters were found by empirical test runs that tradeoff the partial costs against each other.

5) **Mutation and Selection**
   Mutation is done according to the DE algorithm as investigated in Section 3.2. The selection is based on the natural survival of individuals. Depending on the fitness the parents and childs stay within the population or not [17, 28].

6) **Termination**
   The termination condition is met if the number of maximum generations reached or

at least one coefficient set reaches a fitness less than a certain threshold. Otherwise the algorithm continues with step 2).

3.5.2 System Model

For the design of an optimized filter based on the system model the same DE core is reused. Still there are different cost functions that are evaluated at step 4), given as follows:

1) The receiver bit error rate (BER). Therefore a trainings sequence is provided at a certain E_b/N_0 ratio.
2) The area consumed by the filter design
3) The coefficient power contribution

For further details on 2) and 3) refer to Section 3.4.

4 Digital Design Tool Flow

In this section the actual tool flow and the design in digital hardware is presented. The register transfer level (RTL) design is tied to the system and high-level optimization tools presented in Chapter 3. Also, full verification downto gate-level (GL) is performed automatically in a sophisticated design process utilizing these models to provide bit-true reference signals. Two different implementation variants are presented depending on the application of the filter.

4.1 Filter Optimization Tool Flow

This section gives insight into the overall design flow that is used to design a filter. As mentioned and depicted in Figure 4.1, there are two ways to do so.

1) **Filter Specification Tool Flow**
   This flow starts with a lowpass filter specification together with a certain architecture (SOS by default). The specification is based on pass- and stopband constraints as discussed in Section 3.5. By default, the filter optimization tool generates its initial parameters for the scales and coefficients randomly without any prior knowledge. As an alternative MATLABs FDA tool which delivers double-precision floating-point scales and coefficients can be used to do so [14]. These are then quantized for the desired architecture. The filter optimization tool will generate an initial population

around this solution, as the best one is assumed to be in the vicinity of them depending on the quantization granularity. If a standard lowpass filter (Butterworth, Elliptic) is designed, the pre-processing leads to faster convergence although the optimization might deliver a completely different solution. Based on these two different initializations the filter optimization tool utilizes the algorithm described in Section 3.5 to compute optimized coefficients and scales given the constraints from the filter specification (cost functions).

2) **System Model Tool Flow**
Entry point for this flow is the baseband equivalent receiver model described in Section 2. Based on a system simulation a training sequence for optimization is generated. Therefore, an operating point is chosen at a certain E_b/N_0 ratio. Now, the filter optimization tool optimizes for this certain sequence using the algorithm described in Section 3.5.

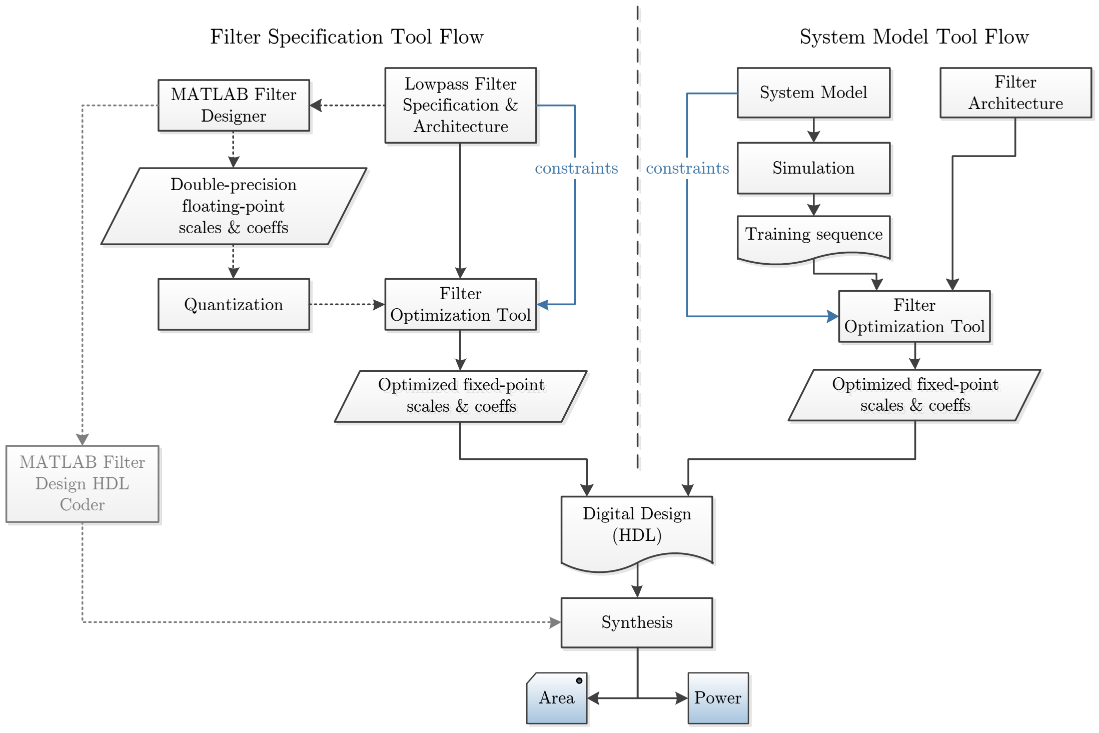

Filter Specification Tool Flow
System Model Tool Flow
MATLAB Filter Designer
Lowpass Filter Specification & Architecture
Double-precision floating-point scales & coeffs
Quantization
Filter Optimization Tool
Optimized fixed-point scales & coeffs
constraints
constraints
System Model
Filter Architecture
Simulation
Training sequence
Filter Optimization Tool
Optimized fixed-point scales & coeffs
MATLAB Filter Design HDL Coder
Digital Design (HDL)
Synthesis
Area
Power

Figure 4.1: Filter optimization tool flow

The optimization tool delivers fully quantized coefficients with the desired wordlengths which can then be plugged into the RTL model. By using a single script, following steps are performed for verification.

1) **RTL compilation**

The RTL code is compiled with the filter coefficients obtained by the optimization tool.

2) **Run RTL simulation**
The testbench driving the actual filter design implementation is provided with the coefficients. It performs an output comparison to the bittrue high-level implementation. A text based simulation result is provided together with a waveform database.

3) **Synthesis**
The filter model is synthesized and a Verilog netlist generated.

4) **GL compilation**
The RTL code is compiled to GL.

5) **GL simulation**
Verifies the model at gate-level by comparing its output to the bittrue high-level implementation.

This provides a straightforward filter implementation flow with full verification for any filter design.

4.2 Architecture and Design

In digital hardware a single SOS filter can be implemented as illustrated in Figure 4.2. It consists of the well know coefficients and a scale value, provided as registers or generics, at the input which ensures that the number range of the following section stays within a certain range. The boundaries of the block (sos_in and sos_out) are registered, i.e. implemented as Flip-Flops. This is very important as the combinatorial depth of the multipliers is large and the hazards/toggling activity needs to be minimized in order to reduce the power consumption. The filter itself has two delay lines, state1 and state2. As the input is assumed to be registered already, the overall filter section has a delay of three clock cycles, two for the delay lines and one for registering the output. This constant delay is negligible in general as the filters run at oversampled clock frequencies compared to the symbol rate.

There are two different implementation variants demanded depending on the application. Firstly, there is a variant with fixed coefficient sets at synthesis/compile time. Secondly, there is a variant that has the ability to configure the coefficients during runtime (programmable). Both of these variants are implemented with the same filter structure and number representation.

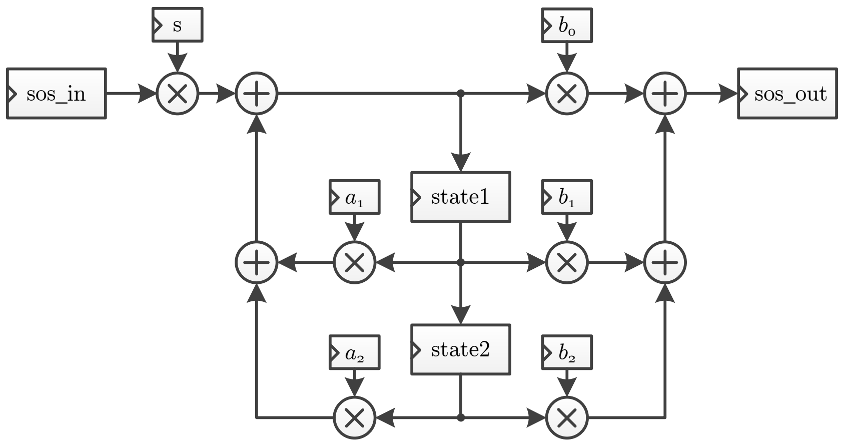


Figure 4.2: Second-order section in digital hardware

4.2.1 Number representation

As the filter is design in fixed-point arithmetic, a standard two's complement (`signed`) representation is used. In addition to that the canonic signed digit (CSD) representation will be evaluated. It's a different radix-two number representation that needs 33% less non-zeros compared to standard two's complement representation on average. This is achieved by introducing a ternary coefficient set $\{0,\ 1,\ \bar{1}\}$, where $\bar{1}$ denotes a subtraction of the actual bit value from the total value of the number. For instance, the number 15 can be represented as

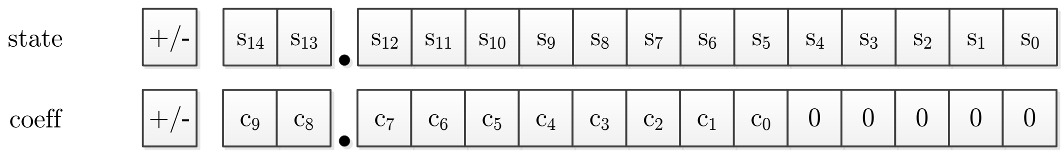
$$15_d = 01111_d = 2^4 + 2^3 + 2^2 + 2^1.$$

Using CSD it can be written as

$$15_d = 1000\bar{1}_{\text{CSD}} = 2^5 - 2^1.$$

The number of non-zero bits is halved in this case. For DSP designs this technique has been widely used as in hardware implementations this reduces both area and power at the same time [13]. For multiplication of a standard two's complement fractional number with a CSD number a modification of *Horner's method* can be utilized [35]. This method searches for the non-zero bits in the CSD multiplicand, i.e. 1 and $\bar{1}$, and in case performs the multiplication with the two's complement number and the corresponding weight (shift by power of two). If the CSD bit is 1 the value is added to the accumulator but subtracted if it is $\bar{1}$. For an example code refer to Listing A.3.

The numbers are represented in the Q-format where fixed-point, two's complement numbers can be represented with integer and fractional portions. A $Qm.n$ number has m integer and n fractional portions. Thus the overall bitwidth $b = m + n + 1$, where one additional bit is used for the sign information. For efficiency reasons, the coefficients do not have any fractions. They are aligned with the fractional point of the state register. From that point it is clear that the bitwidth of the scales and coefficients needs to be less or equal than the bitwidth of the internal state signals. Figure 4.3 shows a multiplication alignment example of a state register s represented as $Q2.13$ and a coefficient c with bitwidth 10. The coefficient is shifted by five bits to the right to be aligned with the fractional point. The other bits are filled up with zeros for multiplication.



state
+/-
s_{14} s_{13} s_{12} s_{11} s_{10} s_9 s_8 s_7 s_6 s_5 s_4 s_3 s_2 s_1 s_0
coeff
+/-
c_9 c_8 c_7 c_6 c_5 c_4 c_3 c_2 c_1 c_0 0 0 0 0 0

Figure 4.3: Alignment of $Q2.13$ state with 10-bit coefficient for multiplication

By looking at the signal flow graph in Figure 4.2 it is clear that by addition/subtraction of the different state variables the number range of the state signals might be exceeded. Therefore, a certain amount of integer bits needs to be preserved depending on the chosen coefficients.

4.2.2 Fixed Coefficient Variant

This implementation variant has fixed coefficients that are once computed/optimized during the filter design process. Multiplication is performed by addition of the intermediate results in hardware. The number of binary multipliers, and hence the number of adders, is proportional to the number of bits set within the coefficient. Therefore, this variant needs reduced area as not all adder paths of the multipliers need to be instantiated. If a bit of the coefficient/scale value is zero, the intermediate result is zero and thus there is no contribution to the result. These adder paths are not instantiated in hardware then.

Introducing CSD coefficients for this implementation variant decreases the area directly and is considered as potential improvement for power consumption as well. Nevertheless, the bitwidth of the coefficient/scales for the actual designs are relatively small compared to that of the state variables at the delay sections. This means, that the amount of non-zeros decreased by CSD has relatively low impact.

4.2.3 Programmable Coefficient Variant

This implementation variant has programmable coefficients that can be changed freely during runtime of the chip via dedicated configuration registers. Thus, all adder paths of the multipliers need to be instantiated which leads to a significant increase in area.

Implementing CSD multipliers for this implementation variant is not effective due to the following reasons:

1) All adder paths of the multipliers need to be realized in hardware. Note that this is independent of the number representation as any binary combination for the coefficients/scales is possible.

2) The specification in a datasheet with CSD multipliers is not welcome. Hence, a CSD to binary wrapper would be necessary, causing additional hardware.

4.2.4 Design

The actual design is hierarchically split up into the major filter blocks. The filter is implemented with second-order section submodules which are cascaded to build the overall filter. A dedicated multiplier cell is designed as utility function which performs the fractional shifting of coefficients, the multiplication and truncation of the result to the desired bitwidth. A second multiplier cell is used for CSD multiplication which can be substituted.

