Chapter 4.

Multipath Channel Simulators

Contents

Channel simulators are essential components for controlled and repeatable test-benches of communication systems, allowing researchers to work independently and compare results. Traditionally, channel simulators have been used for software simulations of receiver/transmitter chains. Nowadays, as systems become increasingly complex, hardware acceleration using rapid prototyping solutions become more and more appealing. This creates the need for hardware channel simulators, as part of hardware test benches on FPGA for example.

In simulations, channel models are most often used with discrete complex base-band signals. For simplification, the WSSUS assumption introduced in **Section 2.1** is used, where the fading processes are considered independent stochastic processes. Regardless of type, any channel profile must define the number of taps, their time delay relative to the first tap, and their average gain relative to the strongest tap (in dB). The tap gains can also be specified as the *rms* of their absolute values. For Rice channels, the Ricean factor K corresponding to the line-of-sight component need also be specified.

Regarding the Doppler fading spectrum of the taps, there are different characterizations, depending on the type of channel. Usually, land-mobile Rayleigh channel models assume a 'Jakes' spectrum for all taps, with the same maximum Doppler shift f_{Dmax}. For ionospheric HF channels, with a Gaussian Doppler spectrum, it is more common to specify the Doppler shift D_{sh} and Doppler spread D_{sp} separately for each tap, as seen in **Table 2.3**.

4.1. Fading Taps Generation

The most difficult task in a channel simulator is the generation of the fading taps. The taps are complex Gaussian processes having the required Doppler spectrum. So far, three approaches are known in the literature [72].

The first approach, called "sum of sinusoids", was proposed by Jakes [43] and consists in adding a number of sinusoids of equal amplitude and randomly distributed phases in order to generate a fading tap. The approach has been refined and extended to generate uncorrelated taps for multi-path channels [15]. Recent improvements include additional randomization, as in [75] and [102]. This method is computationally demanding due to the large number of calls to sin. An advantage, however, is the possibility of varying the Doppler spectrum continuously.

The second approach was introduced in [101] and consists in multiplying a set of independent complex Gaussian variables with a frequency mask equal with the square root of the desired power spectrum. The sequence is zero-padded and its inverse FFT is taken. The resulting sequence has the desired spectrum and is still Gaussian since the FFT operation is linear. The solution is computationally efficient but has the disadvantage that it is block-oriented in nature.

The third approach, which is also used in this thesis, consists in filtering a white Gaussian noise (WGN) by a filter with a frequency response equal with the square root of the Doppler spectrum. The WGN can be easily generate with a pseudo-random source. Since the WGN has a flat spectrum, the spectrum of the filtered signal will be exactly the transfer characteristic of the Doppler shaping filter.

We need to note that for a discrete channel of sampling period T and Doppler frequency f_D the spectrum is limited to a discrete frequency of $f_D T$. For most applications this frequency is very small. For instance, for a system in the 900 MHz band and a vehicle speed of 100 km/h, with a sampling rate of 1 Msps, the discrete Doppler rate is only 0.000083. The spectrum shaping filter

will have to be extremely narrow band, with a sharp cut-off and infinite stop-band attenuation.

These requirements cannot be satisfied with a FIR filter of practical length. Even an IIR filter, although it reduces the number of necessary taps, would be too hard to realize. The solution is to combine an IIR filter with a polyphase interpolator. The polyphase interpolation avoids all unnecessary multiplications with zero and can handle very large interpolation factors.

The complete block schematic of a complex fading tap generator is shown in **Figure 4.1**. The two WGN generators are independent, the Doppler filters are identical, and the interpolator is shared, that is, only its control logic. Between Doppler spectrum shaping and interpolation there is an additional block for introducing a Doppler shift in the spectrum. In the following, we will deal with these blocks in more detail.

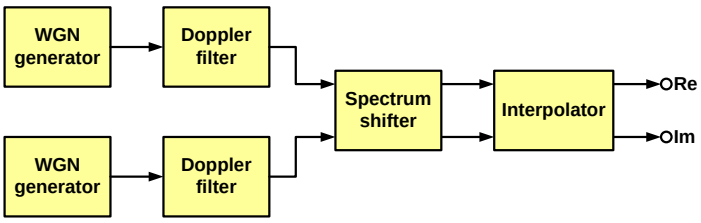


Figure 4.1.: Block schematic of a complex fading tap generator

4.2. White Gaussian Noise Generators

So far there has been little previous work in the area of digital HW generation of white Gaussian noise. Three representative publications, [25] [4] [52], address this issue and propose FPGA implementations.

The most popular methods for generating pseudo-random Gaussian noise in HW are:

1. The Box-Muller method [5]
2. The use of the central limit theorem [50]

4.2.1. The Box-Muller Method

This method produces two independent Gaussian variables by transforming two independent variables u_1 and u_2 with uniform distribution over the interval $[0, 1)$. First, a set of intermediate

functions are computed:

$$f(u_1) = \sqrt{-\ln(u_1)} \tag{4.1}$$
$$g_1(u_2) = \sqrt{2}\sin(2\pi u_2) \tag{4.2}$$
$$g_2(u_2) = \sqrt{2}\cos(2\pi u_2) \tag{4.3}$$

The products x_1 and x_2 will have Gaussian distributions with zero mean and $\sigma = 1$.

$$x_1 = f(u_1)g(u_2) \tag{4.4}$$
$$x_2 = f(u_2)g(u_2) \tag{4.5}$$

The HW implementation of the these equations is straightforward, as shown in **Figure 4.2**. The challenge is how to implement the square root of the logarithm and the sine/cosine efficiently. An efficient solution is to employ hybrid look-up tables with non-uniform segmentation, as proposed in [25] and [52]. Nevertheless, the resulting circuit size is still large and it is not easy to make the design generic and scalable.

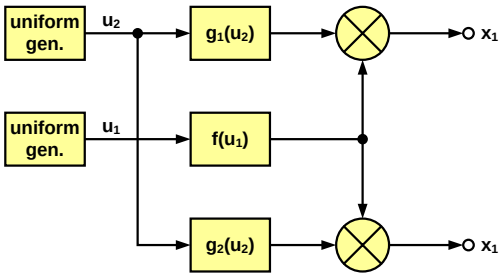


Figure 4.2.: Gaussian random generator using the Box-Muller method

4.2.2. The Use of the Central Limit Theorem

The central limit theorem offers a very simple way to generate random Gaussian noise. According to the central limit theorem, the distribution of a sum of independent random variables with arbitrary distributions converges to a normal distribution as the number of variables increases.

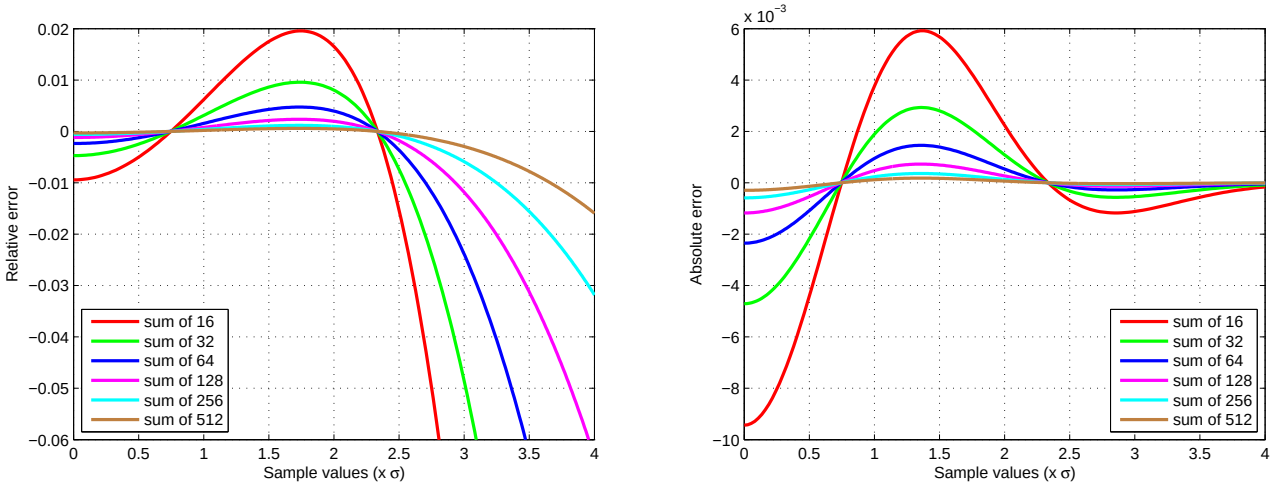
If N independent variables have equal distributions with mean μ and standard deviation σ, the distribution of their sum will have $\mu_N = \mu N$ and $\sigma_N = \sigma\sqrt{N}$. Intuitively, the probability density function (pdf) of the sum of two independent variables can be obtained as the convolution of their pdf's.

In order to gain an estimate of the complexity, we proceed to determine how many uniformly distributed variables need to be summed to achieve a Gaussian distribution with a given accu-

racy. The accuracy is given by the relative error $\xi(x)$ of the resulting distribution $X(x)$ against an ideal Gaussian distribution $N(x)$ with the same σ.

$$\xi(x) = \frac{X(x) - N(x)}{N(x)} \tag{4.6}$$

We also determine the absolute error between $X(x)$ and $N(x)$, considering $N(0)$ as normalized reference level. The results are shown in **Figure 4.3**.



(a) Relative error

(b) Absolute error

Figure 4.3.: Error of a Gaussian distribution obtained from uniform distributions

These results show that in order to achieve a good accuracy, especially at high sample values, we need to sum a very large number of uniformly distributed random variables. For example, in order to get 1% accuracy at 4σ more than 512 random variables have to be summed. At a first glance, this might seem a very tough design problem. However, by finding efficient solutions for generating uniform distributions, the complexity can be significantly reduced.

For digital generators, the most straightforward solution is to sum N discrete variables with uniform distribution. In the case of a uniform distribution, the variance σ^2 has the following expression, where M is the number of discrete values in the distribution:

$$\sigma_u^2 = \frac{M^2 - 1}{12} \tag{4.7}$$

If the uniform variable has a width of B bits, the mean μ_u and the standard deviation σ_u have

Bits	1	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}
Width	1	2	3	4	5	6	7	8	9	10	11
Range	$0 \dots 1$	$0 \dots 2^1$	$0 \dots 2^2$	$0 \dots 2^3$	$0 \dots 2^4$	$0 \dots 2^5$	$0 \dots 2^6$	$0 \dots 2^7$	$0 \dots 2^8$	$0 \dots 2^9$	$0 \dots 2^{10}$
Mean	$1/2$	2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9
Stdev	$1/2$	$1/\sqrt{2}$	2^0	$2^0\sqrt{2}$	2^1	$2^1\sqrt{2}$	2^2	$2^2\sqrt{2}$	2^3	$2^3\sqrt{2}$	2^4

Table 4.1.: Properties of the sum of 2^N random binary variables

the following expressions:

$$\mu_u = \frac{2^B - 1}{2} \tag{4.8}$$

$$\sigma_u = \sqrt{\frac{2^{2B} - 1}{12}} \tag{4.9}$$

The challenge is how to design a multi-bit uniform generator and how to sum so many values, e.g. 256 for a decent accuracy. The complexity can minimized if we sum 1-bit variables. The number of required adders remains the same, but their size will be much smaller. The problem consists now in how to generate so many independent binary variables in parallel.

Summing 1-bit uniform variables has also the advantage that both the mean and the standard deviation have simple expressions, which simplifies the hardware implementation. We hereafter denote by 2^N the number of variables to be summed, always a power of two. **Table 4.1** shows various properties of the resulting sum: bitwidth, range, mean μ, and standard deviation σ.

These results have been obtained by treating the binary variables as unsigned numbers and summing them accordingly. This results in a non-zero mean. When summing binary variables the mean is always a power of two, which is convenient for hardware implementations. In most practical applications, however, a zero-mean Gaussian distribution is desired. This can be achieved either by subtracting the known mean from the sum or by simply summing the binary variables as signed numbers. The latter requires no hardware and is obviously the preferred method. **Figure 4.4** shows the resulting distribution (histogram) when summing 2^6 binary variables, both as signed and as unsigned. The horizontal axis shows the full range of a 4-bit number. The mean and the limits of the resulting sum are also shown on the horizontal axis. Sigma is 2^2 in both cases.

From both **Table 4.1** and **Figure 4.4** it can be observed that the range of the sum is nearly half of the full range. Another useful observation that can be derived from **Table 4.1** is that the relative sigma, i.e. relative to the full range, decreases by $\sqrt{2}$ for every doubling of the number of binary variables. **Figure 4.5** shows the histogram when summing 2^6 and 2^8 variables. The horizontal axis is zoomed to half of the sum range for clarity. When increasing the number of variables 4 times, the samples range also increases 4 times, whereas absolute σ increases only 2 times, which is equivalent to a decrease of relative σ by 2.

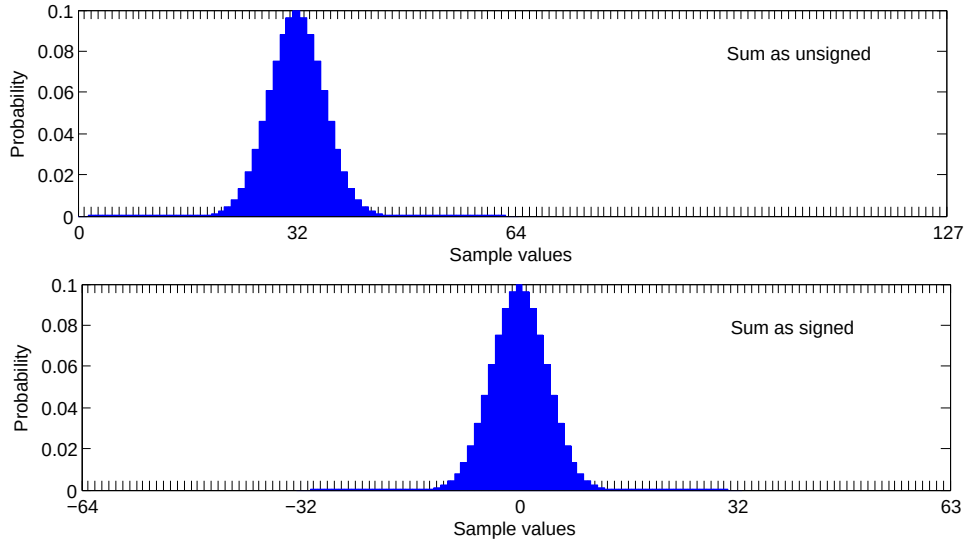


Figure 4.4.: Histogram of the sum of 64 (2^6) independent binary variables

The more variables are summed, the narrower the range of interest becomes relatively to the full range. The range of interest is specified as a multiple of sigma, usually a power of 2. Typical cases are $-4\sigma \ldots 4\sigma$ and $-8\sigma \ldots 8\sigma$, depending on the application. This allows for reducing the number of bits of the generated Gaussian noise to accommodate just the range of interest. The operation consists simply in discarding a number of MSB's. For example, when summing 2^8 binary variables the full range is $-32\sigma \ldots 32\sigma$. Reducing the range to $-8\sigma \ldots 8\sigma$ is performed by discarding two MSB's. These insights will be used later when designing the HW generator.

4.2.3. The LFSR as Random Number Generator

There are two types of LFSR configurations. The Galois configuration has XOR gates (modulo 2 adders) in the shift register path (**Figure 4.6a**) and is also referred to as in-line, internal, or modular type. The Fibonacci configuration has XOR gates in the feedback path (**Figure 4.6b**) and is also named out-of-line, external, or simple type. The Galois implementation is better suited for HW implementation because the critical path always contains only one XOR gate, regardless of the generator polynomial.