By default there are two main filter types, namely Butterworth and Elliptic type implemented. As discussed in Section 3.1 these two implementation variants only differ in the coefficient $b2$. The switching between these architectures is rather easy by simply evaluating this single coefficient. In fact, if $b2$ is set to zero, Butterworth sections are implemented, elleptic ones otherwise.

4.3 MATLAB Filter Design HDL Coder

MATLABs filter design and analysis tool has a built-in code generator that produces VHDL code out of the actual filter design. By analysing the auto-generated code, it can be observed that a high combinatorial depth is used. This results in high toggling activity. The generated code is fixed for one certain bitwidths for scales and coefficients. Therefore it is not worth to use this model as a basis for an IP as parametrisation for different designs is not possible by simply changing the coefficients. One would always need to generate a completely new VHDL code (instead of coefficients) out of a certain filter design. This is impractical as small modifications and adoptions to the auto-generated VHDL model will need to be done over and over again. Furthermore, the tool lacks in configuration possibilities in terms of coding standard. A generated code example is presented in Listing A.4.

The auto-generated testbench is rather easy, it applies stimuli and compares it to a reference model output in time-domain. A lot of utility functions are generated inside an additional package, nevertheless most of these functions aren't used at all. Another drawback is, that the stimuli file is embedded in a VHDL source file. Thus for different input stimuli the testbench needs to be recompiled every time. This is impractical for automation of the verification process.

4.4 Verification

The verification of the designed filter is done within a testbench as defined in [36]. Firstly, it generates a periodic system clock and sampling clock signal for driving the simulation. Secondly, it provides stimuli vectors and applies them to the model under test (MUT) at well-defined moments in time. Thirdly, it loads the stimuli data from the high-level model as a reference against which to compare. Fourthly, it creates a simulation report which points to timing wise problems and optionally prints debug output.

The stimuli test data can be one of the following waveforms:

- Dirac delta impulse
- Unit step function
- Sinusoid
- Input sequence directly taken from the system model

As mentioned the high-level model is fully bittrue (fixed-point arithmetic) and therefore direct comparison with the design can be performed.

4.5 Analysis

In order to compare the design to the existing implementation, an area and power analysis is performed. Since the existing and new design are rather compatible in terms of the outer interface, a meaningful measure can be exhausted. Before presenting the results the synthesis flow is explained in more detail.

4.5.1 Synthesis

The synthesis is done using CADENCE ENCOUNTER® RTL COMPILER, the simplified synthesis flow is presented in Figure 4.4. On RTL the IP is compiled and simulated, actually this is where the verification takes place. The output of this step is the report with the timing wise differences between the MUT and the high-level model. In order to estimate the area of the design a synthesis is performed. Given the input and output delays as constraints, the area, a *Verilog* netlist and the corresponding delay format file (SDF) is created. Finally also a GL simulation is done where the reference signals can be compared against. The waveform file from the GL simulation now specifies exactly the stimuli applied for the power consumption analysis which gives the overall and most important measure of the design.

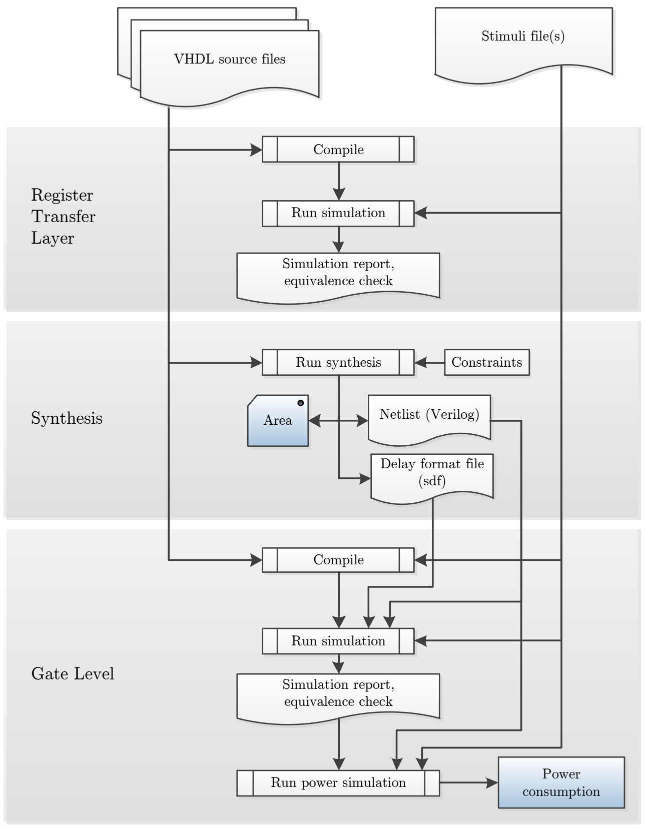


Figure 4.4: Register Transfer Layer, Synthesis and Gate Level simulation flow

Synthesis runtime

The synthesis runtime is decreased by 28% compared to the existing design, the corresponding measurements are presented in Table 4.1.

Existing Design $[s]$	New Design $[s]$
94	68

Table 4.1: Synthesis runtime comparison

4.5.2 Area and Power

In order to compare the area and power of the existing and new design the different designs are fed with the same input data and the synthesis as well as a power simulation is performed. This is provided in Table 4.2 for the two different implementation variants, fixed or programmable coefficients. Also, the number of bits used for the state registers, coefficients and scales are provided. Corresponding area and power for the different designs of a third-order butterworth filter are given based on a 0.18 μm process. The data sequence used for this comparison is based on the system model input data for the filter, thus hardware and application near.

Design	Coeffs	State	Coeff	Scales	Cells	Area	Power
		[bits]	[bits]	[bits]		[μm^2]	[μW]
Existing design	fixed	20	8	16	2894	74733	1238
MATLAB HDL coder	fixed	16	16	16	1841	50470	2693
Improved design	fixed	20	9	17	1881	53640	697
Improved design	prog.	20	9	17	6622	149653	1187
Improved design (CSD)	fixed	20	9	17	1632	43970	532

Table 4.2: Area and power comparison for different designs

The first line of Table 4.2 presents the area and power of the existing design which is the reference to compare. It is fixed to 8 bits for the coefficients and 16 bits for the scales. Unfortunately with MATLAB HDL coder it is not possible to set the same bit lengths since the number of bits for coefficient and scales always needs to be the same. A meaningful comparison is therefore impossible. However, the improved design can be configured to meet exactly the same configuration. For the fixed coefficient variant the area is reduced by roughly 28% while at the same time the power consumption is reduced by 44%. The implementation variant with programmable coefficients has an expected, tremendous increase in area as all multipliers need to be fully instantiated. However, the power consumption is still less compared to the existing implementation with fixed coefficients. There is some further, significant improvement to both area and power by utilizing CSD coefficient multipliers. As mentioned in Section 4.2 CSD multipliers are only implemented for the fixed coefficient variant.

5 Results

This section provides the results by evaluating the different optimization procedures. The filter design problems based on a given filter specification and based on the whole system model are investigated respectively.

5.1 Quantized Filter Optimization

In Figure 5.1 the optimization of the magnitude response is presented. The filter specifications are taken from the system model as defined in Section 2.6 (indicated with horizontal/vertical red lines). For a standard third-order elliptic filter the reference double-precision solution is obtained using MATLABs FDAtool. The directly quantized solution is plotted as well, it matches the reference solution quite well as a bitwidth of 20 is chosen for this simulation run. The reference filter can be compared to the final solution found by the DE. It fits the filter specification exactly. The intermediate best results are plotted in dashed blue lines which are the best candidate solutions found after each generation.

Zooming into the passband (Figure 5.2) shows that the magnitude response is flattened out completely. Especially remarkable is the constant gain of 0 dB in the passband which isn't even achieved by the reference MATLAB double-precision solution.

By looking at the group delay in Figure 5.3 it can be observed that it is smoothed out compared to the reference filter, especially the passband has almost constant group delay. This is possible due to the fact that the transition band of the magnitude response doesn't

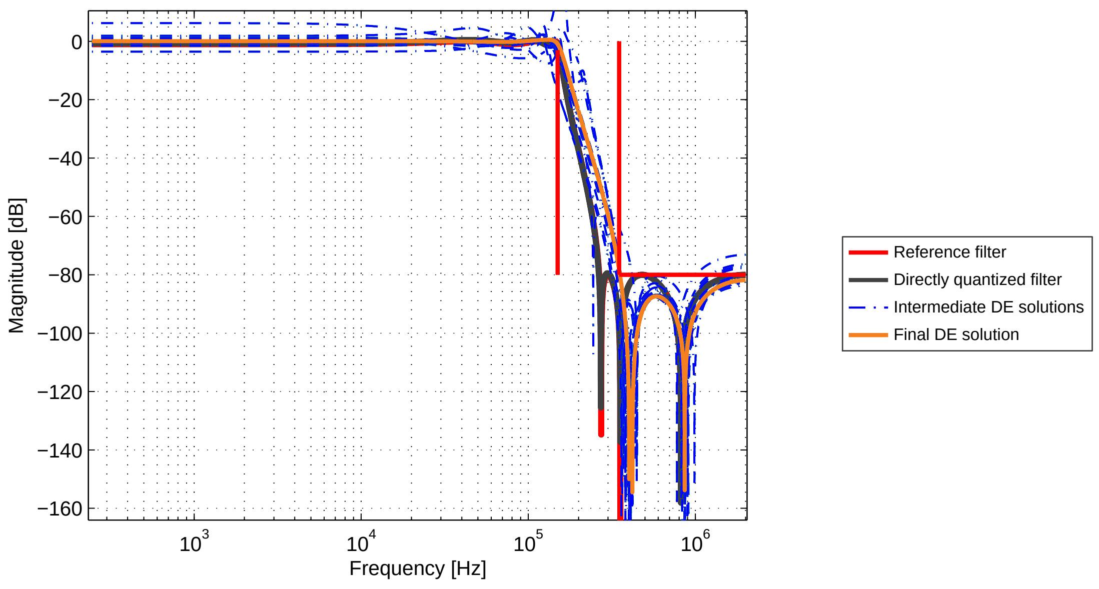


Figure 5.1: Magnitude response

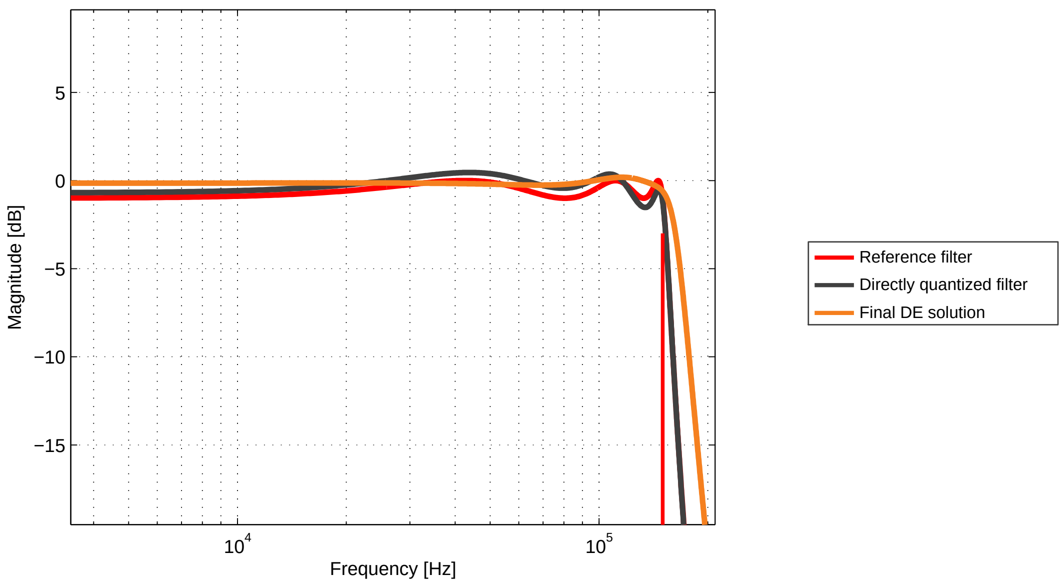


Figure 5.2: Magnitude response (passband zoomed)

need to have such a steep slope as the reference solution provides. Therewith some play is introduced for optimization of the group delay as well.

In order to demonstrate the evolvement over generations, a waterfall plot of the best solutions found iteratively is shown in Figure 5.4. As the inital coefficients are based

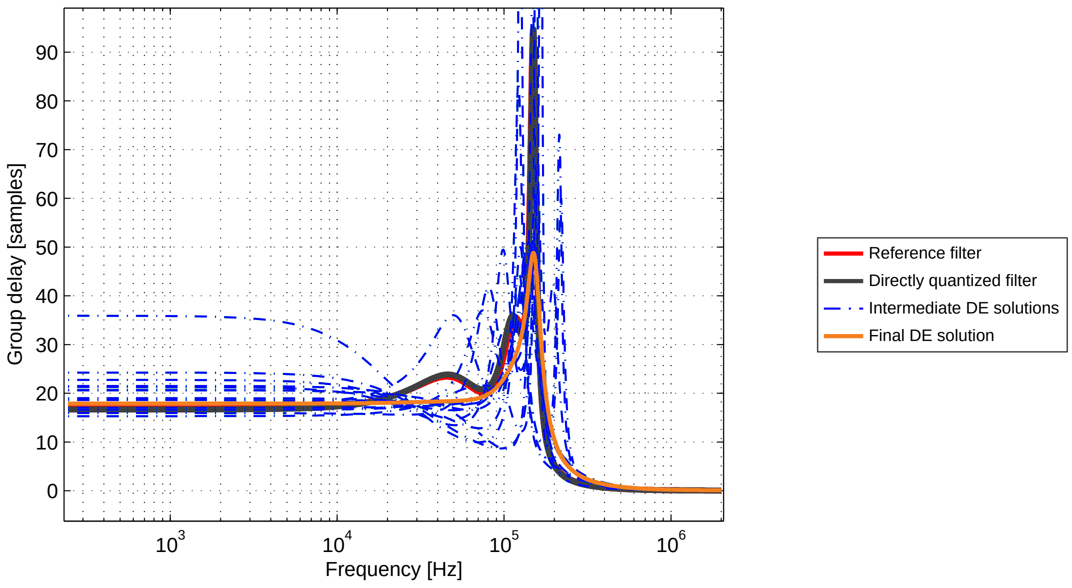


Figure 5.3: Group delay

on the double-precision solution from MATLABs FDAtool for this simulation run, the convergence is quite fast. In contrast to that, the convergence time is much slower if the filter individuals are initialized randomly as depicted in Figure 5.5.

Figure 5.6 shows the evolvement of the best and average fitness over the optimization generations. As in general for population based algorithms the fitness drops very fast in the beginning and stagnates after a while [21]. It is often claimed that algorithms such as GA have premature convergence [2]. This can be partly contradicted by this DE approach by observing Figure 5.7 and 5.8. While the scales and coefficients of the best solution are changing quite frequently initially, the update steps get smaller as the number of iterations increases. Still, after some 2000 generations the algorithm finds a new valley. After approximately 3000 generations the algorithm found a solution with a fitness less than the target threshold.

The poles and zeros are shown in Figure 5.9. By comparing it to Figure 3.4(b) it is noticeable that the placement of poles within the unit circle is retained approximately.

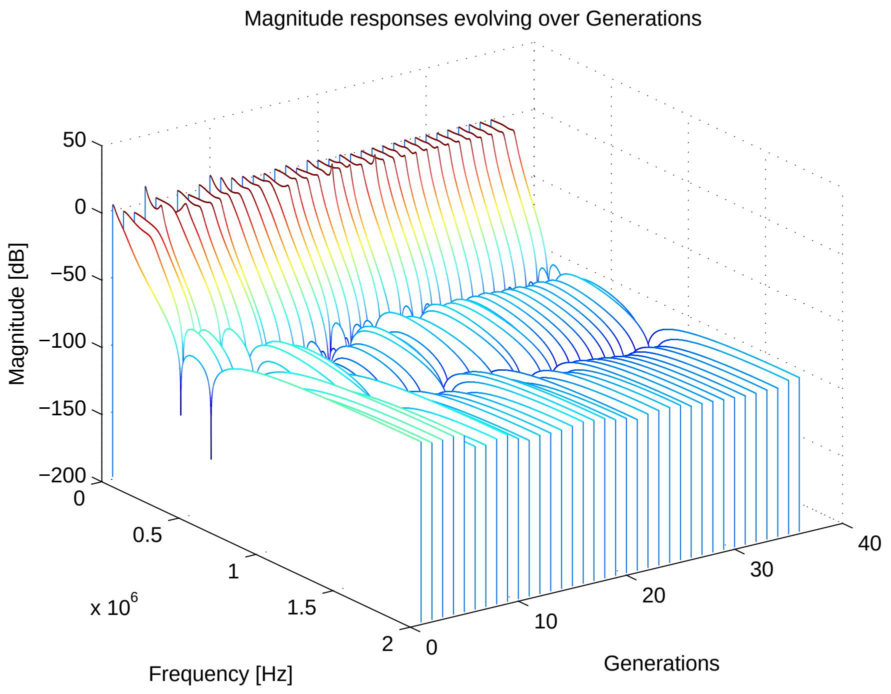


Figure 5.4: Magnitude response over generations

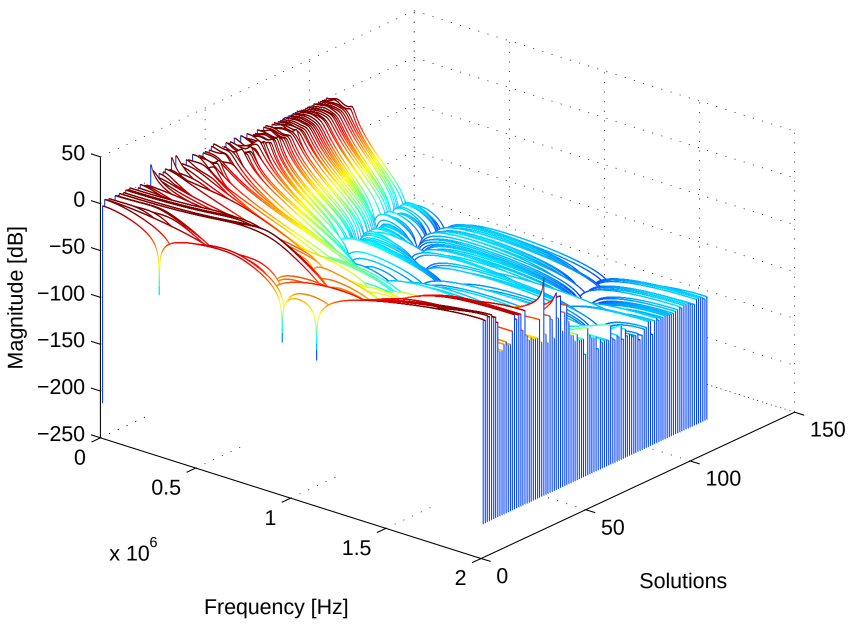


Figure 5.5: Magnitude response over generations (random initial coefficients)

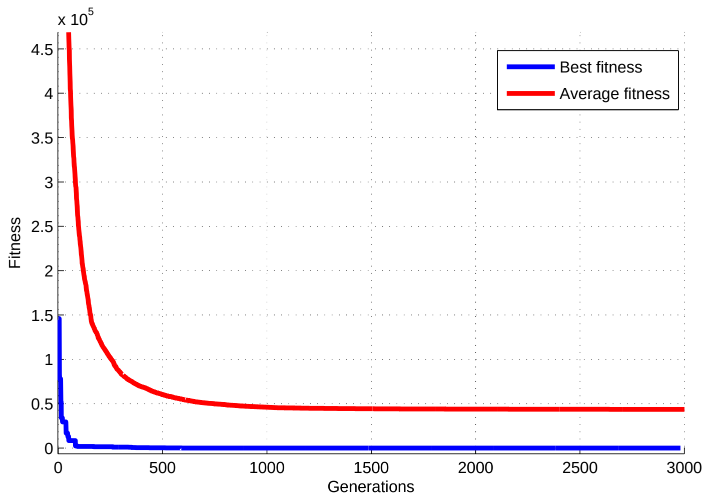


Figure 5.6: Fitness (best individual) and average fitness over generations

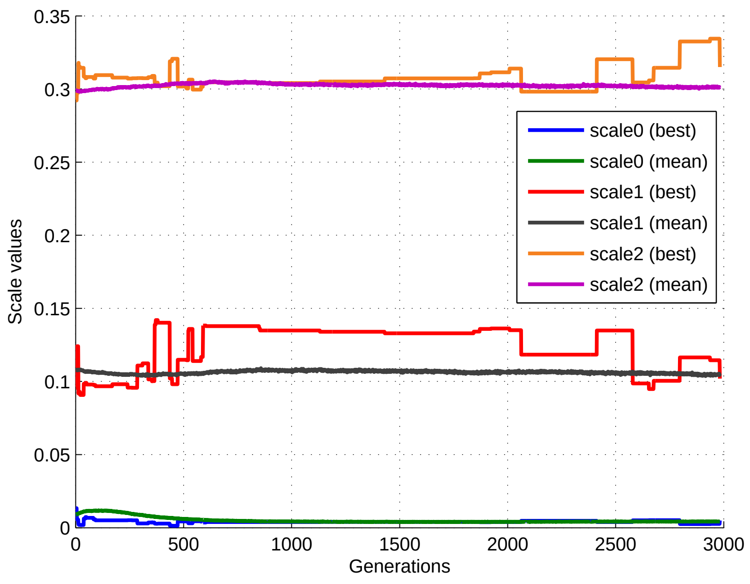


Figure 5.7: Scales over generations

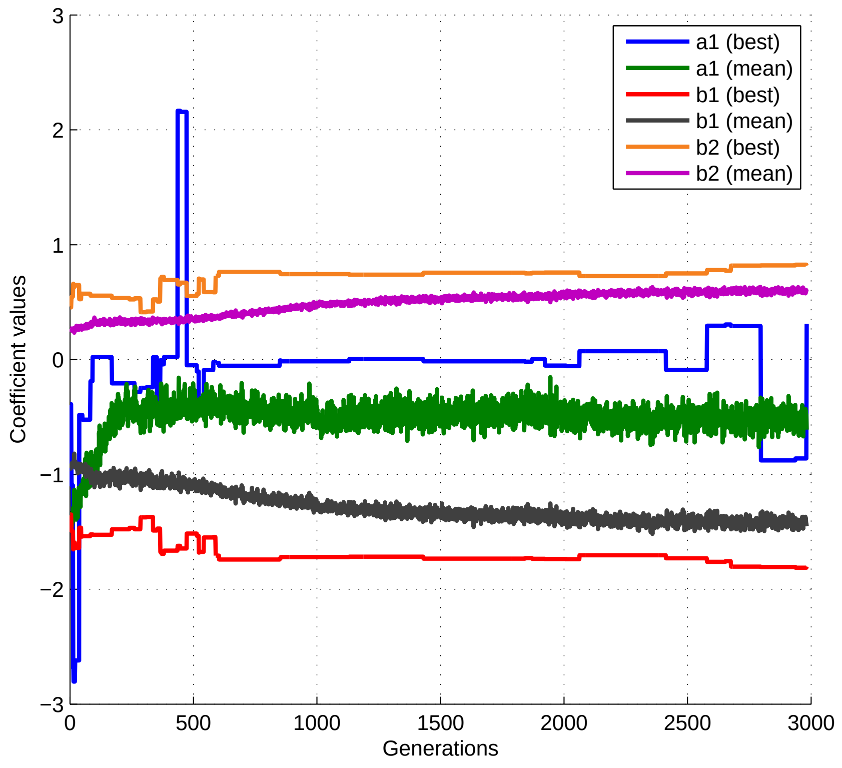


Figure 5.8: Coefficients over generations

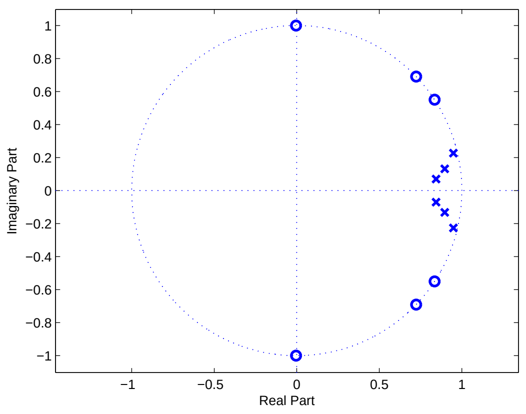


Figure 5.9: z-plane

5.2 System Model Optimization

As mentioned in Section 3.5 a DE is used as well for finding optimum filter coefficients and scales. In Figure 5.10 the recevier BER is evaluated with the optimized solution found by the DE. Note that it reaches the same performance as found be the MMSE optimization filter derived in Section 2.8. The simulations are done for 10^6 bits each.

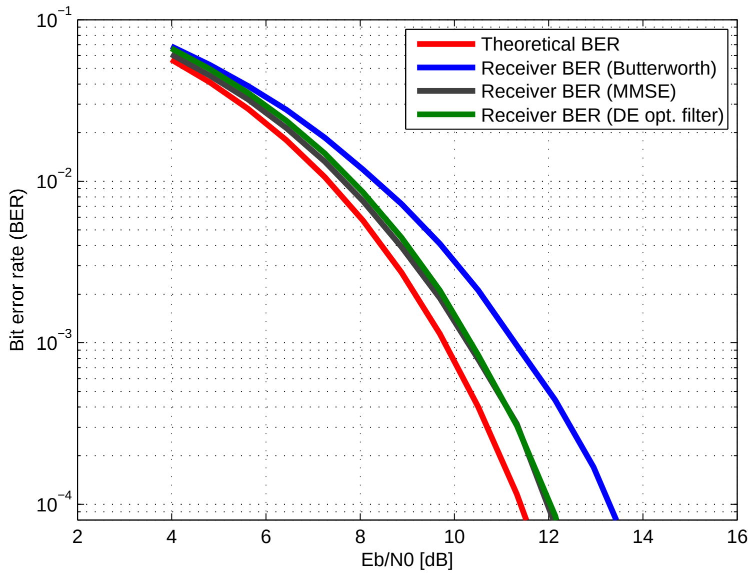


Figure 5.10: Bit error probability

5.2.1 Effects of Quantization

The receiver performance will now be evaluated using fixed- and floating-point arithmetic. As there are only bittrue reference models for a Butterworth and elliptic architecture, this analysis is done based on the existing design solution which is a Butterworth filter. However, the purpose is to demonstrate the effects of quantization on the receiver performance with the two different number formats.