The binary polynomial $P(x)$ is referred to as generator polynomial. When this polynomial is primitive, the generated pseudo-random binary sequence has maximum length $2^L - 1$. A primitive polynomial is a polynomial that generates all elements of an extension field from a base field, in our case GF(2). All primitive polynomials are also irreducible, i.e. they cannot be factored into nontrivial polynomials over the same field, but not all irreducible polynomials are primitive. **Table A.1** lists primitive polynomials for LFSR length L between 2 and 64.

A LFSR can be used to generate multiple binary sequences in parallel, by generating more than

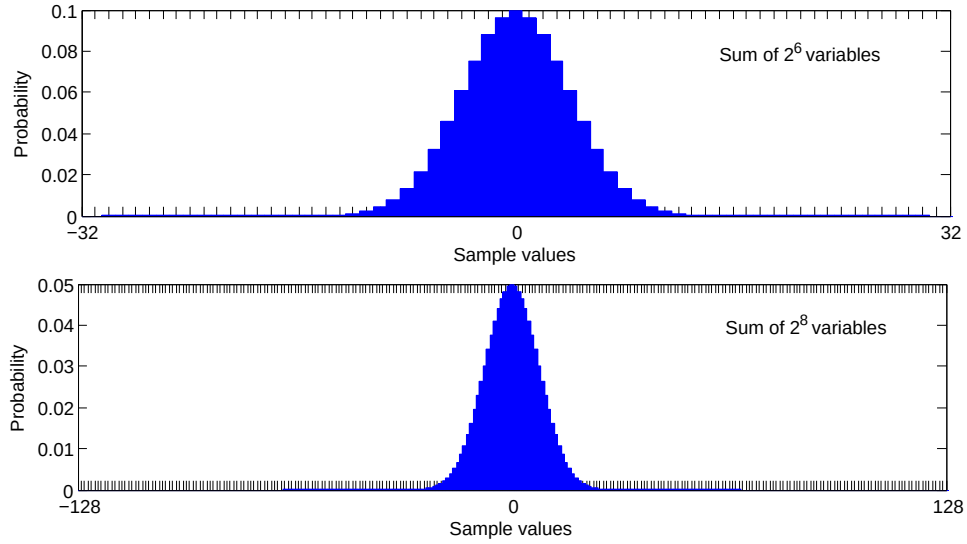
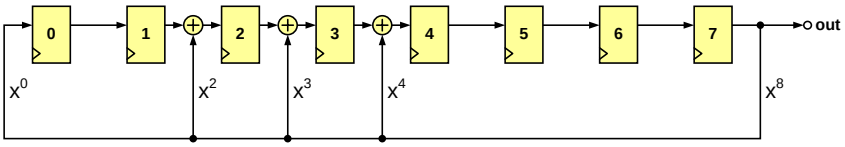
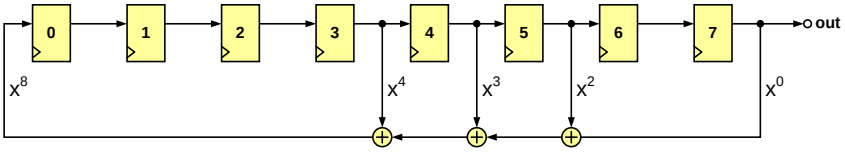


Figure 4.5.: Decrease of relative sigma with the number of binary variables



(a) Galois configuration



(b) Fibonacci configuration

Figure 4.6.: LFSR configuration types, example $P(x) = x^8 + x^4 + x^3 + x^2 + 1$

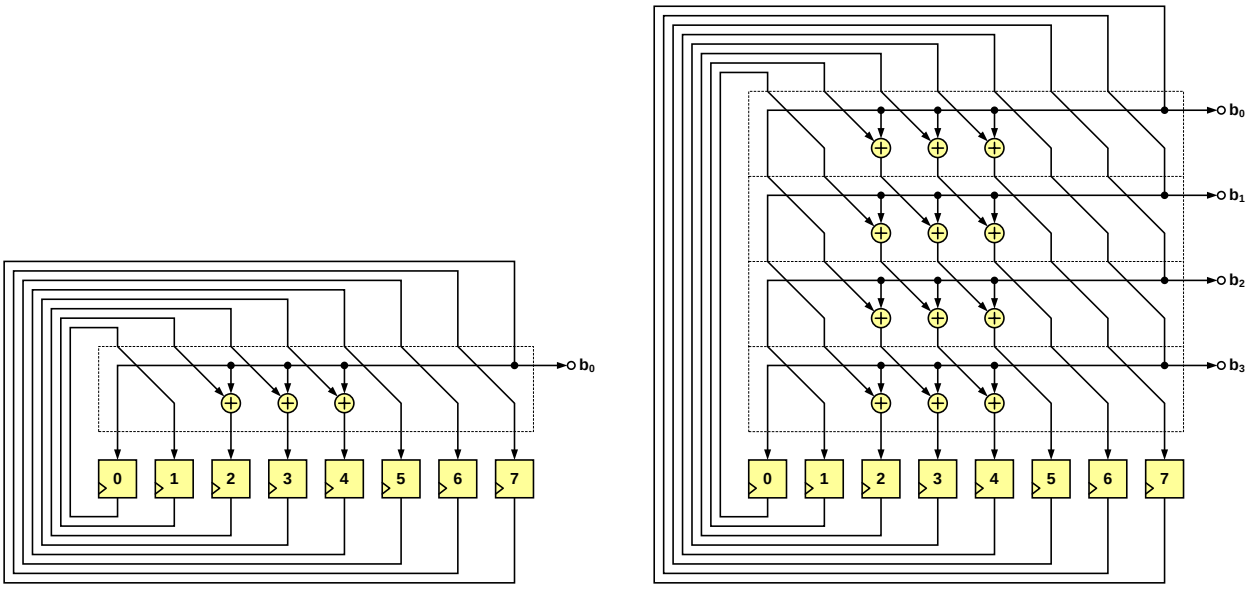
one bit of the pseudo-random sequence every clock cycle, say M. If the number of outputs M is a power of two, the period is preserved. The M streams will be identical, of period $2^N - 1$, and any two adjacent outputs will have a phase shift of $2^N/M$. This is shown in **Figure 4.7** for a LFSR of length 5 which generates 4 consecutive bits in parallel.

In order to determine how to generate more bits of the sequence in parallel, we redraw the schematic of the sequential Galois LFSR so that we emphasize the initial and final states of the registers (**Figure 4.8a**). It can be now easily seen that the LFSR consists of a state register and a combinational block which computes the next state. In order to generate M bits in parallel M such combinational blocks have to be cascaded, as in **Figure 4.8b**.

The M parallel binary sequences are independent and can be added as a first step in obtaining a Gaussian distribution. Due to the cross-correlation between the sequences, the auto-correlation of the sum will have a peak at lag $2^N/M$, thus reducing the effective period of the sum sequence.

Output 0:	b_0	b_4	b_8	b_{12}	b_{16}	b_{20}	b_{24}	b_{28}	b_1	b_5	b_9	b_{13}	b_{17}	b_{21}	b_{25}	b_{29}	b_2	b_6	b_{10}	b_{14}	b_{18}	b_{22}	b_{26}	b_{30}	b_3	b_7	b_{11}	b_{15}	b_{19}	b_{23}	b_{27}
Output 1:	b_1	b_5	b_9	b_{13}	b_{17}	b_{21}	b_{25}	b_{29}	b_2	b_6	b_{10}	b_{14}	b_{18}	b_{22}	b_{26}	b_{30}	b_3	b_7	b_{11}	b_{15}	b_{19}	b_{23}	b_{27}	b_0	b_4	b_8	b_{12}	b_{16}	b_{20}	b_{24}	b_{28}
Output 2:	b_2	b_6	b_{10}	b_{14}	b_{18}	b_{22}	b_{26}	b_{30}	b_3	b_7	b_{11}	b_{15}	b_{19}	b_{23}	b_{27}	b_0	b_4	b_8	b_{12}	b_{16}	b_{20}	b_{24}	b_{28}	b_1	b_5	b_9	b_{13}	b_{17}	b_{21}	b_{25}	b_{29}
Output 3:	b_3	b_7	b_{11}	b_{15}	b_{19}	b_{23}	b_{27}	b_0	b_4	b_8	b_{12}	b_{16}	b_{20}	b_{24}	b_{28}	b_1	b_5	b_9	b_{13}	b_{17}	b_{21}	b_{25}	b_{29}	b_2	b_6	b_{10}	b_{14}	b_{18}	b_{22}	b_{26}	b_{30}

Figure 4.7.: Generation of multiple bits in parallel with LFSR



(a) Sequential Galois LFSR (redrawn) (b) Parallel LFSR (4 bits)

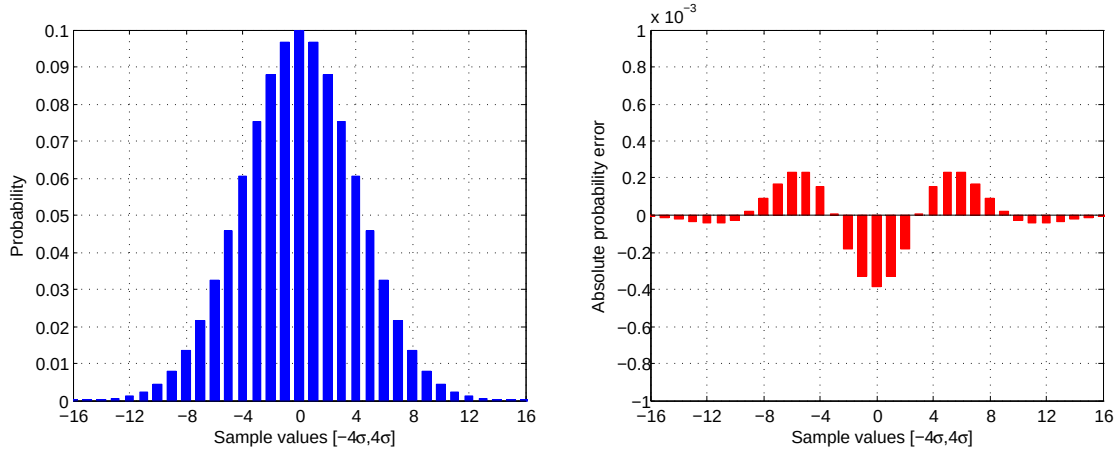
Figure 4.8.: Generating more pseudo-random bits in parallel

The number of parallel outputs that can be generated with a single LFSR is limited since it directly affects the maximum frequency. Their number should not exceed 32.

In the following we want to investigate the statistical properties of the Gaussian noise obtained by summing M consecutive bits generates by the same LFSR of length L. As concrete case we consider $M = 64$. The resulting Gaussian samples range from -32 to 32, with a $\sigma = 4$. **Figure 4.9** shows the discrete distribution and its absolute error compared to an ideal Gaussian, for the noise obtained by summing 64 independent binary variables. The absolute error is consistent with the profile in **Figure 4.3b**.

Contrary to our expectations, when summing binary sequences generated by the same LFSR, the resulting distribution will not be exactly the expected one. The reason is the existing correlation between the generated binary sequences, which originate from the same generator polynomial. The resulting distributions have a very small ($\ll 1$) mean and a larger absolute error compared to the ideal case in **Figure 4.9b**. The error depends mainly on the number of taps of the LFSR and very less on their position or the LFSR length.

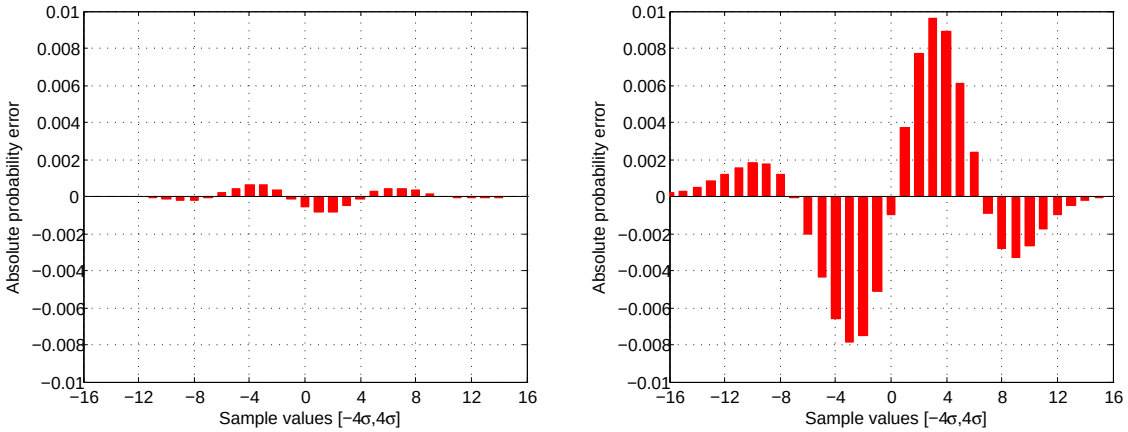
The absolute errors for a 3-tap and a 1-tap LFSR are shown in **Figure 4.10**. The generator



(a) Probability distribution (b) Absolute error

Figure 4.9.: Probability distribution and its absolute error for the noise obtained by summing 64 independent binary variables

polynomials are taken from **Table A.1**. Only 1-tap and 3-tap irreducible polynomials are considered because for all lengths up to 256 there exists at least one such polynomial. The only notable exception occurs at length 37, for which no 1-tap or 3-tap irreducible polynomial exists, the first one having 5 taps. Although we only give the results for two LFSRs, simulations for LFSR lengths up to 64 show that all LFSR with the same number of taps produce almost the same error profile, with a deviation of less than 20%. The error profiles have been obtained by analyzing 2^30 (approx. 1 billion) generated Gaussian samples.

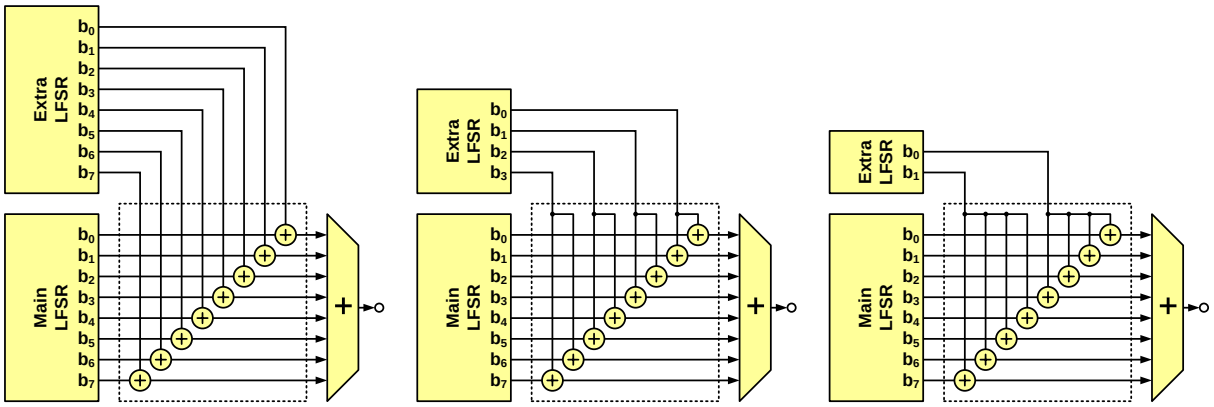


(a) LFSR 32, 3 taps (b) LFSR 33, 1 tap

Figure 4.10.: Probability distribution errors for noise obtained by summing 64 parallel outputs of an LFSR

For the 1-tap LFSR the absolute error peak is around 0.01, whereas for the 3-tap LFSR the error is much smaller, with an error peak around 0.001, still larger than the ideal case in

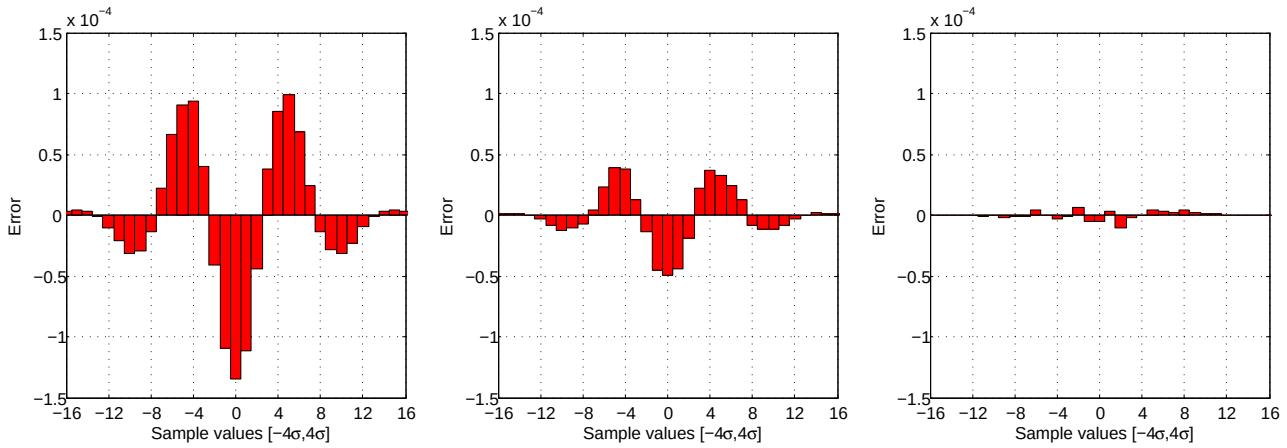
Figure 4.9b. The error of the resulting distribution can be reduced by introducing additional randomness, which reduces the correlation. Our proposal consists in XOR-ing the generated bits by another pseudo-random sequence generated by an additional LFSR. The number of generated bits XOR-ed with the same bit of the extra sequence is a parameter of the solution, and should be a power-of-2. We call this parameter randomization step. **Figure 4.11** shows the configurations for three different randomization steps: 2^0, 2^1, 2^2.