Fixed-point

The fixed-point evaluation is performed using the well known $Qm.n$ format. A BER simulation is done for different number of fractional portions n for the internal state registers of the SOS. Especially for fixed-point arithmetic the magnitude of the input signal is of major importance as no dynamic scaling is performed prior to the filter input. Thus, the number of fractions needs to be sufficient to cover the whole range of the receivers amplitude range. As illustrated in Figure 5.11 with $Q2.10$ and $Q2.14$ the number of fractional bits is not sufficient to receive correct data. Starting from $Q2.20$ the maximum performance for this filter is reached ($Q2.20$, $Q2.22$ and $Q2.24$ result in same BER curves). Obviously, the quantization noise for n less than 20 exceeds the noise introduced by the channel resulting in a worse performance.

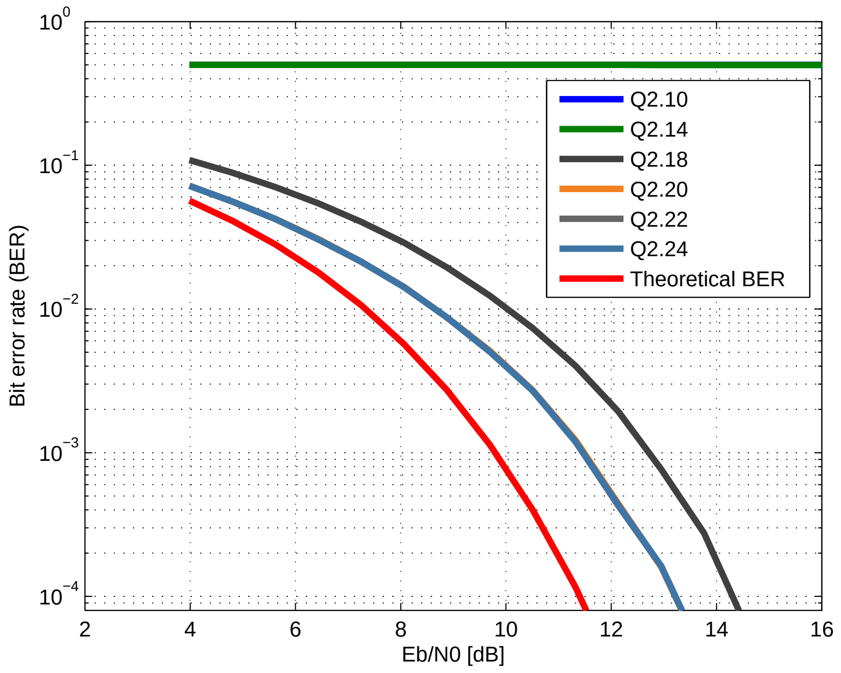


Figure 5.11: Bit error probability with different fixed-point bitwidths

Floating-point

For the floating-point scenario, simulation runs for various floating-point precisions are performed and the BER evaluated. The exponent is kept at a constant value. As illustrated in Figure 5.12 a precision of 5 is not sufficient to receive correct data. Starting from precision 11 the maximum receiving performance is reached. As the exponent is held constant at 7 for all runs, the actual number of bits needed to obtain the maximum performance is 18, compared to 22 for the fixed-point scenario. This evidences the advantageous scaling feature of floating-point arithmetic.

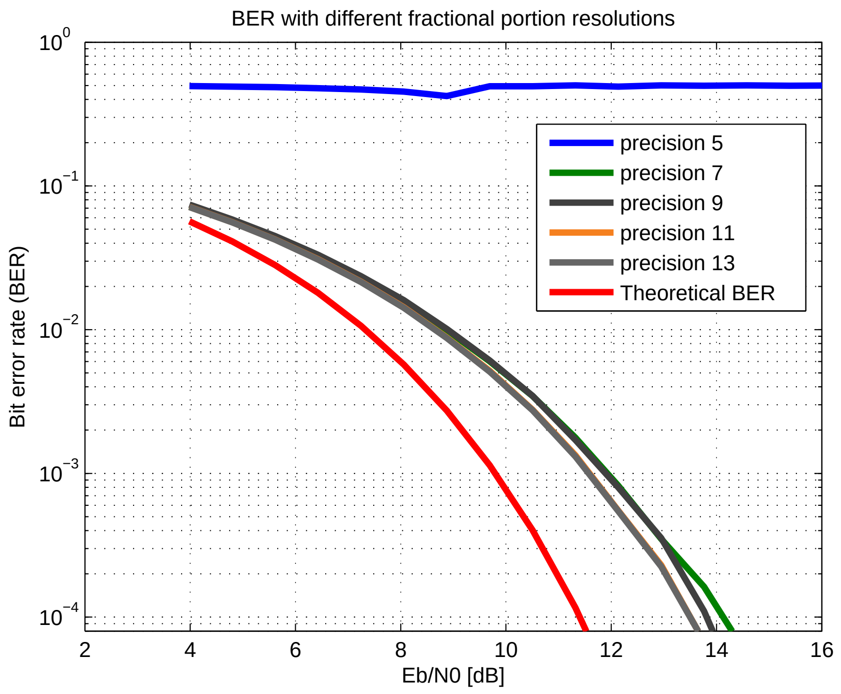


Figure 5.12: Bit error probability with different floating-point precisions

6 Conclusion and Outlook

This thesis shows the optimization potential for the presented RF receiver and a semi-automatized design flow to derive optimized filter coefficients quantized for digital hardware implementations. The RF receiver operating in standard ISM bands is a low-IF receiver with fully digital IF processing and downconversion to baseband. The system is modelled in a complex baseband equivalent. This simplifies the complexity of the model and therewith the optimization procedure.

By considering the system model as fixed, the probability of bit errors is decreased by the optimized filter design by roughly 1.2 dB at the sensitivity limit (Section 5.2). This is accomplished by utilizing a DE approach which targets at minimizing the BER. It achieves the same performance as the MMSE equalizer introduced in Section 2.9 and chosen as reference solution. The DE delivers fully quantized coefficients, optimized for low-area and low-power implementations in digital hardware.

For the RF receiver itself a potential improvement has been proposed. The CIC decimation filter could be cascaded with a compensation filter that flattens the sinc shaped magnitude response in the passband to unity (Section 2.5).

The implemented filter optimization tool for filter design based on filter specifications utilizes the same DE core as for system model optimization. It uses partial cost functions to find the optimum solution for the magnitude response, group delay as well as area and power constraints for digital hardware implementation. The advantage of the population based heuristic is that the partial costs are applied simultaneously. The DE delivers quantized filter coefficients that can be plugged directly into the HDL model. A toolset is implemented to automatize this process.

Due to a new digital implementation of the IIR filter IP an area saving by 28% and a power consumption reduction by 44% is achieved compared to the existing design (Section 4.5). Utilizing CSD coefficients instead of a two-complement representation gains further improvements regarding area and power. However, this number format is impractical if the coefficients are intended to be programmable. Verification of the digital design is performed by simple input-output comparison against the bittrue high-level model with auto-generated test patterns from RTL downto GL.

Regarding the usage of MATLAB's MEX functionality it can be said that the interface to native C code is a potential tool for high-level reference designs. Also, the MPFR library has proven to be a powerful tool for bittrue floating-point computations as it enables the possibility to set variable lengths for the mantissa and exponent. It should be mentioned, however, that the debugging features of MATLAB are no longer available when calling native C code via MEX resulting in a slightly more complex implementation. In addition to that, stability issues arise if the memory handling is erroneous.

A high-level synthesis based on the integrated HDL coder of MATLAB has been investigated. It shows that it is not suitable for reuse in IPs as it lacks in configurability and code readability. A synthesis based on this auto-generated HDL model has been investigated, resulting in a relatively high area and power consumption. It is therefore not recommended to use such an approach for optimized designs. However, this tool could be investigated for prototyping as it delivers a fast, fully synthesizable RTL model.

For future work the implemented framework could be extended to enable the design of arbitrary filter transfer functions as proposed in [28]. This can be easily accomplished as only the cost function needs to be changed, the DE algorithm and its framework are fully reusable. Furthermore, the filter structure could be a possible optimization target. Thus, the heuristic could search for potential alignments of the multipliers and adders by itself.

From the digital design point of view, a floating-point implementation in hardware could be investigated. There are tools to auto-generate code out of a high-level model. Issue at stake is then the resulting area and power consumption, especially as the number of bits for the internal state registers of the filter needed for the floating-point design is less compared to the fixed-point one (Section 5.2).

A Appendix

A.1 System Model MMSE Solution

The MMSE solution, that is used for performance evaluation, will be derived for the receiver system as illustrated in Figure 2.13. The goal is to find an analytic expression for calculating the optimum filter coefficient vector $\boldsymbol{c} = [c_0, c_1, ..., c_{L-1}]^\mathsf{T}$ of length L that minimizes the bit errors. The cost function is defined as

$$J(\boldsymbol{c}) = \sum_{n=0}^{N_b} e^2[n] = E(\boldsymbol{e}^2), \tag{A.1}$$

where N_b is the number of transmitted symbols for the training sequence. The error $e[n]$ compares the delayed transmitted sample to the actual received one, given as

$$\begin{aligned} e[n] &= d_b[n - \Delta] - y_b[n] \\ &= d[m - \Delta M] - y[m] \\ &= d[m - \Delta M] - \boldsymbol{c}^\mathsf{T}\boldsymbol{x}[m], \end{aligned} \tag{A.2}$$

where Δ is the delay caused by the transmission and the group delays of the different filters. The input signal $x[m]$ to the filter consists of the data samples filtered by $H_T(z)$ and $H_R(z)$ as well as the noise $\nu[m]$ filtered by $H_R(z)$, which is

$$x[m] = \boldsymbol{h}^\mathsf{T}\boldsymbol{d}[m] + \boldsymbol{h}_R^\mathsf{T}\boldsymbol{\nu}[m], \tag{A.3}$$

where $\boldsymbol{h} \circ\!\!-\!\!\bullet H(z) = H_T(z)H_R(z)$ and $\boldsymbol{h}_R \circ\!\!-\!\!\bullet H_R(z)$ are the impulse responses of these systems.

For the derivation of the optimum MMSE solution we plug (A.2) into (A.1) to obtain

$$\begin{aligned} J(\boldsymbol{c}) &= E(d[m-\Delta M] - \boldsymbol{c}^\mathsf{T}\boldsymbol{x}[m])^2 \\ &= E([d[m-\Delta M] - \boldsymbol{c}^\mathsf{T}\boldsymbol{x}[m]]\ [d[m-\Delta M] - \boldsymbol{c}^\mathsf{T}\boldsymbol{x}[m]]) \\ &= \underbrace{E(d^2[m-\Delta M])}_{\sigma_d^2} - 2\boldsymbol{c}^\mathsf{T}\underbrace{E(\boldsymbol{x}[m]d[m-\Delta M])}_{\boldsymbol{p}_{xd}} + \boldsymbol{c}^\mathsf{T}\underbrace{E(\boldsymbol{x}[m]\boldsymbol{x}^\mathsf{T}[m])}_{\boldsymbol{R}_{xx}}\boldsymbol{c}, \end{aligned} \tag{A.4}$$

where σ_d^2 is the variance of the input signal $d[m]$, $\boldsymbol{p}_{xd}$ is the cross-correlation vector between $\boldsymbol{x}[m]$ and $d[m]$ and $\boldsymbol{R}_{xx}$ is the autocorrelation matrix [37]. The cost function simplifies to

$$J(\boldsymbol{c}) = \sigma_d^2 - 2\boldsymbol{c}^\mathsf{T}\boldsymbol{p}_{xd} + \boldsymbol{c}^\mathsf{T}\boldsymbol{R}_{xx}\boldsymbol{c}. \tag{A.5}$$

As we are facing an optimization problem we derive the cost function after $\boldsymbol{c}$, i.e.

$$\nabla_{\boldsymbol{c}} J(\boldsymbol{c}) = 0 - 2\boldsymbol{p}_{xd} + 2\boldsymbol{R}_{xx}\boldsymbol{c}. \tag{A.6}$$

To find the minima of the cost function, we set the derivative to zero, which is

$$\nabla_{\boldsymbol{c}} J(\boldsymbol{c}) = \boldsymbol{R}_{xx}\boldsymbol{c}_{\text{opt}} - \boldsymbol{p}_{xd} \equiv 0. \tag{A.7}$$

Thus, the optimal coefficients are computed as

$$\boldsymbol{c}_{\text{opt}} = \boldsymbol{R}_{xx}^{-1}\ \boldsymbol{p}_{xd}. \tag{A.8}$$

A.2 Theoretical BER of the RF Receiver

The data reception is based on a simple threshold decision as the symbols transmitted are binary ASK. The reference BER comes out of theory for the optimum detector that can be realized either by the correlation or matched filter demodulator [24]. As depicted in Figure A.1 the received signal for the symbol s_0 is

$$y(t) = \nu(t) \tag{A.9}$$

while for symbol s_1 it is

$$y(t) = \sqrt{E_b} + \nu(t), \tag{A.10}$$

where $\nu(t)$ is a Gaussian random variable with zero-mean and variance $\sigma_\nu^2 = \frac{N_0}{2}$. As the symbols occur with equal probability, the decision threshold is set as $\sqrt{E_b}/2$. The conditional PDF that either s_0 or s_1 is received is defined as

$$p(y|s_i) = \frac{1}{\sqrt{2\pi}\sigma_n} e^{-\frac{(y-\sqrt{E_b}/2)^2}{2\sigma_\nu^2}} \qquad \forall\ i = 0, 1. \tag{A.11}$$

Substituting σ_n we get

$$p(y|s_i) = \frac{1}{\sqrt{\pi N_0}} e^{-\frac{(y-\sqrt{E_b}/2)^2}{N_0}} \qquad \forall\ i = 0, 1. \tag{A.12}$$

We define a conditioned error criterion, i.e. the probability that the wrong symbol is received. For symbol s_1 that is

$$\begin{aligned} p(e|s_1) &= \frac{1}{\sqrt{\pi N_0}} \int_{-\infty}^{0} e^{-\frac{(y-\sqrt{E_b}/2)^2}{N_0}} dy \\ &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{-\sqrt{E_b/2N_0}} e^{-x^2/2} dx \\ &= \frac{1}{\sqrt{2\pi}} \int_{\sqrt{E_b/2N_0}}^{\infty} e^{-x^2/2} dx. \end{aligned} \tag{A.13}$$

Introducing the Q-function

$$Q(x) = \frac{1}{\sqrt{2\pi}} \int_{x}^{\infty} e^{-t^2/2} dt \tag{A.14}$$

and plugging it into (A.13) we obtain the theoretical BER, defined as

$$p(e|s_1) = Q\left(\sqrt{\frac{E_b}{2N_0}}\right). \tag{A.15}$$

This holds for $p(e|s_0)$ in the same way as the PDFs are fully symmetric and the symbols are assumed to occur with same probability.