(a) Step = 2^0 (b) Step = 2^1 (c) Step = 2^2

Figure 4.11.: Schematics for randomization with different steps

Simulations show that by applying randomization as proposed above, even with large steps, results in a distribution with zero mean. As expected, the smaller the randomization step, the lower the error. **Figure 4.12** shows the error of the resulting histogram in the case of 3-tap LFSR's, when using randomization with steps 2^4, 2^2, and 2^0 respectively. The reference is the histogram of the sum of 2^6 independent binary variables, shown in **Figure 4.9b**.



(a) Step = 2^4 (b) Step = 2^2 (c) Step = 2^0

Figure 4.12.: Error reduction when decreasing the randomization step

The conclusion is that when using extra randomization with step 1, i.e. each generated bit is XOR-ed with a bit of the extra sequence, the error is reduced to zero. The residual error in

Figure 4.12c is actually caused by the limited number of samples (2^{32}) used in the simulation. Even for a randomization step of 2^4, the error is still significantly below the error compared to the ideal Gaussian distribution. Moreover, simulations have shown that the relative error reduction depends on the randomization step alone, regardless of the error with no randomization. For full randomization, i.e. with step 1, any existing correlation between the parallel bitstreams that are summed is destroyed, therefore it does not matter wether the generator polynomial has 1 or 3 taps.

4.2.4. Generic Architecture for WGN Generation

A good Gaussian distribution requires the summation of a large number of binary variables, e.g. 256. Generating so many bits in parallel with a single LFSR is not practically feasible. The solution is to employ more than one LFSR. For 8 LFSR's with 32 outputs each we have already 256 binary variables to sum. It is essential for the LFSR's to be of different lengths in order to maximize the period of the resulting sequence. If the periods of the individual sequences are relatively prime, the period of the resulting sequence is the product of the individual periods.

Figure 4.13 shows an example configuration that illustrates the use of multiple LFSR's. This example employs 4 LFSR's with 8 outputs each, much less than required by a practical design. The bitwidth of the result depends on the number of binary variables, as indicated in **Table 4.1**. The result becomes one bit wider with each addition. If we sum 2^N binary sequences the bitwidth of result will be $N + 1$. The adder for 2^N binary variables is implemented using a symmetrical tree structure, as shown in **Figure 4.13**. Each stage, their width increases by one starting from 1 bit and their number is reduced by two starting from 2^{N-1}. If the adders are implemented as ripple carry, as it is usually the case in FPGA, the number of elementary 1-bit adders is given by the equation below. For 2^8 1-bit inputs, we need for example 247 elementary adders.

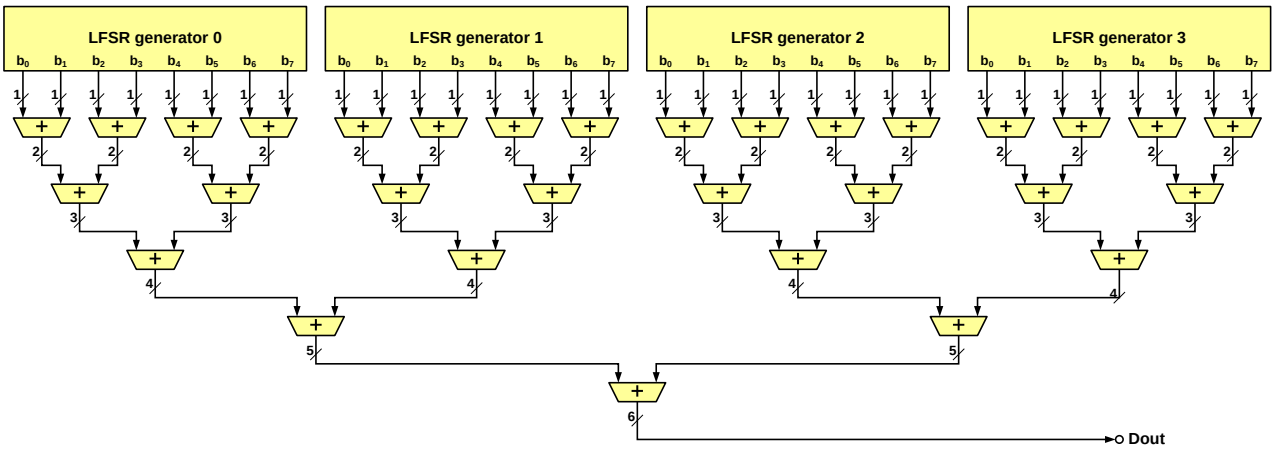


Figure 4.13.: Gaussian noise generation by summing pseudo-random binary sequences

The main properties of an adder tree for 2^N binary variables are summarized below as a function

# bits	2^4	2^5	2^6	2^7	2^8
# elementary adders (total)	22	52	114	240	494
# elementary adders (critical path)	6	10	15	21	28

Table 4.2.: Some properties of an adder tree for 2^N bits

of N. We have assumed that an N-bit adder consists of N 1-bit elementary adders.

- Number of adders: $2^N - 1$
- Number of adders in the critical path: N
- Number of elementary adders: $\sum_{k=1}^{N-1} k \cdot 2^{N-k}$
- Number of elementary adders in the critical path: $\sum_{k=1}^{N-1} k = n(n-1)/2$

Table 4.2 shows the number of elementary adders (total and in the critical path) for N ranging from 4 to 8. For $N = 8$ the depth reaches 28, which limits the operating frequency drastically. Fortunately, the adder tree can be pipelined since there are no loops involved.

The regular structure of the adder tree makes the realization of a scalable design possible. As a proof of concept we have created a completely generic adder tree design in VHDL. The parameters are the number and the width of the input operands, and wether they are treated as signed or unsigned. The number of inputs is not constrained to a power of two. Moreover, the pipelining is configurable through an additional generic parameter that indicates the number of combinational adder stages between two registers, starting from the output. If this parameter is zero, the resulting adder tree is purely combinational. The circuit relies on the recursive instantiation feature in VHDL, which is currently supported by most of the synthesis tools.

In some applications, Gaussian noise samples need not be generated every clock cycle. An example is the white noise generator for the fading taps, where the sample rate before the interpolator is very low because of the very low Doppler spreads. This situation can be exploited to reduce the number of adders for the same number of bits to be summed or to sum more bits with the same adder. The solution is to use an accumulator at the output to sum consecutive samples. If we need to generate a sample every 4 clock cycles, the number of bits to be generated and summed in a clock cycle is reduced by 4. Now instead of 2^8 bits we will only need to generate 2^6 in parallel. According to **Table 4.2**, the total number of elementary adders decreases from 494 to only 114 while the number of adders in the critical path is reduced from 28 to 15, which saves area and increases the maximum frequency.

Figure 4.15 shows the distribution of the noise generated with the architecture presented above, that is, by summing 256 binary variables. In this figure, the probability is normalized to the probability of zero and the range is limited between -4σ and 4σ. The distribution is perfectly smooth, unlike the one obtained in [25] and [52] using the Box-Muller method.

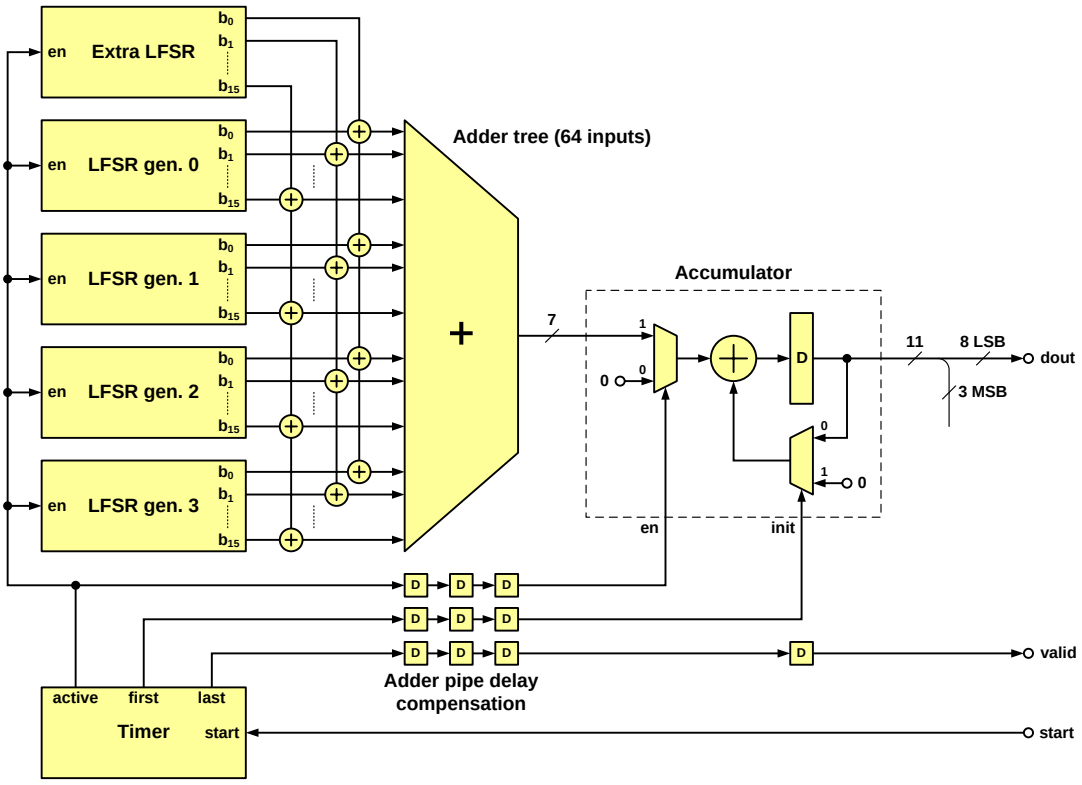


Figure 4.14.: Schematic of a Gaussian noise generator with phase accumulator

FPGA Resource	Sequential factor			Total in device
	4	8	16	
# Slices	283	189	114	960
# Slice FF's	448	311	244	1920
# 4-input LUT's	499	312	238	1920

Table 4.3.: Synthesis results for a Xilinx Spartan3E XC3S100E-5 FPGA

The proposed architecture is also completely scalable, unlike those proposed in [25] and [52], allowing to generate Gaussian noise by summing any power-of-2 number of binary variables. Moreover, it also allows to trade off throughput for precision, which is very desirable for fading tap generators, where data rates are very low before interpolation. **Table 4.3** shows the synthesis results for a Gaussian generator that sums 256 binary variables. This implementation uses 4 LFSR's with lengths 32, 33, 34, and 35. Synthesis results are shown for sequential factors of 4, 8, and 16. The sequential factor indicates the number of clock cycles needed to generate a noise sample. Lower sequential factors denote higher parallelism, which explains why they require more FPGA resources. The reported speed after synthesis only, for a grade-5 device, is about 190 MHz in all three cases.

Compared to the results in [52] our architecture requires far fewer resources, in addition to its excellent scalability. For sake of comparison, we have also synthesized a parallel configuration

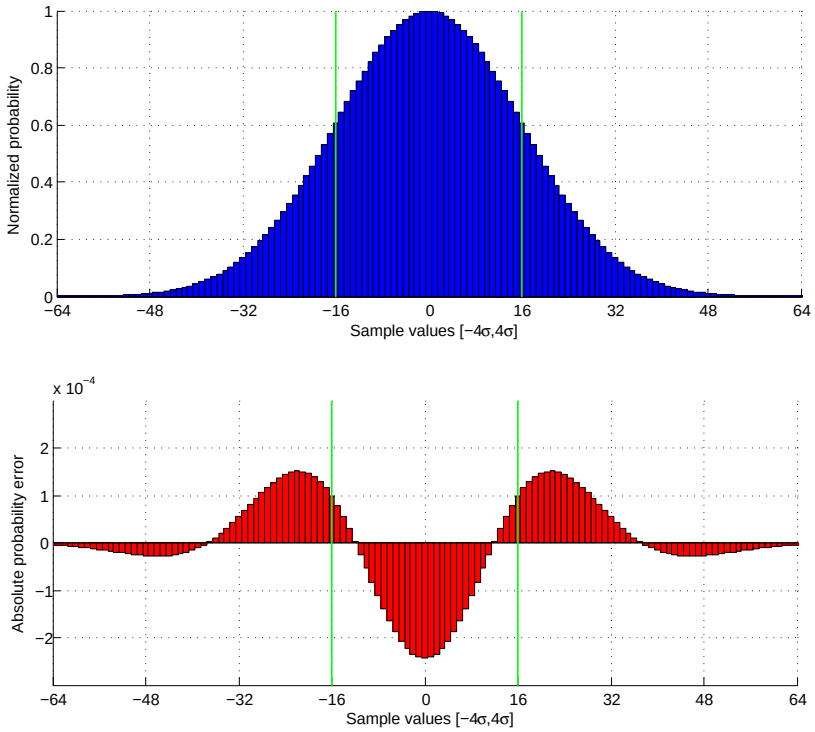


Figure 4.15.: Normalized probability and the absolute error when summing 256 binary variables

with roughly the same performance. This configuration generates a sample each clock cycle and has 8 LFSR's with increasing lengths starting from 32, each with 32 parallel outputs, thus generating 256 random bits in parallel. The used FPGA is Xilinx Virtex2 XC2V4000-6. Out of 23040 slices available, only 880 are used, that is, about 3.8%. The maximum frequency reported after synthesis was 180 MHz. This compares very favorably with the reference implementation, which takes up 10% of the same FPGA and runs at 133 MHz post synthesis.

4.3. Spectrum-Shaping Filter

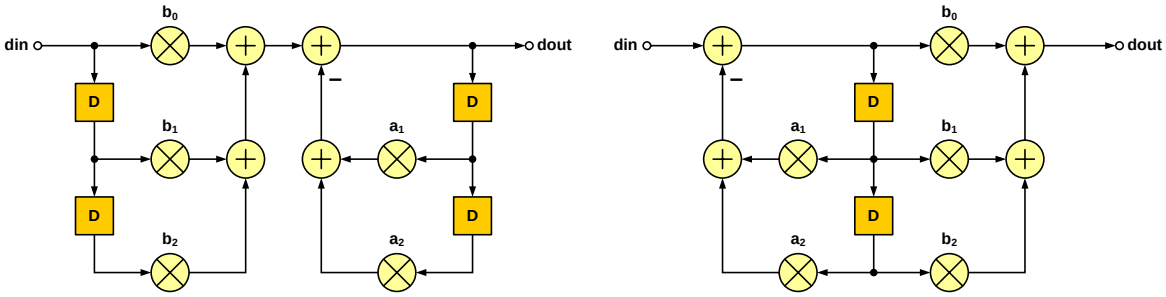
The challenge is how to design a filter with the desired magnitude $Y^d(\omega)$. A FIR filter would require a very large number of taps, while an all-pole or auto-regressive filter, although relatively more efficient, would still requires a high order to approximate the autocorrelation for large lags. Our approach uses an adaptation of the design method presented in [88] for designing IIR filters with an arbitrary impulse response.

According to this method, an IIR filter of order 2N is synthesized as a cascade of N second-order canonic sections (bi-quads). A general second-order filter is defined by five coefficients. Its transfer function has two poles and two zeros, which are usually complex conjugate pairs,

having the following transfer function:

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}} \tag{4.10}$$

The straightforward implementation of this equation leads to the configuration in **Figure 4.16a**, referred to as the direct form I or DF-I. The first stage implements the zeros (numerator), while the second one implements the poles (denominator) or the autoregressive part. If the two stages are swapped, a more hardware-efficient implementation is obtained, which saves two data registers, as shown in **Figure 4.16b**. This configuration is referred to as the direct form II or DF-II, and will be used in our implementation.



(a) Direct form I (b) Direct form II

Figure 4.16.: Second-order canonical filter section

For N cascaded second-order sections, the transfer function is given by the following equation, in which the constant K has been obtained by factoring out the b_0 coefficients. In this way, each section is defined by four coefficients instead of five.

$$H(z) = K \prod_{n=1}^{N} \frac{1 + b_{1,n} z^{-1} + b_{2,n} z^{-2}}{1 + a_{1,n} z^{-1} + a_{2,n} z^{-2}} \tag{4.11}$$

For $z = e^{j\omega}$, $H(z)$ approaches the desired magnitude response $Y^d(\omega)$. The design of the filter is an optimization problem which consists in finding the set of $4N$ coefficients ($a_{1,n}$, $a_{2,n}$, $b_{1,n}$, $b_{2,n}$) that leads to the best approximation of $Y^d(\omega)$.

4.3.1. Filter Design Algorithm

The first step is to discretize $Y^d(\omega)$ by dividing the Nyquist interval $\omega \in [0, \pi]$ into $M + 1$ frequency points. We have that $\omega_i = \pi i / M$ and $Y_i^d = Y^d(\omega_i)$, where $i = 0, 1, \ldots, M$. We also define $z_i = e^{j\omega_i}$.

For a Jakes spectrum, given by (**2.5**), we define $L = \lfloor \rho M \rfloor$, where $\rho \in [0, 1]$ is the desired discrete Doppler rate. The discretized magnitude response is thus given by the following equation:

$$Y_i^d = \begin{cases} \sqrt{\dfrac{1}{\sqrt{1-(i/L)^2}}} & i = 0, 1, \ldots, L-1 \\ \sqrt{L\left(\dfrac{\pi}{2} - \arcsin\left(\dfrac{L-1}{L}\right)\right)} & i = L \\ 0 & i = L+1, \ldots, M \end{cases} \tag{4.12}$$

The response for $i = L$ results from the requirement that the area under the spectrum be equal for the sampled and the continuous cases.

We define the vector $\mathbf{x}$ of length $4N$ that contains the coefficients $b_{1,n}, b_{2,n}, a_{1,n}, a_{2,n}$ and express $H(z) = KF(z, \mathbf{x})$, where $F(z, \mathbf{x})$ is the product of biquad transfer functions in (**4.11**), apart from K. Designing the filter consists now in minimizing the mean squared error (MSE):

$$E(K, \mathbf{x}) = \frac{1}{M+1} \cdot \sum_{i=0}^{M} \left(|KF(z_i, \mathbf{x})| - Y_i^d\right)^2 \tag{4.13}$$

$E(K, \mathbf{x})$ is a function of $4N + 1$ variables. In order to reduce the problem order, we determine the value of K that minimizes $E(K, \mathbf{x})$. Knowing that K is positive, differentiating E with respect to K and equating with zero yields the optimum value K^o:

$$K^o = \frac{\sum_{i=0}^{M} |F(z_i, \mathbf{x})| Y_i^d}{\sum_{i=0}^{M} |F(z_i, \mathbf{x})|^2} \tag{4.14}$$

The problem is now reduced to optimizing $R(\mathbf{x}) = E(A^o\mathbf{x})$ in $4N$ dimensions. The gradient $\nabla_{\mathbf{x}} R(\mathbf{x})$ is a vector with the partial derivatives of $R(\mathbf{x})$ with respect to each element x_v of $\mathbf{x}$, where $v \in [1, 4N]$.

$$\frac{\partial R(\mathbf{x})}{\partial x_v} = \frac{2K^o}{M+1} \cdot \sum_{i=0}^{M} \left(K^o |F(z_i, \mathbf{x})| - Y_i^d\right) \cdot \frac{\partial |F(z_i, \mathbf{x})|}{\partial x_v} \tag{4.15}$$

Evaluating (**4.15**) for all frequencies i and biquad stages n requires the calculation of $4MN$ partial derivatives:

$$\frac{\partial|F(z_i,\mathbf{x})|}{\partial a_{1,n}} = +|F(z_i,\mathbf{x})| \cdot \Re\left\{\frac{z_i^{-1}}{1+a_{1,n}z_i^{-1}+a_{2,n}z_i^{-2}}\right\} \tag{4.16}$$

$$\frac{\partial|F(z_i,\mathbf{x})|}{\partial a_{2,n}} = +|F(z_i,\mathbf{x})| \cdot \Re\left\{\frac{z_i^{-2}}{1+a_{1,n}z_i^{-1}+a_{2,n}z_i^{-2}}\right\} \tag{4.17}$$

$$\frac{\partial|F(z_i,\mathbf{x})|}{\partial b_{1,n}} = -|F(z_i,\mathbf{x})| \cdot \Re\left\{\frac{z_i^{-1}}{1+b_{1,n}z_i^{-1}+b_{2,n}z_i^{-2}}\right\} \tag{4.18}$$

$$\frac{\partial|F(z_i,\mathbf{x})|}{\partial b_{2,n}} = -|F(z_i,\mathbf{x})| \cdot \Re\left\{\frac{z_i^{-2}}{1+b_{1,n}z_i^{-1}+b_{2,n}z_i^{-2}}\right\} \tag{4.19}$$

We have now all quantities needed for iterative optimization. The evaluation of the cost function and the gradient is performed according to the following steps:

1. Starting from a vector $\mathbf{x}$, evaluate $F(z_i,\mathbf{x})$ at all frequencies $i \in [0,M]$.

2. Compute the optimum scaling factor K^o using (**4.14**).

3. Evaluate the error E_i at all frequencies $i \in [0,M]$.
$$E_i = K^o|F(z_i,\mathbf{x})| - Y_i^d \tag{4.20}$$

4. Evaluate the squared error cost function $R(\mathbf{x})$.
$$R(\mathbf{x}) = \frac{1}{M+1} \cdot \sum_{i=0}^{M} E_i^2 \tag{4.21}$$

5. Determine the elements of gradient vector $\nabla_{\mathbf{x}}R(\mathbf{x})$. In the equation below, x_v is one of the $a_{1,n}, a_{2,n}, b_{1,n}, b_{2,n}$ biquad coefficients, where $v \in [1,4N]$ and $n \in [1,N]$.
$$[\nabla_{\mathbf{x}}R(\mathbf{x})]_v = \frac{2K^o}{M+1} \cdot \sum_{i=0}^{M} E_i \frac{\partial|F(z_i,\mathbf{x})|}{\partial x_v} \tag{4.22}$$

Knowing how to evaluate the cost function and the gradient, we can use an iterative optimization technique to find an optimum coefficient set. We have selected the Ellipsoid algorithm because it is very simple to code and works well with highly non-linear target functions. The convergence is not that fast as in the case of a descent method, but since the filter design is done only once this is not a big issue.

4.3.2. A Case Study

In the following, we use the method outlined above to design spectrum-shaping filters for the three Doppler profiles introduced in **Subsection 2.1.2**: Jakes, flat, and Gaussian. The

generated magnitudes are normalized, in the sense that the magnitude is always 1 at DC. We have created a flexible function that generates discrete Doppler profiles, which accepts three parameters:

1. The number of subdivisions M of the Nyquist interval. The resulting Doppler profile will have $M + 1$ elements.
2. The Doppler profile type. It can be either 'jakes', 'flat', or 'gauss'.
3. The Doppler frequency f_D. For the Jakes and flat profiles it is the maximum Doppler frequency f_{Dmax}, while for the Gaussian profiles it represents the Doppler spread σ_D.

For our scenario we used $M = 512$ frequency points since this ensures a good trade-off between precision and computation time. The Doppler frequency f_D was chosen to be 0.25. The rationale is that we want low frequencies because it results in lower interpolation errors. However, if the frequency is too low, the constraints on the spectrum-shaping filter are increased.

The filter design algorithm accepts three parameters:

1. The discrete magnitude response at $M + 1$ frequencies. In this case it is the desired Doppler profile.
2. The number of biquad sections.
3. The target minimum squared error (MSE) for the resulting magnitude response.

In the following, we investigate the relationship between the desired MSE and the required number of biquad sections for the three Doppler profiles, for a discrete Doppler frequency of 0.25. For a number of biquads between 1 and 8 we determine the minimum achievable MSE. The results, shown in **Table 4.4**, indicate that a Gaussian filter requires much fewer stages for a given MSE. The reason is that the magnitude response is very smooth, unlike the Jakes and flat filters which exhibit a very sharp cut-off. The Gaussian filter achieves excellent performance with only two biquad stages, while for the other two at least five stages are required. That is why no values for MSE are given for Gaussian filters with more than two taps. It is also worth mentioning that each extra stage reduces the MSE by approx. 10, i.e. with 10 dB RMS.

Table 4.4 helps us to choose the appropriate number of stages depending on the specific requirements. To be on the safe side, we impose a maximum MSE of 10^{-6}. The appropriate filter sizes and the actual MSE's achieved in one of the optimizations are listed in **Table 4.5**. The MSE is given for the original 512 frequency points at which the filters were optimized.

Figure 4.17 shows the magnitude response of the three designed filters in logarithmic and linear magnitude axes. The logarithmic plots also display the error between the designed and the theoretical magnitude. Since these filters will be used now for both HW and SW implementations, we also list the designed coefficients in **Appendix B** for reference. The multiplicative

# SOS	Doppler type		
	Jakes	**Flat**	**Gaussian**
1	$7.45 \cdot 10^{-2}$	$1.92 \cdot 10^{-2}$	$1.21 \cdot 10^{-4}$
2	$2.09 \cdot 10^{-2}$	$3.65 \cdot 10^{-3}$	$1.74 \cdot 10^{-9}$
3	$2.66 \cdot 10^{-3}$	$8.75 \cdot 10^{-4}$	—
4	$3.03 \cdot 10^{-4}$	$1.20 \cdot 10^{-4}$	—
5	$3.65 \cdot 10^{-5}$	$1.30 \cdot 10^{-5}$	—
6	$4.40 \cdot 10^{-6}$	$1.24 \cdot 10^{-6}$	—
7	$5.10 \cdot 10^{-7}$	$1.24 \cdot 10^{-6}$	—

Table 4.4.: Minimum achievable MSE

Doppler type	Taps	MSE
Jakes	7	$9.50 \cdot 10^{-7}$
Flat	7	$9.81 \cdot 10^{-7}$
Gaussian	2	$1.31 \cdot 10^{-9}$

Table 4.5.: MSE for the designed filters

constant is scaled so that the variance of a white Gaussian noise remains unchanged after filtering.

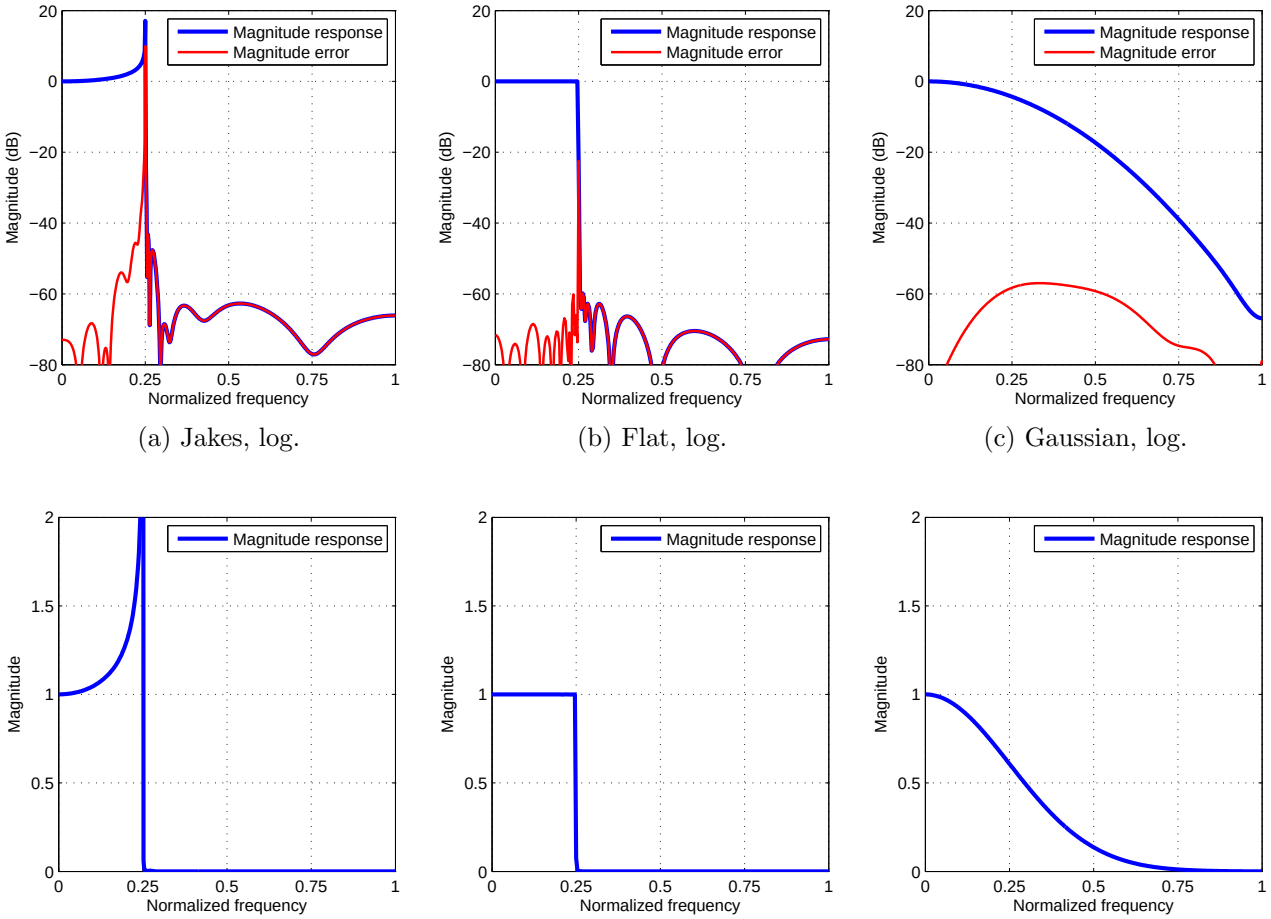
Since the filter introduces correlation between the samples of a white Gaussian noise and reduces the noise variance (power), the signal has to be amplified after filtering to restore its original power level. If we know the impulse response $g(n)$ of the filter (discrete and infinite), the subunit power gain A_p is given by (**4.23**). The higher the number of samples considered, the higher the accuracy.

$$A_p = \frac{\sigma_{out}^2}{\sigma_{in}^2} = \sum_{n=0}^{\infty} g(n)^2 \tag{4.23}$$

If we know the magnitude response $H(f)$ (continuous and finite), the power gain can be also expressed as in , where $N+1$ is the number of frequencies at which the magnitude response is evaluated. For good precision, N must be high enough, e.g. at least 1024.

$$A_p = \frac{\sigma_{out}^2}{\sigma_{in}^2} = \frac{1}{N+1} \sum_{i=0}^{N} H(i/N)^2 \tag{4.24}$$

Once the power gain A_p is determined, the normalization factor can be simply calculated as the inverse of the square root of A_p. For the three filters we designed, we merged the multiplicative constant K with the normalization factor to end up with only a multiplication instead of two.