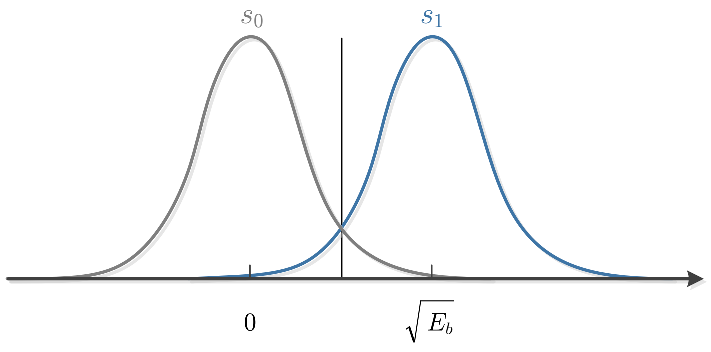


Figure A.1: Binary signal detection: Errors and probability distribution

A.3 MATLAB MEX and MPFR Library

In order to implement a fully bittrue high-level model of the filter with relatively high computation performance native C code is called from MATLAB using the standard MEX interface. Of course one could use the *Fixed-Point Toolbox* with the same outcome, however from a implementation point of view the easy and flexible MEX interface seems more convenient. Furthermore, for floating-point computations the GNU MPFR library is used.

A.3.1 Bittrue fixed-point implementation

This code shows an SOS fixed-point filter implementation for Butterworth and Elliptic form.

```
#include <mex.h>
#include <stdio.h>
#include <math.h>

typedef long filtertype;

int bitwidth        = 16;
int frac            = 13;
int inout_bitwidth  = 16;
int inout_shift     =  4;
int scales_bitwidth = 16;
int coeff_bitwidth  = 16;

// -----------------------------------------------------------------------
// -- multfrac
// -- Fractional multiplication of a and b. oper_bitwidth provides the
// -- bitwidth of b (smaller).
// -- b is aligned with the fractions of a
filtertype multfrac(filtertype a, filtertype b, const int oper_bitwidth)
{
  b = b * pow(2, frac-(oper_bitwidth-1)); // Align fractional point of b

  long long temp = a * b;

  filtertype result = temp >> frac;

  // Saturation
  if(result > ((1<<(bitwidth-1))-1)) result = pow(2, bitwidth-1) - 1;
  if(result < -(1<<(bitwidth-1))) result = -pow(2, bitwidth-1);

  return result;
}

// ---------------------------------------------------------------------------
// -- sosfilt
// -- Second order section filtering
void sosfilt(filtertype* x, filtertype* y, filtertype* coeffs,
             filtertype* scales, const int num_samples,
             const int num_sections)
{
  int i, j, b;
  filtertype states[num_sections][3]; // One second order section consists
                                      // of 3 states with 2 delay lines
                                      // each
  filtertype yi;

  for(i=0; i<num_sections; i++)
    for(j=0; j<3; j++)
      states[i][j] = 0;

  for(i=0; i<num_samples; i++)
  {
    // Shift for correct input signal scaling
    filtertype xi = x[i] << inout_shift;

    for(b=0; b<num_sections; b++)
    {
      // Input scaling
      xi          = multfrac(xi, scales[b], scales_bitwidth);

      // Delay lines
      states[b][2] = states[b][1];
      states[b][1] = states[b][0];

      // Feedback
      states[b][0] = xi -
                     multfrac(states[b][1],
                              coeffs[b+4*num_sections], coeff_bitwidth) -
                     multfrac(states[b][2],
                              coeffs[b+5*num_sections], coeff_bitwidth);

      // Check coeff b1 to select filter type
      if(coeffs[b+1*num_sections] == 0)
      {
        // Butterworth feed-forward
        yi = states[b][0] + (states[b][1] * 2) + states[b][2];
      }
      else
      {
        // Elliptic feed-forward
        yi = states[b][0] +
             multfrac(states[b][1],
                      coeffs[b+1*num_sections], coeff_bitwidth) +
              states[b][2];
        }

        xi = yi; // Output of this SOS is input for the next one
      }

      // Shift for correct output signal scaling
      y[i] = yi >> inout_shift;
    }
}

// ---------------------------------------------------------------------------
// -- MEX interface function
// -- prhs[0]: Samples in
// -- prhs[1]: Coefficient matrix
// -- prhs[2]: Scale values
// -- prhs[3]: Bitwidth of state variables
// -- prhs[4]: Fractional portions of state variables
// -- prhs[5]: Bitwidth of in- and outputs
// -- prhs[6]: Shift (scaling) at filter in- and output
// -- prhs[7]: Bitwidth of scale values
// -- prhs[8]: Bitwidth of coefficients
// -- plhs[0]: Samples out
mexFunction(int nlhs, mxArray *plhs[ ], int nrhs, const mxArray *prhs[ ])
{
  mxArray *samples_in_m, *samples_out_m;
  mxArray *coeffs_m, *scales_m;
  filtertype *samples_in, *samples_out;
  filtertype *coeffs, *scales;
  const mwSize *dim_samples, *dim_coeffs, *dim_scales;
  int i, j;

  // Local copies of MATLAB inputs
  samples_in_m     = mxDuplicateArray(prhs[0]);
  coeffs_m         = mxDuplicateArray(prhs[1]);
  scales_m         = mxDuplicateArray(prhs[2]);
  bitwidth         = (int)mxGetScalar(prhs[3]);
  frac             = (int)mxGetScalar(prhs[4]);
  inout_bitwidth   = (int)mxGetScalar(prhs[5]);
  inout_shift      = (int)mxGetScalar(prhs[6]);
  scales_bitwidth = (int)mxGetScalar(prhs[7]);
  coeff_bitwidth   = (int)mxGetScalar(prhs[8]);

  // Check input dimensions
  dim_samples = mxGetDimensions(prhs[0]);
  if(dim_samples[1] != 1) {
    printf("Samples must be a column vector!\n");
    return;
  }
  int num_samples = dim_samples[0];
  printf("Number of samples: %d\n", num_samples);

  dim_coeffs = mxGetDimensions(prhs[1]);
  printf("SOS Matrix dimension: %dx%d\n", dim_coeffs[0], dim_coeffs[1]);
  if(dim_coeffs[1] != 6) {
    printf("SOS Matrix must have 6 columns!\n");
    return;
  }

  dim_scales = mxGetDimensions(prhs[2]);
  printf("Scales dimension: %dx%d\n", dim_scales[0], dim_scales[1]);
  if(dim_scales[1] != 1) {
    printf("Scales must be a column vector!\n");
    return;
  }

  if(dim_coeffs[0] != dim_scales[0]) {
    printf("Number of sections doesn't match number of scales!\n");
    return;
  }
  int num_sections = dim_coeffs[0];

  samples_out_m = plhs[0] = mxCreateNumericMatrix(dim_samples[0],
                                                   dim_samples[1],
                                                   mxINT64_CLASS, 0);

  // Fetch pointers to get access to data
  samples_in  = (filtertype*)mxGetPr(samples_in_m);
  coeffs      = (filtertype*)mxGetPr(coeffs_m);
  scales      = (filtertype*)mxGetPr(scales_m);
  samples_out = (filtertype*)mxGetPr(samples_out_m);

  // Apply filter
  printf("Filtering...\n");
  sosfilt(samples_in, samples_out, coeffs, scales, num_samples,
          num_sections);
}
```

Listing A.1: Second-order section fixed-point filtering using MEX

A.3.2 Bittrue CSD fixed-point multiplication

The multiplication can also be performed using CSD multipliers according to [35] as provided in the source code below.

```
filtertype multfrac_csd(filtertype a, filtertype csd_value,
                        filtertype csd_sign, const int oper_bitwidth,
                        const int oper_frac)
{
  int bit_index;
  int prev_bit_index = 0, index_diff;
  filtertype result, value;
  int negative, csd_negative = 0;
  filtertype max_value = pow(2, bitwidth)-1;

  // Align fractions of coefficient value and sign
  csd_value = csd_value * pow(2, frac-oper_frac);
  csd_sign  = csd_sign  * pow(2, frac-oper_frac);

  value  = a;
  result = a;

  // Extract sign information
  if(value == 0)      return (filtertype) 0;
  else if(value < 0) negative = 1;
  else                negative = 0;

  for(bit_index=1; bit_index<oper_bitwidth; bit_index++)
  {
    // Check if bit is set within the CSD value
    if(((csd_value >> bit_index) & 0x01) == 0x01)
    {
      // Compute difference to previous bit index
      index_diff       = bit_index - prev_bit_index;
      result          >>= index_diff;
      prev_bit_index   = bit_index;

      // Check the sign bit and add/subtract accordingly
      if(((csd_sign >> bit_index) & 0x01) == 0x01)
        result -= value;
      else
        result += value;
    }
  }

  // Check sign bit of coefficient
  if(((csd_sign >> (frac+1)) & 0x01) == 0x00)
  {
    index_diff = frac - prev_bit_index;
    result   >>= index_diff;
  }
  else
  {
    result <<= 1;
    csd_negative = 1;
  }

  // In case a negative result is expected, convertion to unsigned
       representation needs
  // to be performed
  if((csd_negative == 1 && negative == 0) || (csd_negative == 0 &&
       negative == 1))
    result = (filtertype)(result - max_value) % max_value;

  return result;
}
```

Listing A.2: Second-order section CSD fixed-point multiplication

A.3.3 Bittrue floating-point implementation

Utilizing the MPFR library an equivalent implementation of the SOS can be exhausted with floating-point arithmetic. Important to note is that this library is capable to change the mantissa and exponent to arbitrary values. The minimum and maximum exponent is set via `mpfr_set_emin()` and `mpfr_set_emax()` respectively. The precision is set whenever a MPFR variable is instantiated via `mpfr_init2()`. The handling of vectors/arrays of standard type `mpfr_t` is not recommended as some instability issues have been encountered during implementation. The implementation therefore keeps the in- and output samples in a `double*` and converts it to a `mpfr_t` sample by sample.

```
#include <mex.h>
#include <stdio.h>
#include <gmp.h>
```

```c
#include <mpfr.h>

// ---------------------------------------------------------------------------
// -- sosfilt
// -- Second order section filtering
void sosfilt(double* x, double* y, mpfr_t* sos, mpfr_t* scales,
             const int num_samples, const int num_biquads,
             const int prec)
{
  int i, j, b;
  char buf[80];
  mpfr_t states[num_biquads][3]; // One second order section consists of 3
                                 // states with 2 delay lines each
  mpfr_t tmp;
  mpfr_t xi, yi;

  mpfr_init2(tmp, prec);
  mpfr_init2(xi, prec);
  mpfr_init2(yi, prec);

  for(i=0; i<num_biquads; i++)
  {
    for(j=0; j<3; j++)
    {
      mpfr_init2(states[i][j], prec);
      mpfr_set_d(states[i][j], 0, GMP_RNDN);
    }
  }

  for(i=0; i<num_samples; i++)
  {
    mpfr_set_d(xi, x[i], GMP_RNDN);

    for(b=0; b<num_biquads; b++)
    {
      // Input scaling
      mpfr_mul(xi, xi, scales[b], GMP_RNDN);

      // Delay lines
      mpfr_set(states[b][2], states[b][1], GMP_RNDN);
      mpfr_set(states[b][1], states[b][0], GMP_RNDN);

      // Feedback
      // state1 * a1
      mpfr_mul(tmp, states[b][1], sos[b+4*num_biquads], GMP_RNDN);
      mpfr_sub(states[b][0], xi, tmp, GMP_RNDN);
      // state2 * a2
      mpfr_mul(tmp, states[b][2], sos[b+5*num_biquads], GMP_RNDN);
      mpfr_sub(states[b][0], states[b][0], tmp, GMP_RNDN);

      // Feed-forward
      // state0 * b0
      mpfr_mul(yi, states[b][0], sos[b+0*num_biquads], GMP_RNDN);
      // state1 * b1
      mpfr_mul(tmp, states[b][1], sos[b+1*num_biquads], GMP_RNDN);
      mpfr_add(yi, yi, tmp, GMP_RNDN);
      // state2 * b2
      mpfr_mul(tmp, states[b][2], sos[b+2*num_biquads], GMP_RNDN);
      mpfr_add(yi, yi, tmp, GMP_RNDN);

      // Output of this SOS is input for the next one
      mpfr_set(xi, yi, GMP_RNDN);
    }

    y[i] = mpfr_get_d(yi, GMP_RNDN);
  }

  // Free allocated MPFR variables
  mpfr_clear(**states);
  mpfr_clear(tmp);
  mpfr_clear(xi);
  mpfr_clear(yi);
}

// -----------------------------------------------------------------------
// -- MEX interface function
// -- prhs[0]: Samples in
// -- prhs[1]: SOS Matrix
// -- prhs[2]: Scale values
// -- prhs[3]: Precision
// -- plhs[0]: Samples out
mexFunction(int nlhs, mxArray *plhs[ ], int nrhs, const mxArray *prhs[ ])
{
  int i, j;
  mxArray *samples_in_m, *samples_out_m;
  mxArray *sos_m, *scales_m;
  double *samples_in, *samples_out;
  double *sos, *scales;
  const mwSize *dim_samples, *dim_sos, *dim_scales;
  int prec;

  if(nrhs != 4) {
    printf("Wrong number of input arguments!\n");
    return;
  }

  // Local copies of MATLAB inputs
  samples_in_m = mxDuplicateArray(prhs[0]);
  sos_m        = mxDuplicateArray(prhs[1]);
  scales_m     = mxDuplicateArray(prhs[2]);
  prec         = (int)mxGetScalar(prhs[3]);

  printf("Floating-point precision: %d\n", prec);

  // Check input dimensions
  dim_samples = mxGetDimensions(prhs[0]);
  if(dim_samples[1] != 1) {
    printf("Samples must be a column vector!\n");
    return;
  }
  int num_samples = dim_samples[0];
  printf("Number of samples: %d\n", num_samples);

  dim_sos = mxGetDimensions(prhs[1]);
  printf("SOS Matrix dimension: %dx%d\n", dim_sos[0], dim_sos[1]);
  if(dim_sos[1] != 6) {
    printf("SOS Matrix must have 6 columns!\n");
    return;
  }

  dim_scales = mxGetDimensions(prhs[2]);
  printf("Scales dimension: %dx%d\n", dim_scales[0], dim_scales[1]);
  if(dim_scales[1] != 1) {
    printf("Scales must be a column vector!\n");
    return;
  }

  if(dim_sos[0] != dim_scales[0]) {
    printf("Number of sections doesn't match number of scales!\n");
    return;
  }
  int num_biquads = dim_sos[0];

  samples_out_m = plhs[0] = mxCreateDoubleMatrix(dim_samples[0],
                                               dim_samples[1],
                                               mxREAL);

  // Fetch pointers to get access to data
  samples_in  = (double*)mxGetPr(samples_in_m);
  sos         = (double*)mxGetPr(sos_m);
  scales      = (double*)mxGetPr(scales_m);
  samples_out = (double*)mxGetPr(samples_out_m);

  // MPFR: Set default precision and exponent range
  mpfr_set_default_prec (prec);
  mpfr_set_emin(-7);
  mpfr_set_emax(7);

  mpfr_t sos_f[num_biquads*6];
  for(i=0; i<num_biquads*6; i++)
  {
    mpfr_init2(sos_f[i], prec);
    mpfr_set_d(sos_f[i], sos[i], GMP_RNDN);
  }
  mpfr_t scales_f[num_biquads];
  for(i=0; i<num_biquads; i++)
  {
    mpfr_init2(scales_f[i], prec);
    mpfr_set_d(scales_f[i], scales[i], GMP_RNDN);
  }

  // Apply filter
  printf("Filtering...\n");
  sosfilt(samples_in, samples_out, sos_f, scales_f, num_samples,
          num_biquads, prec);

  // Free allocated MPFR variables
  mpfr_clear(*sos_f);
  mpfr_clear(*scales_f);
}
```

Listing A.3: Second-order section floating-point filtering using MEX and MPFR library

A.4 MATLAB HDL Coder

A.4.1 MATLAB Filter Design HDL Coder

As the MATLAB FDA tool provides auto-generation of VHDL code, an example is provided here. The rather complex code needs detailed study for full understanding. This code is therefore not suitable for maintaining or even reusable in other IPs with different filter configurations. However, it is fully synthesizeable right away.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.numeric_std.ALL;
ENTITY sd_iirlp_core IS
   PORT( sys_clk                         :   IN    std_logic;
         sys_clk_en                      :   IN    std_logic;
         reset_n                         :   IN    std_logic;
         blk_en                          :   IN    std_logic;
         init                            :   IN    std_logic;
         data_in                         :   IN    signed(15 DOWNTO 0);
         data_out                        :   OUT   signed(15 DOWNTO 0)
         );

END sd_iirlp_core;


----------------------------------------------------------------
--Module Architecture: sd_iirlp_core
----------------------------------------------------------------
ARCHITECTURE rtl OF sd_iirlp_core IS
  -- Local Functions
  -- Type Definitions
  TYPE delay_pipeline_type IS ARRAY (NATURAL range <>) OF signed(15 DOWNTO 0);
  -- Constants
  CONSTANT scaleconst1                  : signed(15 DOWNTO 0) := to_signed(107, 16);
  CONSTANT coeff_b1_section1            : signed(15 DOWNTO 0) := to_signed(8192, 16);
  CONSTANT coeff_b2_section1            : signed(15 DOWNTO 0) := to_signed(16384, 16);
  CONSTANT coeff_b3_section1            : signed(15 DOWNTO 0) := to_signed(8192, 16);
  CONSTANT coeff_a2_section1            : signed(15 DOWNTO 0) := to_signed(-15024, 16);
  CONSTANT coeff_a3_section1            : signed(15 DOWNTO 0) := to_signed(7258, 16);
  CONSTANT scaleconst2                  : signed(15 DOWNTO 0) := to_signed(97, 16);
  CONSTANT coeff_b1_section2            : signed(15 DOWNTO 0) := to_signed(8192, 16);
  CONSTANT coeff_b2_section2            : signed(15 DOWNTO 0) := to_signed(16384, 16);
  CONSTANT coeff_b3_section2            : signed(15 DOWNTO 0) := to_signed(8192, 16);
  CONSTANT coeff_a2_section2            : signed(15 DOWNTO 0) := to_signed(-13674, 16);
  CONSTANT coeff_a3_section2            : signed(15 DOWNTO 0) := to_signed(5871, 16);
  CONSTANT scaleconst3                  : signed(15 DOWNTO 0) := to_signed(92, 16);
  CONSTANT coeff_b1_section3            : signed(15 DOWNTO 0) := to_signed(8192, 16);
  CONSTANT coeff_b2_section3            : signed(15 DOWNTO 0) := to_signed(16384, 16);
  CONSTANT coeff_b3_section3            : signed(15 DOWNTO 0) := to_signed(8192, 16);
  CONSTANT coeff_a2_section3            : signed(15 DOWNTO 0) := to_signed(-13000, 16);
  CONSTANT coeff_a3_section3            : signed(15 DOWNTO 0) := to_signed(5177, 16);
  -- Signals
  SIGNAL input_register                 : signed(15 DOWNTO 0);
  SIGNAL scale1                         : signed(34 DOWNTO 0);
  SIGNAL mul_temp                       : signed(31 DOWNTO 0);
  SIGNAL scaletypeconvert1              : signed(15 DOWNTO 0);
  -- Section 1 Signals
  SIGNAL a1sum1                         : signed(39 DOWNTO 0);
  SIGNAL a2sum1                         : signed(39 DOWNTO 0);
  SIGNAL b1sum1                         : signed(39 DOWNTO 0);
  SIGNAL b2sum1                         : signed(39 DOWNTO 0);
  SIGNAL typeconvert1                   : signed(15 DOWNTO 0);
  SIGNAL delay_section1                 : delay_pipeline_type(0 TO 1);
  SIGNAL inputconv1                     : signed(15 DOWNTO 0);
  SIGNAL a2mul1                         : signed(31 DOWNTO 0);
  SIGNAL a3mul1                         : signed(31 DOWNTO 0);
  SIGNAL b1mul1                         : signed(31 DOWNTO 0);
  SIGNAL b2mul1                         : signed(31 DOWNTO 0);
  SIGNAL b3mul1                         : signed(31 DOWNTO 0);
  SIGNAL sub_cast                       : signed(39 DOWNTO 0);
  SIGNAL sub_cast_1                     : signed(39 DOWNTO 0);
  SIGNAL sub_temp                       : signed(40 DOWNTO 0);
  SIGNAL sub_cast_2                     : signed(39 DOWNTO 0);
  SIGNAL sub_cast_3                     : signed(39 DOWNTO 0);
  SIGNAL sub_temp_1                         : signed(40 DOWNTO 0);
  SIGNAL b1multypeconvert1                  : signed(39 DOWNTO 0);
  SIGNAL add_cast                           : signed(39 DOWNTO 0);
  SIGNAL add_cast_1                         : signed(39 DOWNTO 0);
  SIGNAL add_temp                           : signed(40 DOWNTO 0);
  SIGNAL add_cast_2                         : signed(39 DOWNTO 0);
  SIGNAL add_cast_3                         : signed(39 DOWNTO 0);
  SIGNAL add_temp_1                         : signed(40 DOWNTO 0);
  SIGNAL section_result1                    : signed(15 DOWNTO 0);
  SIGNAL scale2                             : signed(34 DOWNTO 0);
  SIGNAL mul_temp_1                         : signed(31 DOWNTO 0);
  SIGNAL scaletypeconvert2                  : signed(15 DOWNTO 0);
  -- Section 2 Signals
  SIGNAL a1sum2                             : signed(39 DOWNTO 0);
  SIGNAL a2sum2                             : signed(39 DOWNTO 0);
  SIGNAL b1sum2                             : signed(39 DOWNTO 0);
  SIGNAL b2sum2                             : signed(39 DOWNTO 0);
  SIGNAL typeconvert2                       : signed(15 DOWNTO 0);
  SIGNAL delay_section2                     : delay_pipeline_type(0 TO 1);
  SIGNAL inputconv2                         : signed(15 DOWNTO 0);
  SIGNAL a2mul2                             : signed(31 DOWNTO 0);
  SIGNAL a3mul2                             : signed(31 DOWNTO 0);
  SIGNAL b1mul2                             : signed(31 DOWNTO 0);
  SIGNAL b2mul2                             : signed(31 DOWNTO 0);
  SIGNAL b3mul2                             : signed(31 DOWNTO 0);
  SIGNAL sub_cast_4                         : signed(39 DOWNTO 0);
  SIGNAL sub_cast_5                         : signed(39 DOWNTO 0);
  SIGNAL sub_temp_2                         : signed(40 DOWNTO 0);
  SIGNAL sub_cast_6                         : signed(39 DOWNTO 0);
  SIGNAL sub_cast_7                         : signed(39 DOWNTO 0);
  SIGNAL sub_temp_3                         : signed(40 DOWNTO 0);
  SIGNAL b1multypeconvert2                  : signed(39 DOWNTO 0);
  SIGNAL add_cast_4                         : signed(39 DOWNTO 0);
  SIGNAL add_cast_5                         : signed(39 DOWNTO 0);
  SIGNAL add_temp_2                         : signed(40 DOWNTO 0);
  SIGNAL add_cast_6                         : signed(39 DOWNTO 0);
  SIGNAL add_cast_7                         : signed(39 DOWNTO 0);
  SIGNAL add_temp_3                         : signed(40 DOWNTO 0);
  SIGNAL section_result2                    : signed(15 DOWNTO 0);
  SIGNAL scale3                             : signed(34 DOWNTO 0);
  SIGNAL mul_temp_2                         : signed(31 DOWNTO 0);
  SIGNAL scaletypeconvert3                  : signed(15 DOWNTO 0);
  -- Section 3 Signals
  SIGNAL a1sum3                             : signed(39 DOWNTO 0);
  SIGNAL a2sum3                             : signed(39 DOWNTO 0);
  SIGNAL b1sum3                             : signed(39 DOWNTO 0);
  SIGNAL b2sum3                             : signed(39 DOWNTO 0);
  SIGNAL typeconvert3                       : signed(15 DOWNTO 0);
  SIGNAL delay_section3                     : delay_pipeline_type(0 TO 1);
  SIGNAL inputconv3                         : signed(15 DOWNTO 0);
  SIGNAL a2mul3                             : signed(31 DOWNTO 0);
  SIGNAL a3mul3                             : signed(31 DOWNTO 0);
  SIGNAL b1mul3                             : signed(31 DOWNTO 0);
  SIGNAL b2mul3                             : signed(31 DOWNTO 0);
  SIGNAL b3mul3                             : signed(31 DOWNTO 0);
  SIGNAL sub_cast_8                         : signed(39 DOWNTO 0);
  SIGNAL sub_cast_9                         : signed(39 DOWNTO 0);
  SIGNAL sub_temp_4                         : signed(40 DOWNTO 0);
  SIGNAL sub_cast_10                        : signed(39 DOWNTO 0);
  SIGNAL sub_cast_11                        : signed(39 DOWNTO 0);
  SIGNAL sub_temp_5                         : signed(40 DOWNTO 0);
  SIGNAL b1multypeconvert3                  : signed(39 DOWNTO 0);
  SIGNAL add_cast_8                         : signed(39 DOWNTO 0);
  SIGNAL add_cast_9                         : signed(39 DOWNTO 0);
  SIGNAL add_temp_4                         : signed(40 DOWNTO 0);
  SIGNAL add_cast_10                        : signed(39 DOWNTO 0);
  SIGNAL add_cast_11                        : signed(39 DOWNTO 0);
  SIGNAL add_temp_5                         : signed(40 DOWNTO 0);
  SIGNAL output_typeconvert                 : signed(15 DOWNTO 0);
  SIGNAL output_register                    : signed(15 DOWNTO 0);