(a) Jakes, log. (b) Flat, log. (c) Gaussian, log.

(d) Jakes, lin. (e) Flat, lin. (f) Gaussian, lin.

Figure 4.17.: Magnitude responses of the designed Doppler filters

The constants K in **Table B.1** are already normalized to ensure a unity power gain for WGN input.

4.3.3. Implementation Guidelines

The filter design algorithm described in the previous section returns second-order sections with poles and zeros in arbitrary order. Finite-precision hardware realizations, however, require the optimization of the filter in order to reach one of the following two goals: 1) minimizing the probability of overflow or 2) minimizing the peak round-off noise. In the case of a Doppler spectrum shaping filter, the latter optimization is more desirable.

For a given transfer function implemented with second-order sections, minimizing the peak round-off noise is achieved by 2-norm scaling and by reordering of the sections so that the poles

are in descending order [91], which means that the first section has the poles closest to the unit circle. The zeros and the poles are then grouped according to their proximity, starting with the poles closest to the unit circle and successively matching each pole with the closest remaining zeros until all of them are matched.

The goal of the 2-norm scaling is to achieve a constant noise variance after each section. The 2-norm of a filter is defined by the equation below and is intuitively the area under the squared magnitude response or the power gain of a filtered white noise.

$$\|H\|_2 = \sqrt{\frac{1}{2\pi}\int_0^{2\pi} |H(e^{j\omega})|^2 \, d\omega} \tag{4.25}$$

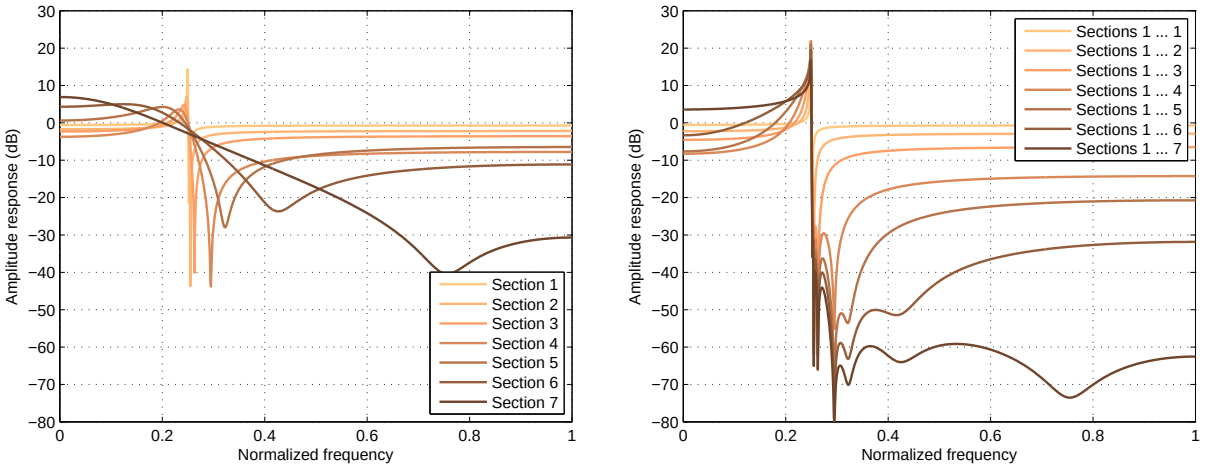
The normalization is performed by adjusting the b_0 coefficient of each section so that the 2-norm of the filter formed by cascading the current section with all previous sections becomes one. The process starts with the first section and is described by the equation below. In this equation H_k is the magnitude response of the individual sections, while $H_{1,n}$ is the magnitude response of all cascaded sections up to and including current section n of the total number of sections N.

$$\|H_{1,n}\|_2 = \sqrt{\frac{1}{2\pi}\int_0^{2\pi} \left|\prod_{k=1}^{n} H_k(e^{j\omega})\right|^2 d\omega} = 1, \quad \forall n \in 1 \ldots N \tag{4.26}$$

In the case of a Doppler spectrum shaping filter, the variance of the input white noise has to be preserved. Using the above scaling method, the variance remains the same after each filter section, and the additional multiplicative constant K is no longer necessary. With each new section, the frequency response converges to the overall filter response. As an example, we consider the Jakes filter designed in the previous section, which has seven second-order sections. **Figure 4.18** shows the frequency response of the individual sections, as well as their cumulative response.

The hardware realization of a digital filter involves the discretization of the data samples and the coefficients. After multiplications and accumulation, the bitwidth of the intermediate results becomes very large and has to be scaled down. Dedicated scaling (casting) blocks are used for this purpose. For a discretized second-order section, the schematic containing the casting blocks is shown in **Figure 4.19**. The casting to a higher precision is denoted by <<, while the casting to a lower precision by >>. In order to avoid the two subtractions in the straightforward DF-II implementation, shown in **Figure 4.16b**, the minus is embedded into a_1 and a_2, which incurs no additional hardware costs.

Figure 4.19 also shows the bitwidth at different points in the signal path. The four bitwidths encountered in the implementation are the following:



(a) Individual sections (b) Cumulative sections

Figure 4.18.: Magnitude response of the second-order sections for the designed Jakes filter

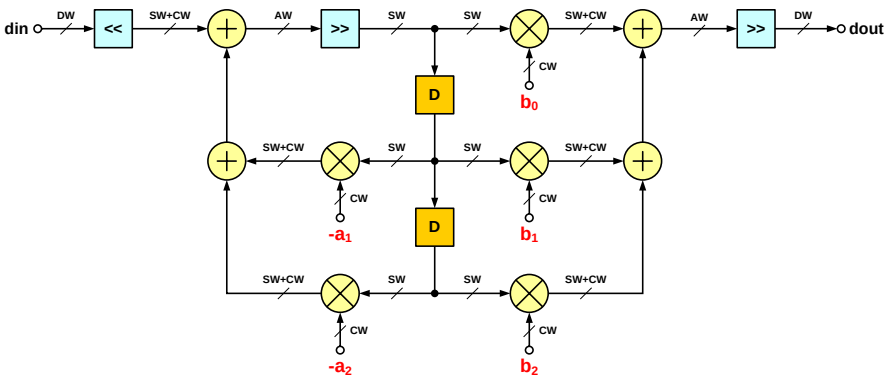


Figure 4.19.: Discretized second-order filter section with casting blocks

- **DW**: data bitwidth
- **CW**: coefficients bitwidth
- **SW**: states bitwidth
- **AW**: accumulator bitwidth

The result after multiplications and accumulation has the largest bitwidth, which must be at least as large as the bitwidth after multiplication, SW+CW. The accumulation result needs to be cast down to the bitwidth of the filter states or of the output. Depending on the requirements, casting can be performed by truncation or rounding. Rounding produces a lower round-off noise but requires an extra adder. Another parameter of the casting is the overflow behavior, which can be either wrapping or saturation. Wrapping is obtained with no hardware costs, by simply discarding a number of MSB's. Saturation, however, requires two comparators and two

multiplexors. Unlike casting to a lower precision, casting to a higher precision only consists in appending a number of zeros.

If multiple second-order sections are cascaded, casting of the input to the accumulator precision and of the accumulator to the output precision must be only performed once. No casting is necessary between the sections. Moreover, the accumulation of the multiplications with the a coefficients of a section can be combined with the accumulation of the multiplications with the b coefficients of the previous section. These considerations can be better observed in **Figure 4.20**, which shows three cascaded second-order sections, with the shared accumulations marked by dashed rectangles.

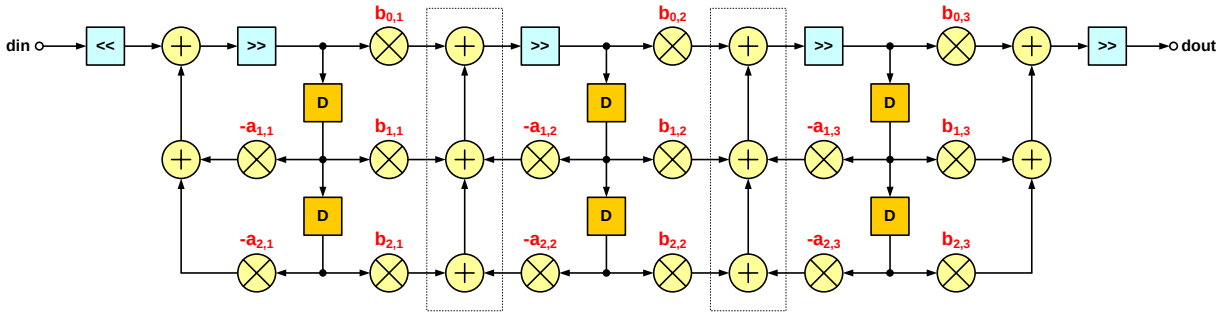


Figure 4.20.: Cascade of three second-order filter sections

Depending on the required throughput, different architectures can be used for the implementation of a cascade of second-order filter sections. In the following we denote with N the number of filter sections. The three most straightforward solutions are outlined in the following list.

- Fully parallel. This solution enures the highest throughput, of one sample per clock cycle, but also requires the most hardware resources. Each section has 4 multipliers, 5 adders, and 2 registers, exactly like in **Figure 4.16b**. The requirements grow linearly with N.

- Individual sequential. Each section contains one multiply-and-accumulate (MAC) unit, which is operated sequentially under the control of a small state machine. The computation of one output sample takes 5 clock cycles, which ensures a throughput independent of N. The complexity, however, grows linearly with N as well.

- Fully sequential. All computations are performed sequentially using a single MAC unit under the control of a state machine. The computation time is 5N clock cycles, growing linearly with the number of sections. The essential advantage is that it has the smallest area. One drawback, however, is the fact that the internal bitwidths must be the same for all sections, which precludes the local fine tuning possible for the other two solutions.

As the required throughput before interpolation is very low in the case of a Doppler spectrum generator, the fully sequential architecture is the ideal candidate for a hardware implementation. A possible implementation solution is presented in **Figure 4.21**. The filter states for all sections

are stored in a single synchronous RAM block, whereas the constant coefficients are stored in a dedicated ROM. The memory maps for both ROM and RAM are shown in **Figure 4.21**.

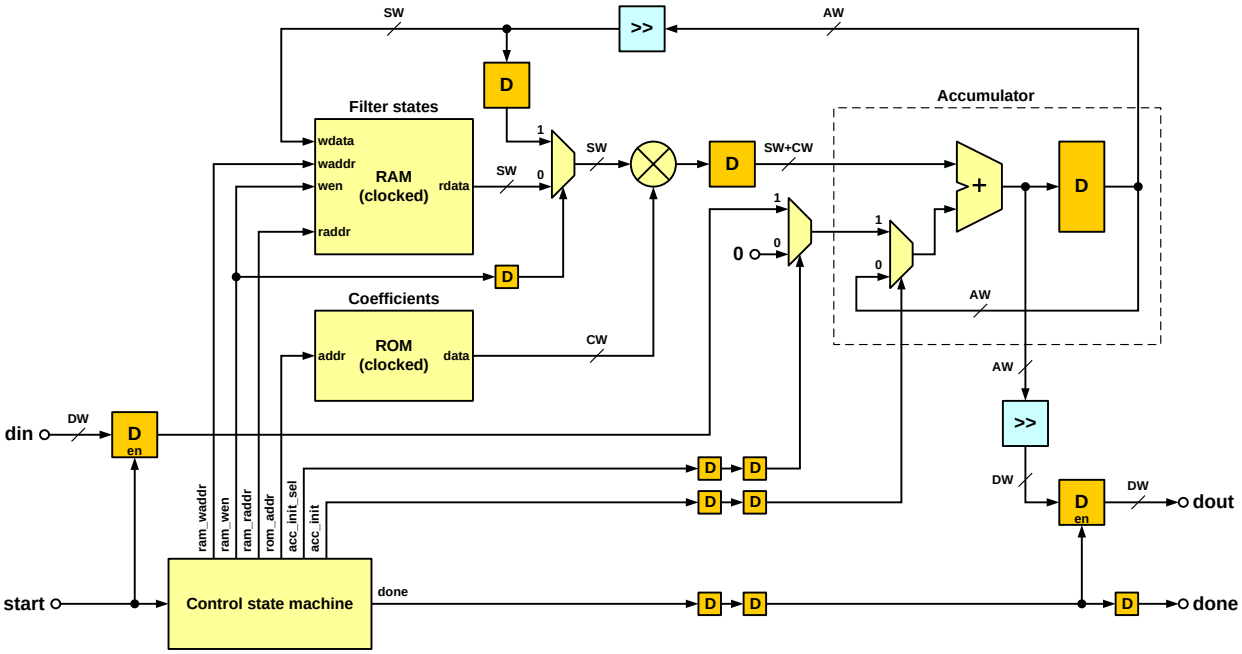


Figure 4.21.: Sequential architecture for cascaded second-order sections

It is worth mentioning that no full dual-port RAM is needed because requires the same address is used for read/write. Moreover, if the RAM supports the read-after-write mode, as it is the case in many FPGA embedded RAM blocks, the multiplexor in front of the multiplier and the delay register on the feedback path can be saved. The accumulation result that is written into the RAM would be available at its output in the next clock cycle.

If the fading tap generator needs to support different Doppler profiles, such as Jakes, flat, or Gaussian, the number of sections and the filter coefficients must be made configurable. This can be easily achieved with the presented architecture by making the control state machine dependent on the number of sections (configuration parameter) and by replacing the coefficients ROM with a dual-port RAM block that can be written by a central processor. Synchronous on-chip memories, which can be used as RAM or ROM, are readily available in almost all modern FPGA devices. Most FPGA synthesis tools are able to infer them from HDL, which results in generic and scalable designs.

The control state machine is relatively simple and consists of two counters plus a few additional logic gates. One counter iterates through all sections, while the other sequences the 5 MAC operations for each section. The output signals of the state machine are shown in **Figure 4.23** in the case of a three-section filter. The proposed architecture has no wait states and keeps the MAC busy for the entire duration of a computation, so that the computation takes exactly 5N clock cycles, where N is the number of filter sections. Pipelining increases the filter latency with 3 cycles, but does not affect its throughput.

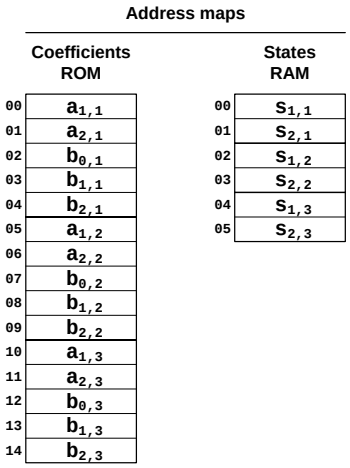


Figure 4.22.: Address maps for the coefficients and the filter states

cycle #	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
start	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
rom_addr		$a_{2,1}$	$a_{1,1}$	$b_{2,1}$	$b_{1,1}$	$b_{0,1}$	$a_{2,2}$	$a_{1,2}$	$b_{2,2}$	$b_{1,2}$	$b_{0,2}$	$a_{2,3}$	$a_{1,3}$	$b_{2,3}$	$b_{1,3}$	$b_{0,3}$
ram_addr		$s_{2,1}$	$s_{1,1}$	$s_{2,1}$	$s_{1,1}$	$s_{2,1}$	$s_{2,2}$	$s_{1,2}$	$s_{2,2}$	$s_{1,2}$	$s_{2,2}$	$s_{2,3}$	$s_{1,3}$	$s_{2,3}$	$s_{1,3}$	$s_{2,3}$
ram_wen		0	0	0	0	1	0	0	0	0	1	0	0	0	0	1
acc_init		1	0	1	0	0	0	0	1	0	0	0	0	1	0	0
acc_init_sel		1	-	0	-	-	-	-	0	-	-	-	-	0	-	-
done		0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

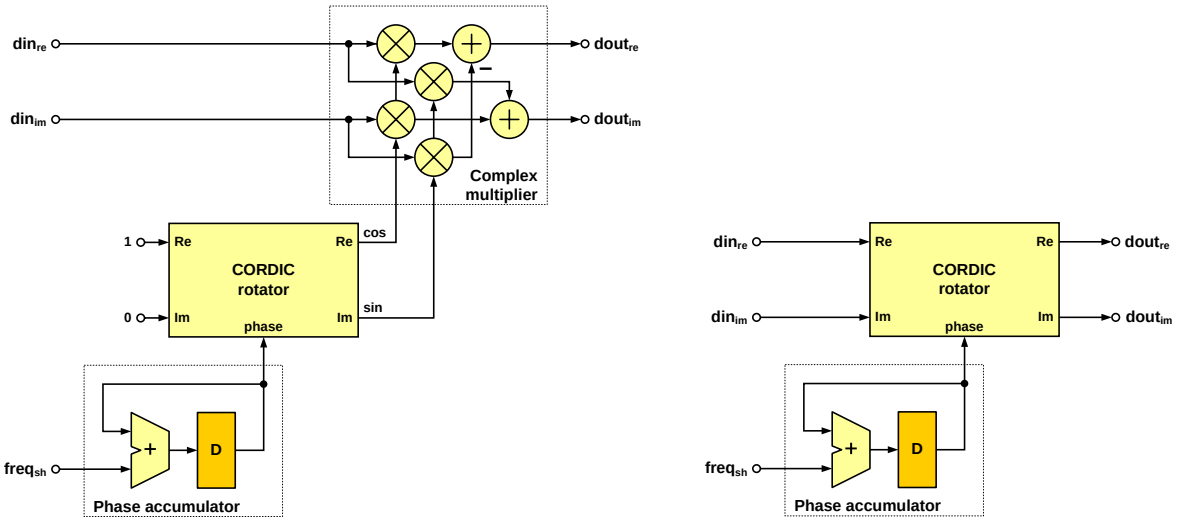
Figure 4.23.: Signals generated by the control FSM for a three-section filter

4.4. Spectrum Shifter

In some cases, such as the ionospheric channel profiles, a constant Doppler shift is specified for each path, in addition to the Doppler spread. This frequency shift is performed by multiplying the filtered complex signal with a complex exponential $exp(-j\pi f_{Dsh}n)$, where f_{Dsh} is the normalized Doppler shift between 0 and 1 and n is the sample index. It is essential that the frequency shift do not translate the original symmetric spectrum past the Nyquist limit, i.e. the condition $f_{Dmax} + f_{Dsh} < 1$ has to be fulfilled. Otherwise aliasing would occur.

Implementing a frequency shift in SW is trivial and consists in a call to *sin* and *cos* followed by a complex multiplication with the input signal. In HW, however, generating a *sin*/*cos* pair and then performing the complex multiplication is not the most efficient approach. A discrete *sin*/*cos* generator with programmable frequency is usually implemented using a phase accumulator and a look-up table. An alternative solution is to use a CORDIC rotator to compute the *sin*/*cos* values for the generated phase. This HW solution is shown in **Figure 4.24a**.

The complex multiplier can be completely eliminated if we realize that there is no need to actually generate *sin* and *cos*, but only to rotate the complex input signal with the phase



(a) Direct implementations (b) Optimized implementations

Figure 4.24.: Frequency shifter implementations

computed by the phase accumulator. The same CORDIC rotator used for *sin/cos* generation can be directly inserted in the signal path, thus obviating the need for a complex multiplier. This HW solution is shown in **Figure 4.24b** for comparison.

The width of the phase accumulator depends on the required frequency resolution. For a N-bit accumulator, the frequency can be varied linearly between 0 and 1 in 2^{N-1} increments, with a frequency resolution of $1/2^{N-1}$. If the phase accumulator increment is A_{inc}, the normalized frequency shift is exactly $A_{inc}/2^{N-1}$.

4.5. Polyphase Interpolator

The interpolation factors can be very high. The narrower the desired spectrum, the higher the interpolation factor. Values of 1000 are not uncommon in many applications, like Doppler fading tap generators.

Various interpolation solutions for fading taps generators have been proposed in the literature. In [51] the interpolation factor is restricted to integers and a single poly-phase interpolation stage is used. In [82] a multi-stage approach is preferred, such as described in [14]. The interpolation factor is in this case a composite number, i.e. $L = \prod_{k=0}^{K-1} L_k$, where K is the number of successive interpolation stages with integer interpolation factors.

Restricting the interpolation factor to integers may not be flexible enough for certain applications. One example would be a simulation that requires the Doppler frequency to sweep a given interval with many intermediate steps. The accuracy will suffer because of the discontinuities caused by the discrete set of the Doppler frequencies that can be generated. However,

the computational efficiency of the integer factor interpolation is very good for a poly-phase implementation, especially for single-stage solutions.

Another essential disadvantage of the conventional interpolation solutions is the fact that they are very inflexible for hardware implementation. Changing the interpolation factor entails the changing of all poly-phase coefficient sets. This is not an issue in software, where the coefficients are computed "off-line" before the simulation starts. In hardware, however, the different coefficients need to be computed off-line during design and stored, for each interpolation factor. There are practical solutions that alleviate this problem by reducing the number of stored coefficients, but they all suffer from increased complexity and reduced flexibility.

The solution we propose eliminates all the above mentioned problems. It offers very high flexibility in choosing the interpolation factor, without sacrificing the computational efficiency and scalability, being at the same time suitable for both software and hardware implementation. Moreover, both up-sampling and down-sampling can be implemented with the same structure, which might be of interest in some applications. In software, it allows for arbitrary interpolation factors, the only limitation being the intrinsic floating-point number precision. In hardware, the interpolation factors are restricted to the formula $2^N/X$, where N is the width of an accumulator register and X is an integer increment.

4.5.1. Architecture Overview

The architecture of the interpolator (resampler), shown in **Figure 4.25**, consists of three main components:

- Phase accumulator
- Coefficients generator
- Interpolation (resampling) FIR filters

The phase accumulator keeps track of the sub-sample phase by incrementing its value with a constant value for each output sample. A new sample is read from the input each time the accumulator overflows. We denote with N the width of the accumulator and with A_{inc} the increment. The upsampling factor, $f_{s,out}/f_{s,in}$, is given by the formula:

$$F = \frac{2^N}{A_{inc}} \tag{4.27}$$

For upsampling, we have that $A_{inc} < 2^N$. An example of upsampling process is shown in **Figure 4.26** for $N = 4$ and $A_{inc} = 5$, with a resulting upsampling factor of 16/5. In the figure, red arrows mark the transitions for which accumulator overflow occurs. It is readily apparent that the accumulator phase represents the sub-pixel phase of the output interpolated samples.

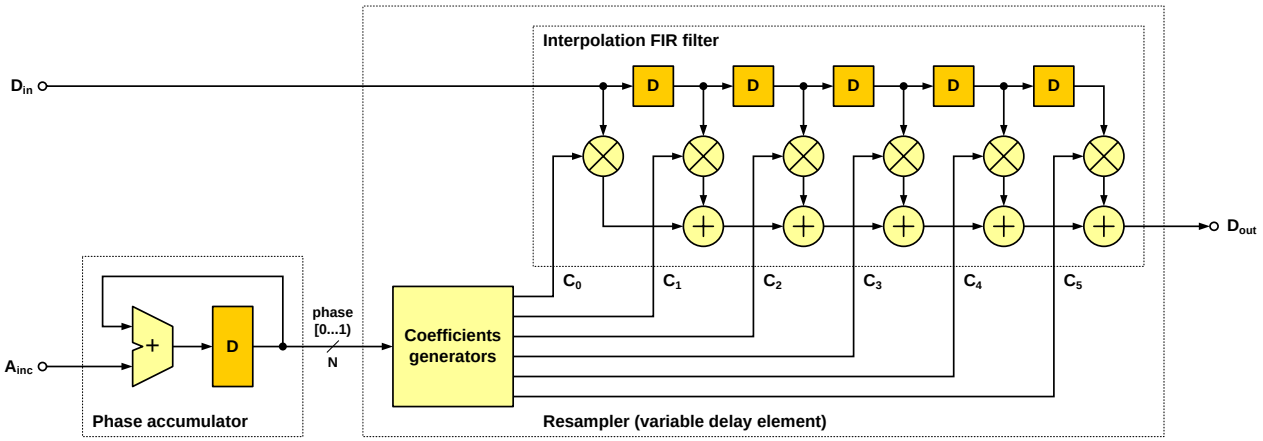


Figure 4.25.: Resampler architecture

This normalized phase is computed as $A_{inc}/2^N$ and lies in the range $[0, 1)$.

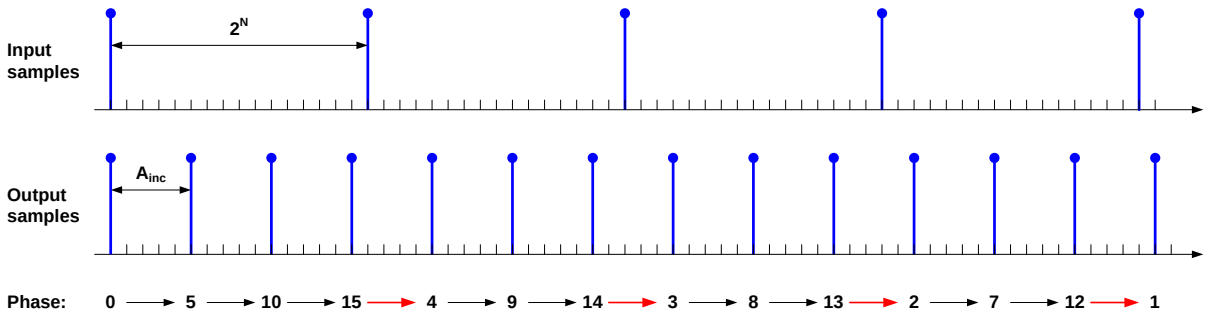


Figure 4.26.: Example of upsampling process

The interpolator works in pull mode, i.e. it is driven by the output. The output data rate is fixed and controls the phase accumulator. The input rate is lower and variable, depending on the actual interpolation factor. Decimators, on the other hand, work in push mode, where the accumulator is operated with the fixed rate of the input, while the output rate depends on the decimation factor.

The sub-pixel phase generated by the phase accumulator is now used for resampling. Resampling means to generate an output sample from a predefined number of input samples, with a relative phase between two original samples. The resampling process consists in multiplying the input samples with a set of weights that depend on the desired phase. Physically, the resampler is implemented as a FIR filter with variable coefficients and a block that generates the appropriate coefficients based on the desired phase. In the field of communication, the resampler block is also referred to as variable delay element, since it can be regarded as introducing a sub-sample delay in the input signal.

An advantage of the proposed interpolation architecture is apparent the case of multi-channel interpolation, i.e. when more than one data channel is upsampled at the same time. In our

case, the signal to be upsampled is complex, so we have two channels processed in parallel. Other cases include multi-channel audio sampling rate conversion and image scaling, e.g. RGB or YUV. In this case, the phase accumulator and the coefficients generator can be shared and only one interpolation FIR filter per extra channel is needed, as shown in **Figure 4.27**.

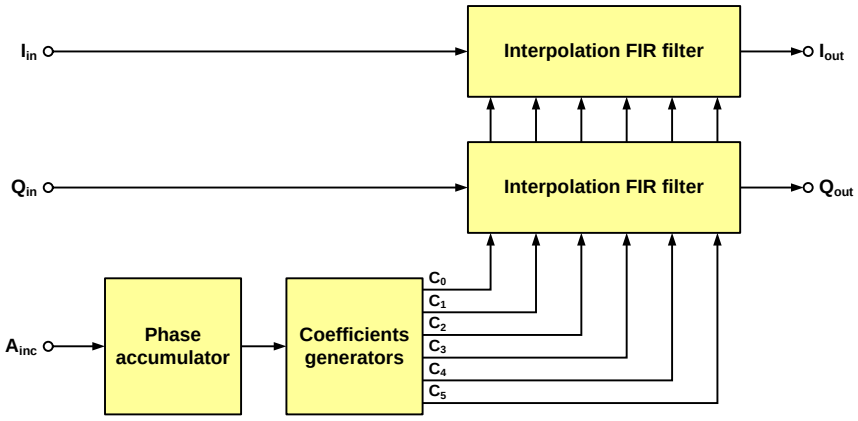
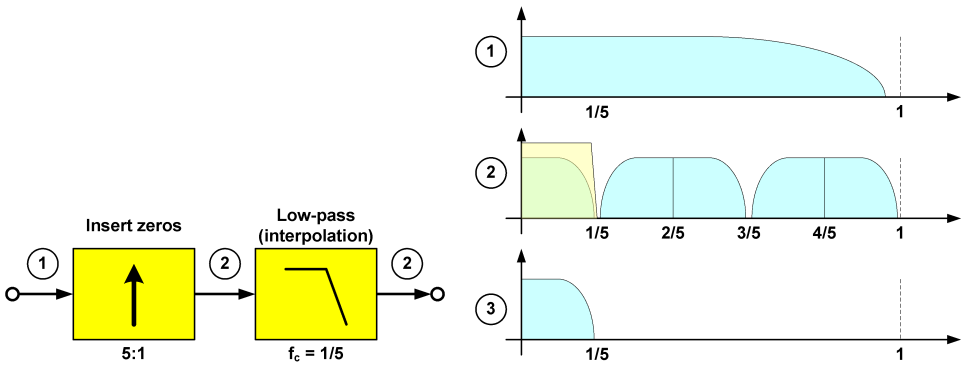


Figure 4.27.: Multi-channel interpolation architecture

4.5.2. Polyphase Coefficients Generator

In order to determine the relationship between the desired phase and the coefficients of the interpolation filter we discuss first the poly-phase decomposition of interpolation functions. The principle of the interpolation is to first insert zero samples between the original samples, then apply a low-pass filter to reject the images created by oversampling at multiples of the sampling frequency. In **Figure 4.28** we show the block schematic and the spectra for a 5x interpolation. First, 4 zeros are inserted between original samples, followed by a low-pass filtering with $f_c = 1/5$.



(a) Structure (b) Spectra

Figure 4.28.: Interpolation structure and spectra

If the interpolation filter is ideal, i.e. rectangular with $f_c = 1/5$, no loss of information occurs. When real filters are used, however, the interpolated signal will be distorted. There are two classes of distortions: a) linear distortions, due to the high-frequency attenuation of the filter, and b) non-linear distortions, caused by the insufficient attenuation of the image components.