BEGIN

  -- Block Statements
  input_reg_process : PROCESS (sys_clk, reset_n)
  BEGIN
    IF reset_n = '0' THEN
      input_register <= (OTHERS => '0');
    ELSIF sys_clk'event AND sys_clk = '1' THEN
      IF sys_clk_en = '1' THEN
        input_register <= data_in;
      END IF;
    END IF;
  END PROCESS input_reg_process;

  mul_temp <= input_register * scaleconst1;
  scale1 <= resize(mul_temp, 35);

  scaletypeconvert1 <= resize(shift_right(scale1(34 DOWNTO 0) + ( "0" & (scale1(19) & NOT
          scale1(19) & NOT scale1(19) & NOT scale1(19) & NOT scale1(19) & NOT
          scale1(19) & NOT scale1(19) & NOT scale1(19) & NOT scale1(19) & NOT
          scale1(19) & NOT scale1(19) & NOT scale1(19) & NOT scale1(19) & NOT
          scale1(19) & NOT scale1(19) & NOT scale1(19))), 19), 16);

  --   ------------------ Section 1 ------------------

  typeconvert1 <= resize(shift_right(a1sum1(28 DOWNTO 0) + ( "0" & (a1sum1(13) & NOT a1sum1(13) &
          NOT a1sum1(13) & NOT a1sum1(13) & NOT a1sum1(13) & NOT a1sum1(13) & NOT a1sum1(13) & NOT
          a1sum1(13) & NOT a1sum1(13) & NOT a1sum1(13) & NOT a1sum1(13) & NOT a1sum1(13) & NOT
          a1sum1(13))), 13), 16);

  delay_process_section1 : PROCESS (sys_clk, reset_n)
  BEGIN
    IF reset_n = '0' THEN
      delay_section1 <= (OTHERS => (OTHERS => '0'));
    ELSIF sys_clk'event AND sys_clk = '1' THEN
      IF sys_clk_en = '1' THEN
        delay_section1(1) <= delay_section1(0);
        delay_section1(0) <= typeconvert1;
      END IF;
    END IF;
  END PROCESS delay_process_section1;

  inputconv1 <= scaletypeconvert1;

  a2mul1 <= delay_section1(0) * coeff_a2_section1;

  a3mul1 <= delay_section1(1) * coeff_a3_section1;

  b1mul1 <= resize(typeconvert1(15 DOWNTO 0) & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0'
          & '0' & '0' & '0' & '0', 32);

  b2mul1 <= resize(delay_section1(0)(15 DOWNTO 0) & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' &
          '0' & '0' & '0' & '0' & '0' & '0', 32);

  b3mul1 <= resize(delay_section1(1)(15 DOWNTO 0) & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' &
          '0' & '0' & '0' & '0' & '0' & '0', 32);

  sub_cast <= resize(inputconv1(15 DOWNTO 0) & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0'
          & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0', 40);
  sub_cast_1 <= resize(a2mul1, 40);
  sub_temp <= resize(sub_cast, 41) - resize(sub_cast_1, 41);
  a2sum1 <= sub_temp(39 DOWNTO 0);

  sub_cast_2 <= a2sum1;
  sub_cast_3 <= resize(a3mul1, 40);
  sub_temp_1 <= resize(sub_cast_2, 41) - resize(sub_cast_3, 41);
  a1sum1 <= sub_temp_1(39 DOWNTO 0);

  b1multypeconvert1 <= resize(b1mul1, 40);

  add_cast <= b1multypeconvert1;
  add_cast_1 <= resize(b2mul1, 40);
  add_temp <= resize(add_cast, 41) + resize(add_cast_1, 41);
  b2sum1 <= add_temp(39 DOWNTO 0);

  add_cast_2 <= b2sum1;
  add_cast_3 <= resize(b3mul1, 40);
  add_temp_1 <= resize(add_cast_2, 41) + resize(add_cast_3, 41);
  b1sum1 <= add_temp_1(39 DOWNTO 0);

  section_result1 <= resize(shift_right(b1sum1(33 DOWNTO 0) + ( "0" & (b1sum1(18) & NOT b1sum1(18)
          & NOT b1sum1(18) & NOT b1sum1(18) & NOT b1sum1(18) & NOT b1sum1(18) & NOT b1sum1(18) &
          NOT b1sum1(18) & NOT b1sum1(18) & NOT b1sum1(18) & NOT b1sum1(18) & NOT b1sum1(18) & NOT
          b1sum1(18) & NOT b1sum1(18) & NOT b1sum1(18) & NOT b1sum1(18) & NOT b1sum1(18) & NOT
          b1sum1(18))), 18), 16);

  mul_temp_1 <= section_result1 * scaleconst2;
  scale2 <= resize(mul_temp_1(31 DOWNTO 0) & '0' & '0' & '0', 35);

  scaletypeconvert2 <= resize(shift_right(scale2(34 DOWNTO 0) + ( "0" & (scale2(19) & NOT
          scale2(19) & NOT scale2(19) & NOT scale2(19) & NOT scale2(19) & NOT scale2(19) & NOT
          scale2(19) & NOT scale2(19) & NOT scale2(19) & NOT scale2(19) & NOT scale2(19) & NOT
          scale2(19) & NOT scale2(19) & NOT scale2(19) & NOT scale2(19) & NOT scale2(19) & NOT
          scale2(19) & NOT scale2(19) & NOT scale2(19))), 19), 16);

  --   ------------------ Section 2 ------------------
```

```

typeconvert2 <= resize(shift_right(a1sum2(28 DOWNTO 0) + ( "0" & (a1sum2(13) & NOT a1sum2(13) &
        NOT a1sum2(13) & NOT a1sum2(13) & NOT a1sum2(13) & NOT a1sum2(13) & NOT a1sum2(13) & NOT
        a1sum2(13) & NOT a1sum2(13) & NOT a1sum2(13) & NOT a1sum2(13) & NOT a1sum2(13) & NOT
        a1sum2(13))), 13), 16);

delay_process_section2 : PROCESS (sys_clk, reset_n)
BEGIN
  IF reset_n = ’0’ THEN
    delay_section2 <= (OTHERS => (OTHERS => ’0’));
  ELSIF sys_clk’event AND sys_clk = ’1’ THEN
    IF sys_clk_en = ’1’ THEN
      delay_section2(1) <= delay_section2(0);
      delay_section2(0) <= typeconvert2;
    END IF;
  END IF;
END PROCESS delay_process_section2;

inputconv2 <= scaletypeconvert2;

a2mul2 <= delay_section2(0) * coeff_a2_section2;

a3mul2 <= delay_section2(1) * coeff_a3_section2;

b1mul2 <= resize(typeconvert2(15 DOWNTO 0) & ’0’ & ’0’ & ’0’ & ’0’ & ’0’ & ’0’ & ’0’ & ’0’ & ’0’
        & ’0’ & ’0’ & ’0’ & ’0’, 32);

b2mul2 <= resize(delay_section2(0)(15 DOWNTO 0) & ’0’ & ’0’ & ’0’ & ’0’ & ’0’ & ’0’ & ’0’ & ’0’ &
        ’0’ & ’0’ & ’0’ & ’0’ & ’0’ & ’0’, 32);

b3mul2 <= resize(delay_section2(1)(15 DOWNTO 0) & ’0’ & ’0’ & ’0’ & ’0’ & ’0’ & ’0’ & ’0’ & ’0’ &
        ’0’ & ’0’ & ’0’ & ’0’ & ’0’, 32);

sub_cast_4 <= resize(inputconv2(15 DOWNTO 0) & ’0’ & ’0’ & ’0’ & ’0’ & ’0’ & ’0’ & ’0’ & ’0’ &
        ’0’ & ’0’ & ’0’ & ’0’ & ’0’ & ’0’ & ’0’ & ’0’ & ’0’ & ’0’ & ’0’ & ’0’ & ’0’, 40);
sub_cast_5 <= resize(a2mul2, 40);
sub_temp_2 <= resize(sub_cast_4, 41) - resize(sub_cast_5, 41);
a2sum2 <= sub_temp_2(39 DOWNTO 0);

sub_cast_6 <= a2sum2;
sub_cast_7 <= resize(a3mul2, 40);
sub_temp_3 <= resize(sub_cast_6, 41) - resize(sub_cast_7, 41);
a1sum2 <= sub_temp_3(39 DOWNTO 0);

b1multypeconvert2 <= resize(b1mul2, 40);

add_cast_4 <= b1multypeconvert2;
add_cast_5 <= resize(b2mul2, 40);
add_temp_2 <= resize(add_cast_4, 41) + resize(add_cast_5, 41);
b2sum2 <= add_temp_2(39 DOWNTO 0);

add_cast_6 <= b2sum2;
add_cast_7 <= resize(b3mul2, 40);
add_temp_3 <= resize(add_cast_6, 41) + resize(add_cast_7, 41);
b1sum2 <= add_temp_3(39 DOWNTO 0);

section_result2 <= resize(shift_right(b1sum2(33 DOWNTO 0) + ( "0" & (b1sum2(18) & NOT b1sum2(18)
        & NOT b1sum2(18) & NOT b1sum2(18) & NOT b1sum2(18) & NOT b1sum2(18) & NOT b1sum2(18) &
        NOT b1sum2(18) & NOT b1sum2(18) & NOT b1sum2(18) & NOT b1sum2(18) & NOT b1sum2(18) & NOT
        b1sum2(18) & NOT b1sum2(18) & NOT b1sum2(18) & NOT b1sum2(18) & NOT b1sum2(18) & NOT
        b1sum2(18))), 18), 16);

mul_temp_2 <= section_result2 * scaleconst3;
scale3 <= resize(mul_temp_2(31 DOWNTO 0) & ’0’ & ’0’ & ’0’, 35);

scaletypeconvert3 <= resize(shift_right(scale3(34 DOWNTO 0) + ( "0" & (scale3(19) & NOT
        scale3(19) & NOT scale3(19) & NOT scale3(19) & NOT scale3(19) & NOT scale3(19) & NOT
        scale3(19) & NOT scale3(19) & NOT scale3(19) & NOT scale3(19) & NOT scale3(19) & NOT
        scale3(19) & NOT scale3(19) & NOT scale3(19) & NOT scale3(19) & NOT scale3(19) & NOT
        scale3(19) & NOT scale3(19) & NOT scale3(19))), 19), 16);

--  ------------------ Section 3 ------------------

typeconvert3 <= resize(shift_right(a1sum3(28 DOWNTO 0) + ( "0" & (a1sum3(13) & NOT a1sum3(13) &
        NOT a1sum3(13) & NOT a1sum3(13) & NOT a1sum3(13) & NOT a1sum3(13) & NOT a1sum3(13) & NOT
        a1sum3(13) & NOT a1sum3(13) & NOT a1sum3(13) & NOT a1sum3(13) & NOT a1sum3(13) & NOT
        a1sum3(13))), 13), 16);

delay_process_section3 : PROCESS (sys_clk, reset_n)
BEGIN
  IF reset_n = ’0’ THEN
    delay_section3 <= (OTHERS => (OTHERS => ’0’));
  ELSIF sys_clk’event AND sys_clk = ’1’ THEN
    IF sys_clk_en = ’1’ THEN
```

```
        delay_section3(1) <= delay_section3(0);
        delay_section3(0) <= typeconvert3;
      END IF;
    END IF;
  END PROCESS delay_process_section3;

  inputconv3 <= scaletypeconvert3;

  a2mul3 <= delay_section3(0) * coeff_a2_section3;

  a3mul3 <= delay_section3(1) * coeff_a3_section3;

  b1mul3 <= resize(typeconvert3(15 DOWNTO 0) & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0'
        & '0' & '0' & '0' & '0', 32);

  b2mul3 <= resize(delay_section3(0)(15 DOWNTO 0) & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' &
        '0' & '0' & '0' & '0' & '0' & '0', 32);

  b3mul3 <= resize(delay_section3(1)(15 DOWNTO 0) & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' &
        '0' & '0' & '0' & '0' & '0', 32);

  sub_cast_8 <= resize(inputconv3(15 DOWNTO 0) & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' &
        '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0', 40);
  sub_cast_9 <= resize(a2mul3, 40);
  sub_temp_4 <= resize(sub_cast_8, 41) - resize(sub_cast_9, 41);
  a2sum3 <= sub_temp_4(39 DOWNTO 0);

  sub_cast_10 <= a2sum3;
  sub_cast_11 <= resize(a3mul3, 40);
  sub_temp_5 <= resize(sub_cast_10, 41) - resize(sub_cast_11, 41);
  a1sum3 <= sub_temp_5(39 DOWNTO 0);

  b1multypeconvert3 <= resize(b1mul3, 40);

  add_cast_8 <= b1multypeconvert3;
  add_cast_9 <= resize(b2mul3, 40);
  add_temp_4 <= resize(add_cast_8, 41) + resize(add_cast_9, 41);
  b2sum3 <= add_temp_4(39 DOWNTO 0);

  add_cast_10 <= b2sum3;
  add_cast_11 <= resize(b3mul3, 40);
  add_temp_5 <= resize(add_cast_10, 41) + resize(add_cast_11, 41);
  b1sum3 <= add_temp_5(39 DOWNTO 0);

  output_typeconvert <= resize(shift_right(b1sum3(30 DOWNTO 0) + ( "0" & (b1sum3(15) & NOT
        b1sum3(15) & NOT b1sum3(15) & NOT b1sum3(15) & NOT b1sum3(15) & NOT b1sum3(15) & NOT
        b1sum3(15) & NOT b1sum3(15) & NOT b1sum3(15) & NOT b1sum3(15) & NOT b1sum3(15) & NOT
        b1sum3(15) & NOT b1sum3(15) & NOT b1sum3(15) & NOT b1sum3(15))), 15), 16);