Since most of the multiplications in the filter are with zero samples, the straightforward implementation of the textbook structure in **Figure 4.28** is very inefficient. For larger interpolation factors the situation becomes worse. The standard solution is to use a filter with coefficients that depend on the phase of the output signal. For 8× interpolation, there are eight possible output phases and therefore eight coefficient sets.

An example of polyphase decomposition of the original interpolation FIR filter is shown in **Figure 4.29**, where the eight phases have been shown using different colors. The original filter is symmetrical with 47 taps, whereas the poly-phase filter has only 6 taps and 8 sets of coefficients. Thus, the number of MAC operations per output sample decreased from 47 to 6. The saving is even more significant for very high interpolation factors, for which the textbook filter approach would be extremely inefficient.

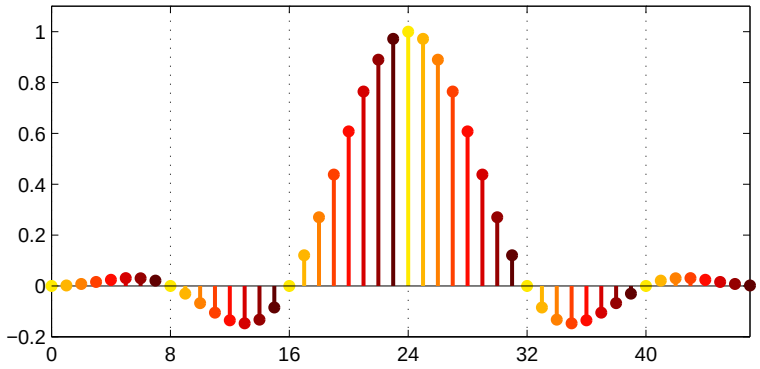


Figure 4.29.: Interpolation filter poly-phase decomposition

Besides the reduced number of MAC operations, another essential advantage of the polyphase structure is that the interpolation filter size is independent of the interpolation factor. The latter determines only the number of coefficients sets. Such a poly-phase implementation can be used for generating interpolated samples with any phase between 0 and 1 by simply using the appropriate coefficients set. If the number of desired phases becomes very large, the storage needed for coefficients becomes prohibitive.

The solution we propose enables the generation of any output phase using a relatively low number of stored coefficients sets. The idea is to store coefficients for a limited number of equidistant phases P, usually a power of 2. For a given phase, the actual filter coefficients are computed by linear interpolation between two adjacent coefficients sets. **Figure 4.30** shows the interpolation process for the two central taps of a 6-tap filter. The number of coefficients sets for P phases is P+1 because the coefficients for phase 1 are needed for linear interpolation. These are simply the coefficients for phase 0 reversed. In the case of 4 poly-phases, 5 coefficients sets are stored, one for each of the following phases: 0, 1/4, 2/4, 3/4 and 1.

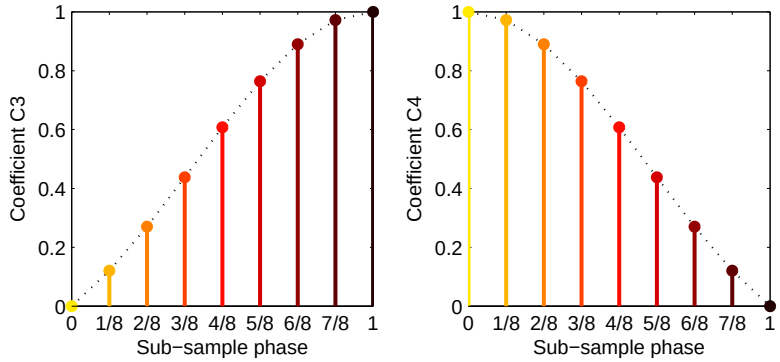


Figure 4.30.: Linear interpolation between stored coefficients

The sampling interval is thus divided into P segments of equal widths. In order to perform linear interpolation, the segment number K_s and the intra-segment phase ϕ_s have to be computed, using the following relationships. The desired output phase in range $[0 \ldots 1)$ is denoted here by ϕ_o.

$$K_s = \lfloor P * \phi_o \rfloor, \quad K_s \in [0, 1, \ldots, P-1] \tag{4.28}$$

$$\phi_s = P \cdot \phi_o - K_s, \quad \phi_s \in [0 \ldots 1) \tag{4.29}$$

The output coefficient C_{int} is computed by linear interpolation between the selected adjacent coefficients C_{K_s} and C_{K_s+1}:

$$C_{int} = C_{K_s} + \phi_s \left(C_{K_s+1} - C_{K_s} \right) \tag{4.30}$$

In hardware, the phase ϕ_o is encoded using a fixed number of bits N. If the number of coefficients sets is a power-of-2, say 2^Q, the selection of the two adjacent coefficients is done with two multiplexors controlled by the first Q MSB's of the N-bit phase word. The remaining $N - Q$ bits represent the intra-segment phase ϕ_s and are directly used for linear interpolation. The schematic is shown in **Figure 4.31** for $2^Q = 4$. Such a structure is needed for every filter coefficient.

In most applications, the coefficients for each phase are normalized, i.e. their sum is one. This ensures that the response at DC is the same for all phases. If this condition is not met, ripple occurs for slowly varying signals, which shows up as high-frequency spurious components in the spectrum of the interpolated signal.

When coefficients have finite precision, the normalization of the interpolated coefficients might be affected, even if the original coefficients sets are normalized. Simulations show that for discretized coefficients having the sum for each phase equal with 256, the resulting sum after linear interpolation can vary with ± 2 around the average 256. The solution is to renormalize

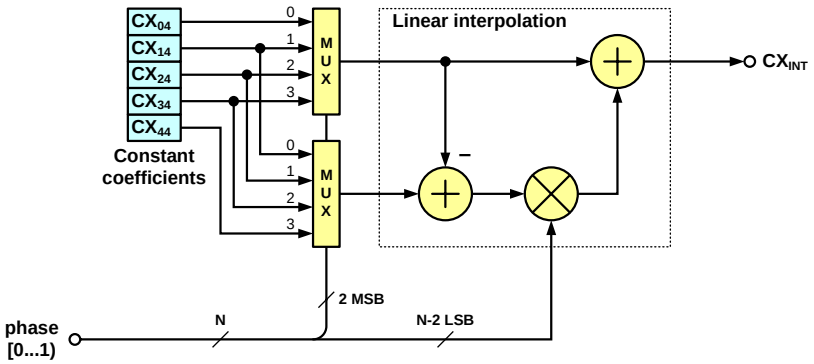


Figure 4.31.: Schematic of a coefficient generator

the coefficients after linear interpolation. Renormalization is performed by computing the error of the sum and subtracting it from one of the two central coefficients or, even better, from the largest of them. The schematic of the proposed solution is shown in **Figure 4.32** for a 4-tap interpolation filter.

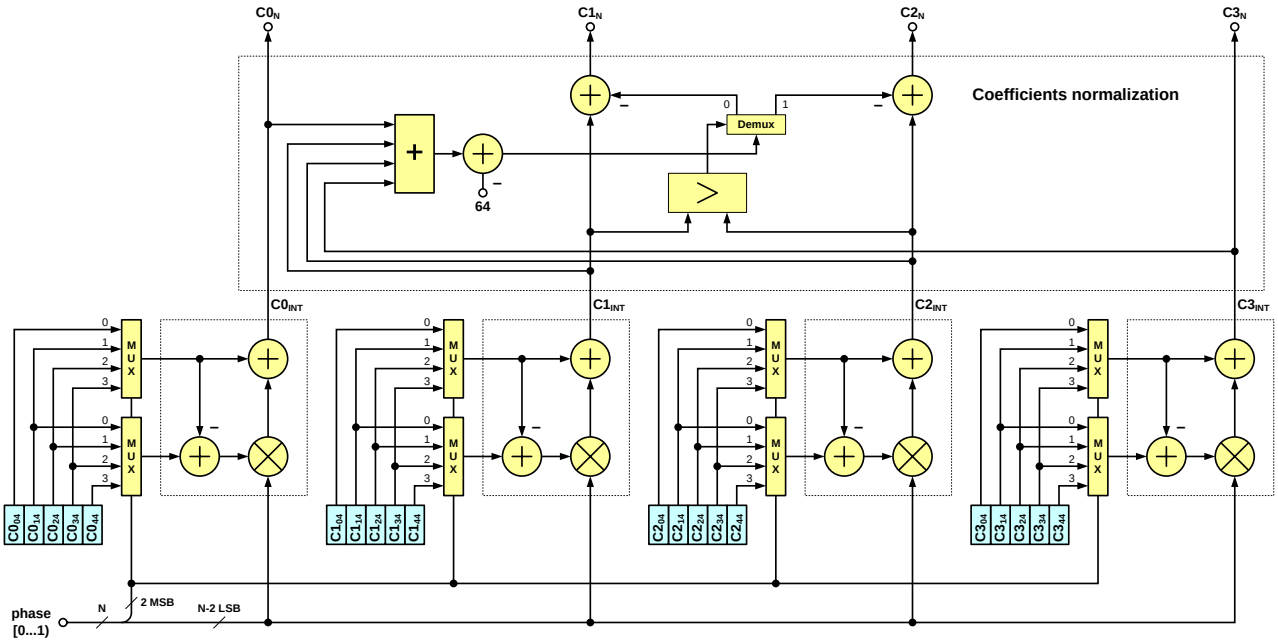


Figure 4.32.: Post-interpolation coefficients renormalization

4.5.3. Interpolation Functions

There are several types of interpolation functions known in the literature, which can be divided into three main categories:

- Polynomial: Lagrange, spline

- Windowed sinc: Lanczos, Hamming, Hanning
- Optimal, matched to the signal's spectrum

The two main parameters of an interpolation function are the number of samples of the original signal that are taken into account for interpolation and the integer interpolation factor. Additionally, the optimal signal-matched interpolation requires the knowledge of the signal spectrum or its autocorrelation function. For our study we consider the Lagrange, Lanczos, and the signal-matched interpolation.

The interpolation function is decomposed into its poly-phase components for efficiency reasons, as shown in **Subsection 4.5.2**. Each output sample is obtained by multiplying a fixed number of neighboring input samples by a set of coefficients that depend on the desired phase.

We want to compare the performance of various interpolation functions for a given input signal. In the following analysis we consider a flat-spectrum band-limited Gaussian noise. First, we consider a bandwidth of 0.25 and determine the mean square error (MSE) of the interpolated output as a function of the sub-pixel phase between 0 and 1. The test bench, shown in **Figure 4.33**, consists of an ideal band-limited noise generator, followed by a decimator and an interpolator. The decimation and the interpolation factors are equal with the number of phases for which the analysis is performed. The bandwidth of the noise generator is the desired bandwidth before interpolation divided by the interpolation factor. It is essential that the generated noise has very low spectral components outside the band of interest, otherwise these would fold back in the Nyquist band after decimation and would appear as interpolation errors.

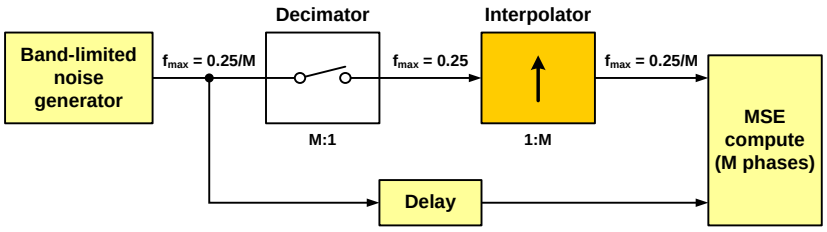
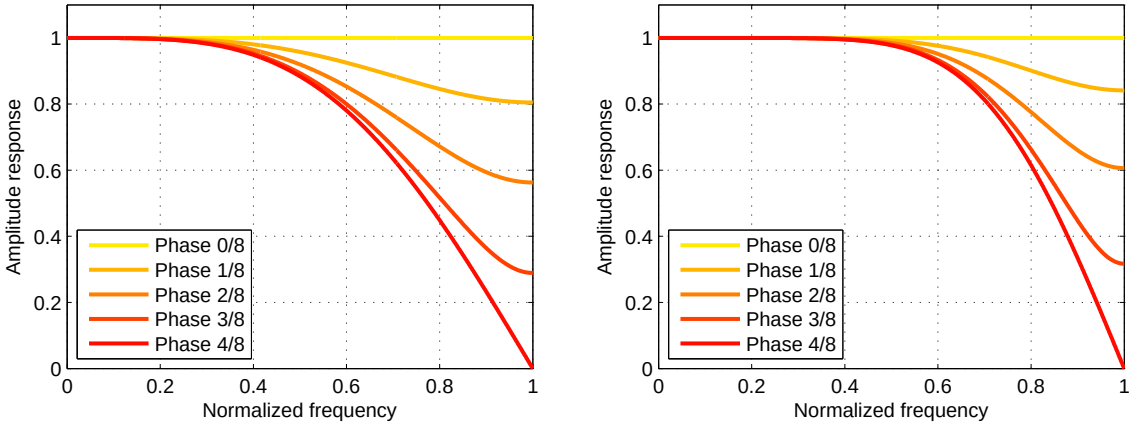


Figure 4.33.: Interpolation error measurement

The frequency response of the **Lagrange interpolation** is shown in **Figure 4.36** for subsample phases between 0 and 0.5, for 4 and 8 taps respectively. As the filter is symmetrical, the coefficients for phase ϕ are the coefficients of phase $1 - \phi$ reversed, and their frequency responses are identical.

The frequency response of the overall Lagrange interpolation filter for an interpolation factor of 8 is shown in **Figure 4.35**, for 4, 6, and 8 taps. As expected, the longer the filter, the better its impulse response. The ideal interpolation filter should have a constant frequency response up to the Nyquist frequency, which is marked with a vertical line on the figure, while outside the Nyquist band the response should be zero. These conditions can only be fulfilled by a 'sinc' filter with an infinite number of taps. Real interpolation filters, however, cannot fulfill either of these conditions, which gives rise to two categories of interpolation errors:



(a) 4 taps (b) 8 taps

Figure 4.34.: Frequency response of Lagrange poly-phase filters

- Linear distortions due to the attenuation of the upper frequencies in the original Nyquist band.
- Non-linear distortions due to insufficient rejection of the image components outside the original Nyquist band (aliasing).