  Output_Register_process : PROCESS (sys_clk, reset_n)
  BEGIN
    IF reset_n = '0' THEN
      output_register <= (OTHERS => '0');
    ELSIF sys_clk'event AND sys_clk = '1' THEN
      IF sys_clk_en = '1' THEN
        output_register <= output_typeconvert;
      END IF;
    END IF;
  END PROCESS Output_Register_process;

  -- Assignment Statements
  data_out <= output_register;
END rtl;
```

Listing A.4: Auto-generated MATLAB filter design HDL coder output

A.4.2 MATLAB HDL Coder for Simulink

This section investigates a simple example how to generate HDL code out of a MATLAB Simulink model. Exemplarily an SOS filter is implemented. The model is not quantized to a certain bitwidth, thus the generated code contains signals of type `real`. Anyway, the intention of the example is to show the principle structure of the generated code.

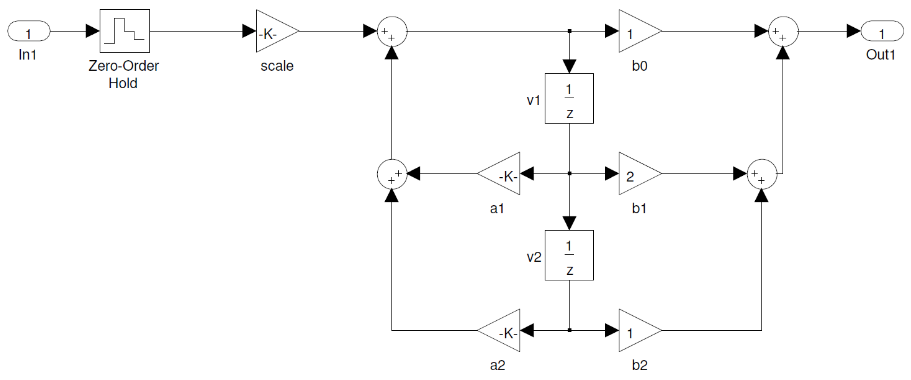


Figure A.2: Simulink model of a second-order section (SOS)

The automatically generated code by MATLAB looks as follows.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

ENTITY sos_model_double IS
  PORT( clk          :   IN    std_logic;
        reset        :   IN    std_logic;
        clk_enable   :   IN    std_logic;
        In1          :   IN    real;  -- double
        clk_en       :   OUT   std_logic;
        Out1         :   OUT   real  -- double
        );
END sos_model_double;


ARCHITECTURE rtl OF sos_model_double IS

  -- Signals
  SIGNAL enb          : std_logic;
  SIGNAL scale_out1   : real := 0.0;  -- double
  SIGNAL v1_out1      : real := 0.0;  -- double
  SIGNAL a1_out1      : real := 0.0;  -- double
  SIGNAL Sum_out1     : real := 0.0;  -- double
  SIGNAL v2_out1      : real := 0.0;  -- double
  SIGNAL a2_out1      : real := 0.0;  -- double
  SIGNAL Sum1_out1    : real := 0.0;  -- double
  SIGNAL b1_out1      : real := 0.0;  -- double
  SIGNAL b0_out1      : real := 0.0;  -- double
  SIGNAL b2_out1      : real := 0.0;  -- double
  SIGNAL Sum3_out1    : real := 0.0;  -- double
  SIGNAL Sum2_out1    : real := 0.0;  -- double

BEGIN
  scale_out1 <= 2.0 * In1;

  enb <= clk_enable;

  a1_out1 <= 2.0 * v1_out1;

  v1_process : PROCESS (clk, reset)
  BEGIN
    IF reset = '1' THEN
      v1_out1 <= 0.0;
    ELSIF clk'EVENT AND clk = '1' THEN
      IF enb = '1' THEN
        v1_out1 <= Sum_out1;
      END IF;
    END IF;
  END PROCESS v1_process;


  v2_process : PROCESS (clk, reset)
  BEGIN
    IF reset = '1' THEN
      v2_out1 <= 0.0;
    ELSIF clk'EVENT AND clk = '1' THEN
      IF enb = '1' THEN
        v2_out1 <= v1_out1;
      END IF;
    END IF;
  END PROCESS v2_process;


  a2_out1 <= 2.0 * v2_out1;

  Sum1_out1 <= a2_out1 + a1_out1;

  Sum_out1 <= scale_out1 + Sum1_out1;

  b1_out1 <= 2.0 * v1_out1;

  b0_out1 <= 2.0 * Sum_out1;

  b2_out1 <= 2.0 * v2_out1;

  Sum3_out1 <= b1_out1 + b2_out1;

  Sum2_out1 <= b0_out1 + Sum3_out1;

  Out1 <= Sum2_out1;

  clk_en <= clk_enable;

END rtl;
```

Listing A.5: MATLAB Simulink HDL Coder sample output

List of Abbreviations

ADC	Analog to Digital Converter
ASIC	Application Specific Standard Product
ASK	Amplitude Shift Keying
AWGN	Additive White Gaussian Noise
BER	Bit Error Rate
CIC	Cascaded-Integrator-Comb
CMOS	Complementary Metal Oxide Semiconductor
CSD	Canonic Signed Digit
DE	Differential Evolution
DSP	Digital Signal Processing
FIR	Finite Impulse Response
FSK	Frequency Shift Keying
GA	Genetic Algorithm
GL	Gate Level
HDL	Hardware Description Language
IIR	Infinite Impulse Response
IF	Intermediate Frequency
IP	Intellectual Property
ISI	Inter-Symbol Interference
LMS	Least-Mean Squares
LNA	Low Noise Amplifier
MMSE	Minimum-Mean Squared Error
MUT	Model Under Test
PDF	Probability Density Function
PQN	Pseudo Quantization Noise
PSD	Power Spectral Density
RTL	Register Transfer Level
SA	Simulated Annealing
SOS	Second-order section
VHDL	Very High speed integrated circuit Hardware Description language
ZF	Zero-Forcing

List of Figures

List of Tables

Bibliography

[1] Maxim Integrated. (2012) KeylessGo. [Online]. Available: http://www.maximintegrated.com/products/wireless/integrated-rf/keyless-go.cfm

[2] N. Karaboga, "Digital IIR filter design using differential evolution algorithm," *EURASIP J. Appl. Signal Process.*, pp. 1269–1276, Jan 2005. [Online]. Available: http://dx.doi.org/10.1155/ASP.2005.1269

[3] M. Quelhas and A. Petraglia, "On the design of IIR digital filter using linearized equation systems," in *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, June 2010, pp. 2702–2705.

[4] ——, "Digital filter design optimization using partial cost functions," in *Circuits and Systems, 2009. ISCAS 2009. IEEE International Symposium on*, May 2009, pp. 285–288.

[5] J.-J. Shyu and Y.-C. Lin, "Design of finite-wordlength iir digital filters in the time/spatial domain," in *Circuits and Systems, 1995. ISCAS '95., 1995 IEEE International Symposium on*, vol. 3, Apr 1995, pp. 2027–2030, vol.3.

[6] M. Valliappan, B. Evans, M. Gzara, M. Lutovac, and D. Tosic, "Joint optimization of multiple behavioral and implementation properties of digital iir filter designs," in *Circuits and Systems, 2000. Proceedings. ISCAS 2000 Geneva. The 2000 IEEE International Symposium on*, vol. 4, 2000, pp. 77–80, vol.4.

[7] A. Eletri, E. Salem, A. Zerek, and S. Elgandus, "Effect of coefficient quantization on the frequency response of an IIR digital filter by using software," in *Systems Signals and Devices (SSD), 2010 7th International Multi-Conference on*, Jun 2010, pp. 1–6.

[8] Y. Zhao, J. Zhou, X. Zhou, and G. Sobelman, "General lattice wave digital filter with phase compensation scheme," in *ASIC (ASICON), 2011 IEEE 9th International Conference on*, Oct 2011, pp. 220–223.

[9] C. Huang, Z. Xu, G. Li, H. Xu, and L. Chang, "An improved lattice IIR digital filter structure," in *Information, Communications and Signal Processing (ICICS) 2011 8th International Conference on*, Dec 2011, pp. 1–5.

[10] A. Oppenheim and R. Schafer, *Discrete-Time Signal Processing*, ser. Prentice Hall

signal processing series. Pearson Education, 2009.

[11] L. Milic and M. Lutovac, “Design of multiplierless elliptic IIR filters with a small quantization error,” *Signal Processing, IEEE Transactions on*, pp. 469–479, Feb 1999.

[12] T. Bose, Z. Zhang, O. Chauhan, and M. Radenkovic, “Design of multiplier-free state-space digital filters,” in *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*, May 2005, pp. 2591–2594, vol. 3.

[13] J. O. Coleman, “Fractions in the canonical-signed-digit number system,” in *in Conf. on Information Sciences and Systems*, 2001.

[14] M. Haseyama and D. Matsuura, “A filter coefficient quantization method with genetic algorithm, including simulated annealing,” *Signal Processing Letters, IEEE*, pp. 189–192, Apr 2006.

[15] Y. Yu and Y. Xinjie, “Cooperative coevolutionary genetic algorithm for digital IIR filter design,” *Industrial Electronics, IEEE Transactions on*, vol. 54, no. 3, pp. 1311–1318, Jun 2007.

[16] M. Nilsson, M. Dahl, and I. Claesson, “Digital filter design of IIR filters using real valued genetic algorithm,” in *Proceedings of the 2nd WSEAS International Conference on Electronics, Control and Signal Processing*, ser. ICECS’03. Stevens Point, Wisconsin, USA: World Scientific and Engineering Academy and Society (WSEAS), 2003, pp. 3:1–3:6. [Online]. Available: http://dl.acm.org/citation.cfm?id=1376214.1376217

[17] R. Storn, “Differential evolution design of an IIR-filter,” in *Evolutionary Computation, 1996., Proceedings of IEEE International Conference on*, May 1996, pp. 268–273.

[18] B. Luitel and G. Venayagamoorthy, “Differential evolution particle swarm optimization for digital filter design,” in *Evolutionary Computation, 2008. CEC 2008. (IEEE World Congress on Computational Intelligence). IEEE Congress on*, Jun 2008.

[19] K.-S. Tang, K.-F. Man, S. Kwong, and Z.-F. Liu, “Design and optimization of IIR filter structure using hierarchical genetic algorithms,” *Industrial Electronics, IEEE Transactions on*, pp. 481–487, Jun 1998.

[20] S. Hashemi and B. Nowrouzian, “A novel finite-wordlength particle swarm optimization technique for frm IIR digital filters,” in *Circuits and Systems (ISCAS), 2011 IEEE International Symposium on*, May 2011, pp. 2745–2748.

[21] C. Dai, W. Chen, and Y. Zhu, “Seeker optimization algorithm for digital IIR filter design,” *Industrial Electronics, IEEE Transactions on*, pp. 1710–1718, May 2010.

[22] R. Storn, “System design by constraint adaptation and differential evolution,” *Evolutionary Computation, IEEE Transactions on*, pp. 22–34, Apr 1999.

[23] S. Chattopadhyay and S. Sanyal, "The effect of gaussian pulse-shaping filter roll-off factor on the performance of QPSK modulated system," in *Signals, Circuits and Systems, 2009. ISSCS 2009. International Symposium on*, Jul 2009, pp. 1–4.

[24] J. Proakis, *Digital Communications*, ser. McGraw-Hill Series in Electrical and Computer Engineering. McGraw-Hill, 2001.

[25] G. Dolecek and F. Harris, "On design of two-stage CIC compensation filter," in *Industrial Electronics, 2009. ISIE 2009. IEEE International Symposium on*, Jul 2009, pp. 903–908.

[26] S. J. Fang, A. Bellaouar, S. T. Lee, and D. Allstot, "An image-rejection downconverter for low-IF receivers," *Microwave Theory and Techniques, IEEE Transactions on*, pp. 478–487, Feb 2005.

[27] Mathworks, *Filter Design Toolbox Users Guide.* The MathWorks, Inc., 2001.

[28] R. Storn, "Designing nonstandard filters with differential evolution," *Signal Processing Magazine, IEEE*, pp. 103–106, Jan 2005.

[29] M. Gilli and P. Winker, *Heuristic Optimization Methods in Econometrics, in Handbook of Computational Econometrics.* John Wiley & Sons, Ltd, Chichester, UK, 2009.

[30] S. Luke, *Essentials of Metaheuristics.* Lulu, 2009, available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.

[31] S. H. Yeung and K. F. Man, "Multiobjective optimization," *Microwave Magazine, IEEE*, pp. 120–133, Oct 2011.

[32] B. Widrow and I. Kollár, *Quantization Noise: Roundoff Error in Digital Computation, Signal Processing, Control, and Communications.* Cambridge University Press, 2008.

[33] B. Widrow, I. Kollar, and M.-C. Liu, "Statistical theory of quantization," *Instrumentation and Measurement, IEEE Transactions on*, pp. 353–361, Apr 1996.

[34] L. B. Jackson, "On the interaction of roundoff noise and dynamic range in digital filters," *Bell Syst. Techn. J. 49(2) (February, 1970)*, pp. 159–184, 1970.

[35] K. Venkat, *Efficient Multiplication and Division Using MSP430, Application Report, Texas Instruments*, Sep 2006.

[36] H. Kaeslin, *Digital Integrated Circuit Design. From VLSI Architectures to CMOS Fabrication.* Cambridge: Cambridge University Press, 2008.

[37] G. Moschytz and M. Hofbauer, *Adaptive Filter.* Springer, 2000.