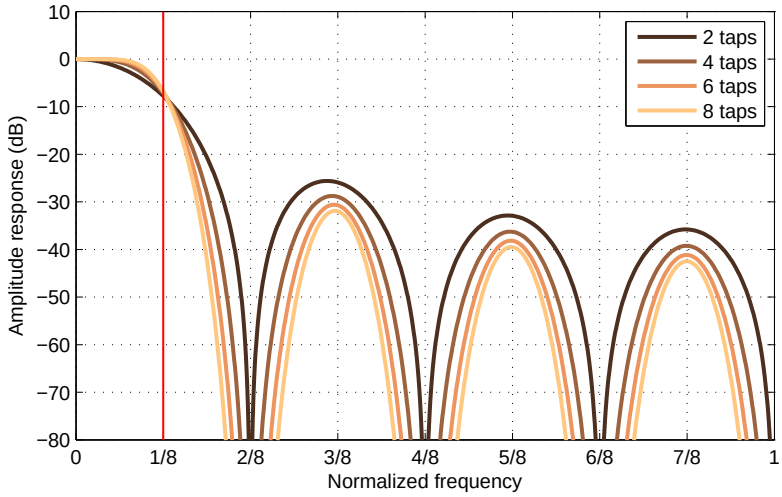


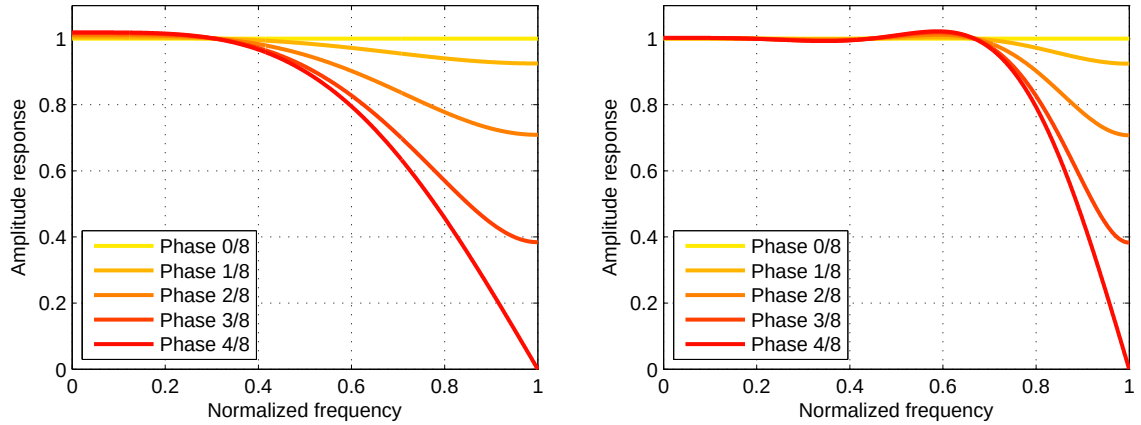
Figure 4.35.: Frequency response of Lagrange interpolation filters

It must be mentioned here that the ultimate cause of aliasing is the difference in frequency response of the poly-phase filters in **Figure 4.36**. If the responses were the same for all phases, no aliasing would occur, only attenuation of the high-frequency components of the original signal.

The Lanczos interpolation belongs to the windowed-sinc family of interpolation functions. In this case, the sinc function is windowed with the main lobe of a wider sinc. The relative width

of later sinc is the interpolation factor, as shown in (**4.31**) for a factor of 3. The frequency response of the poly-phase components for Lanczos interpolation are shown in **Figure 4.36**, for 4 and 8 taps respectively. It can be seen that Lanczos is better than Lagrange for higher-order interpolation filters and for signals with significant high-pass components.

$$L(x) = \begin{cases} \dfrac{\sin(\pi x)}{\pi x}\dfrac{\sin(\pi x/3)}{\pi x/3} & |x| < 3 \\ 0 & |x| \geq 3 \end{cases} \tag{4.31}$$



(a) 4 taps (b) 8 taps

Figure 4.36.: Frequency response of Lanczos poly-phase filters

The frequency response of the overall Lanczos interpolation filter for an interpolation factor of 8 is shown in **Figure 4.37**, for 4, 6, and 8 taps. Unlike the Lagrange filter, whose frequency response has lobes centered around odd multiples of the original Nyquist frequency, the Lanczos filter has a faster decaying frequency response, without clear periodic structures. This property is typical for all windowed sinc filters.

The optimal signal-matched interpolation, described in detail in **Subsection 5.4.2**, has a frequency response close to that of the Lagrange interpolation, e.g. with lobes at odd multiples of the original Nyquist frequency, except that the lobes are smaller. Since this filter is MSE-optimized for a specific signal spectrum, it offers the best interpolation performance, provided that the actual signal has the spectrum for which the filter has been designed. This condition is always fulfilled in the case of a fading tap generator, where the Doppler spectrum before interpolation is known by design.

In the following, we want to evaluate the three interpolation functions for various signal bandwidths and interpolation filter lengths and select the best for our requirements. We consider a frequency range from 1/8 to 1 and filter lengths in range $4 \ldots 16$. The testbench used is the one in **Figure 4.33**. The decimation/interpolation factor does not affect the result and has been

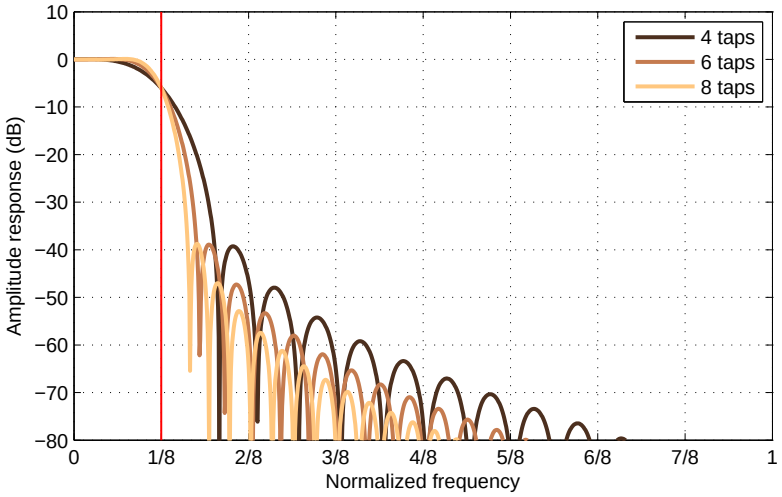


Figure 4.37.: Frequency response of Lanczos interpolation filters

set to 4 in this case. The results are plotted in **Figure 4.38** for a bandwidth between 1/8 and 1 on a logarithmic scale.

The truncated sinc interpolation was added as a reference to show that a smooth window is necessary to achieve decent interpolation results. As expected, the results show that the signal-match interpolation offers the smallest error for all bandwidths. For high bandwidths, Lanczos interpolation is superior to Lagrange. The former has a slightly oscillating error floor at lower bandwidths. For comparison purposes, **Figure 4.39** shows the results for the three interpolation types on the same axes, for 8 and 16 taps, this time only for two octaves between 1/4 and 1.

For 8 taps, the error is less than -80 dB for bandwidths below 0.25, for both Lagrange and signal-matched interpolation. At 0.5 bandwidth, the error of the signal-match interpolation is with 20 dB smaller than for Lagrange. Two important conclusions can be drawn from this analysis. First, Lanczos interpolation, like other windowed-sinc varieties, offers the worst performance for bandwidths below 0.5 and is not appropriate for fading tap generation. Second, interpolation error decreases very fast with the bandwidth, roughly 40 dB per octave for 8 taps and 60 dB per octave for 16 taps. For a bandwidth of 0.25, the interpolation error is less than -90 dB for both Lagrange and signal-matched interpolation. The later achieves -90 dB even with 4 taps instead of 8, which doubles the computational efficiency.

4.5.4. Performance Analysis

The interpolation performance analysis in the previous section has been performed for a fixed interpolation factor, assuming the filter coefficients to be exactly computed for the desired phases. The number of phases is in this case the interpolation factor, so that the coefficients

can be computed off-line and stored in a look-up table. In the case of a resampler, the number of phase is very large, usually a power-of-2. The successive phases are computed using a phase accumulator with an increment A_{inc}. The resulting output frequency has the following expression, where N is the width of the phase accumulator.

$$f_{out} = f_{in} \frac{A_{inc}}{2^N} \tag{4.32}$$

This equation shows that the output frequency depends linearly on the input frequency. Such a resampler architecture can generate 2^N equally spaced f_{out} in range $0 \cdots f_{in}$. The increment A_{inc} can be larger than 2^N, in which case the resampler performs downsampling and f_{out} is larger than f_{in}. In our case the input bandwidth is constant 0.25, while the output bandwidth can be varied between 0 and 0.5.

Unlike interpolation, such a resampler architecture requires the generation of output samples for a very large number of possible sub-sample phases. Storing coefficients for all these phases in impractical. As mentioned in **Subsection 4.5.2**, an efficient solution is to store coefficients for a predefined number of phases and linearly interpolate for all others. One of the aims of the present performance analysis is to investigate how the number of coefficients sets affects the resampling performance.

In the specific case of fading tap generation, resampling performance is defined in terms of spurious frequency components outside the desired Doppler bandwidth. Both the highest peak and the variance of these spurious components are of interest. In order to determine them, the testbench in **Figure 4.40** is used. The low-pass filter limits the noise bandwidth to 0.25 and has a very sharp cut-off and a flat response in the band-pass. It is an Elliptic filter of order 16, with pass-band ripple of 0.1 dB and stop-band attenuation of -120 dB, and is scaled so that the noise variance after filtering is 1. It is implemented as 8 cascaded second-order sections (bi-quads). The tough constraints force the poles very close to the unit circle, which makes a straightforward direct-form implementation impossible. Even with double-precision floating point coefficients, such a filter would be unstable.

The high-pass filter isolates only the desired out-of-band spurious components. Its constraints are identical to those of the low-pass filter. The cut-off frequency would be ideally the highest frequency in the output spectrum, f_{Dmax}. However, due to the finite width of the transition bands, the cut-off frequency is chosen to be $1.1 \cdot f_{Dmax}$. This will only introduce a small error when computing the variance of the spurious components.

One way to evaluate the purity of the interpolated signal is by spectral analysis of the out-of-band resampling artifacts. The power spectral density (PSD) is obtained using Welch's method [96] with an FFT window size of 4096 (2^{12}), which ensures a good trade-off between accuracy and computational efficiency. The PSD of the original signal signal with $f_{Dmax} = 0.25$ is shown in **Figure 4.41**, where the decaying tail is a side-effect of the spectrum estimation method.

The spectrum of the spurious components depends on the spectrum of the original signal and on the frequency response of the interpolation filter. **Figure 4.41** shows the spurious interpolation

artifacts in the case of a 6-tap filter, for an input bandwidth of 0.25 and an interpolation factor of 8. The corresponding frequency response of the interpolation filters is plotted on the same axes. It is now apparent why Lagrange interpolation is better than windowed sinc types for lower input bandwidths.

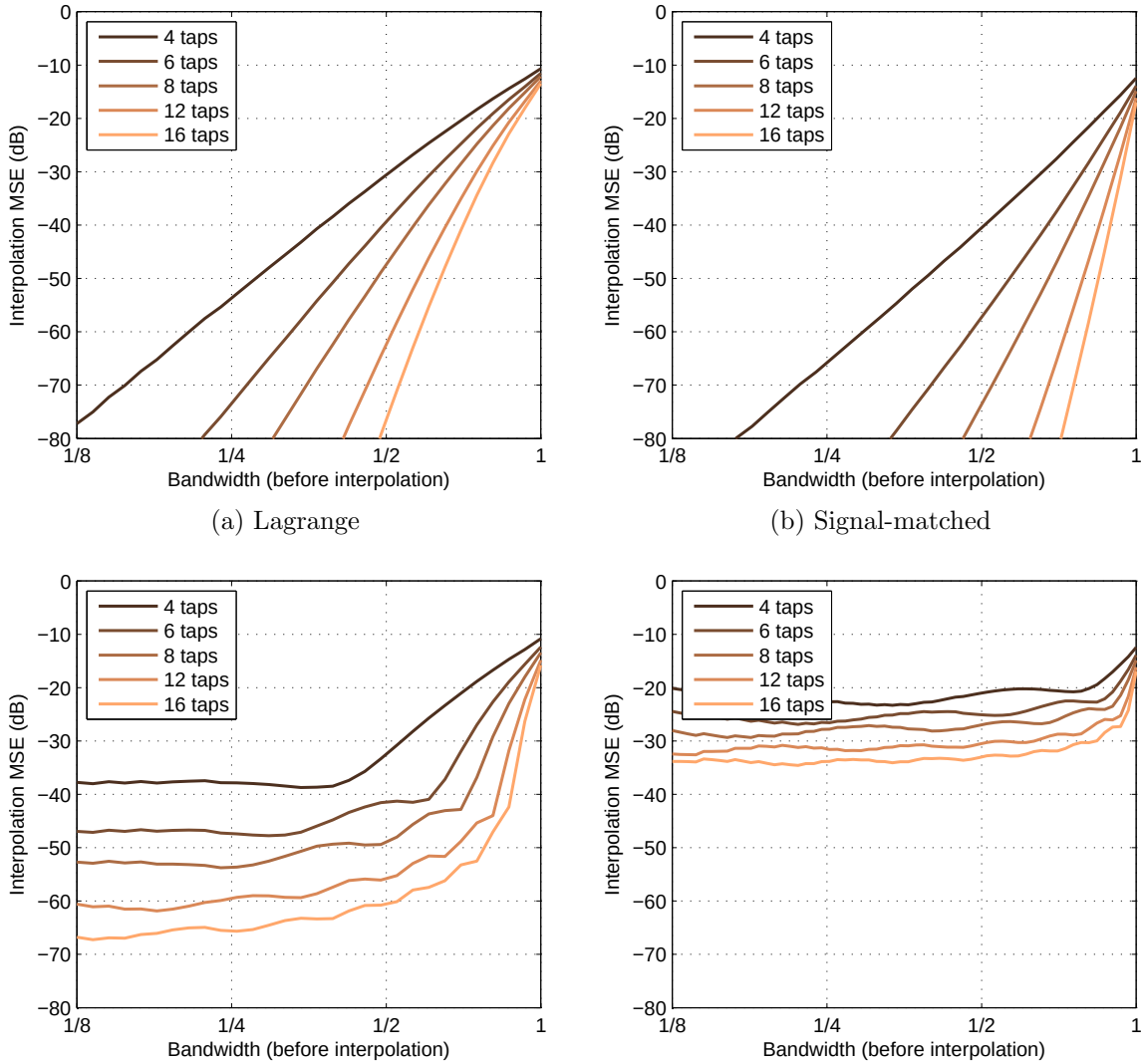
The results presented so far assumed a power-of-2 interpolation factor and exact filter coefficients. In reality, however, the upsampling factor is a rational number, which causes the intermediate sub-sample phase to take many different values. As the coefficients are computed by linear interpolation between a fixed number of coefficients sets, usually a power-of-2, this will contribute to the overall interpolation error. Our goal is to evaluate this error and to understand its variation with the interpolation factor. To this end we use the test-bench in **Figure 4.40** and measure the variance of the out-of-band components in a given range of interpolation factors.

As in the previous analysis, the input signal has a flat spectrum and a bandwidth of 0.25. The resampling factors are chosen so that the bandwidth of the resampled signal covers four octaves, between 1/16 and 1/2. **Figure 4.43a** shows the results for Lagrange interpolation with 4, 6, and 8 taps. The number of poly-phase coefficients sets has been chosen sufficiently large sufficiently large (256), so that the results are not affected by it. As expected, at bandwidths 0.25 (no resampling) and 0.5 (decimation by 2) the error is zero. The reason is that no intermediate samples are generated and the output consists of original samples only. The error is otherwise relatively constant, varying only slightly with the frequency.

Figure 4.43b shows the same analysis in the case of 4 coefficients sets. The results show that the linear interpolation errors create an additional error floor, so that the error for 6 and 8 taps is the same. Unlike the ideal case, the error depends now on the actual resampling factor. For resampling factors 1/2 and 1/4 the error reaches the ideal level in **Figure 4.43a** because the phase only takes values for which coefficients sets are precomputed and no linear interpolation is needed: 0, 1/4, 2/4, and 3/4.

As the coefficients need to be stored, the number of poly-phase sets directly affects the hardware complexity. This is less of an issue in software, were a few extra bytes of constant storage are easily available. In order to determine the optimum number of poly-phase coefficients sets we need to know how this number affects the error floor. The test configuration in **Figure 4.40** is also used in this case. We consider interpolation filters with 4, 6, and 8 taps respectively, for a resampling factor and output bandwidth at which no error minimum occurs, such as 0.1. **Figure 4.44** shows the resampling artifacts variance for Lagrange and signal-matched interpolation.

The conclusion that can be drawn is that the number of coefficients sets must be correlated with the filter size for a target interpolation performance. It can be seen that a 4-tap signal-matched interpolation filter with 8 coefficients sets offer excellent interpolation results, with a -70 dB variance of the out-of-band artifacts, relative to the variance of the signal. It can be also observed that for Lagrange interpolation with 4 and 256 coefficients sets the results are consistent with those presented in **Figure 4.43**.



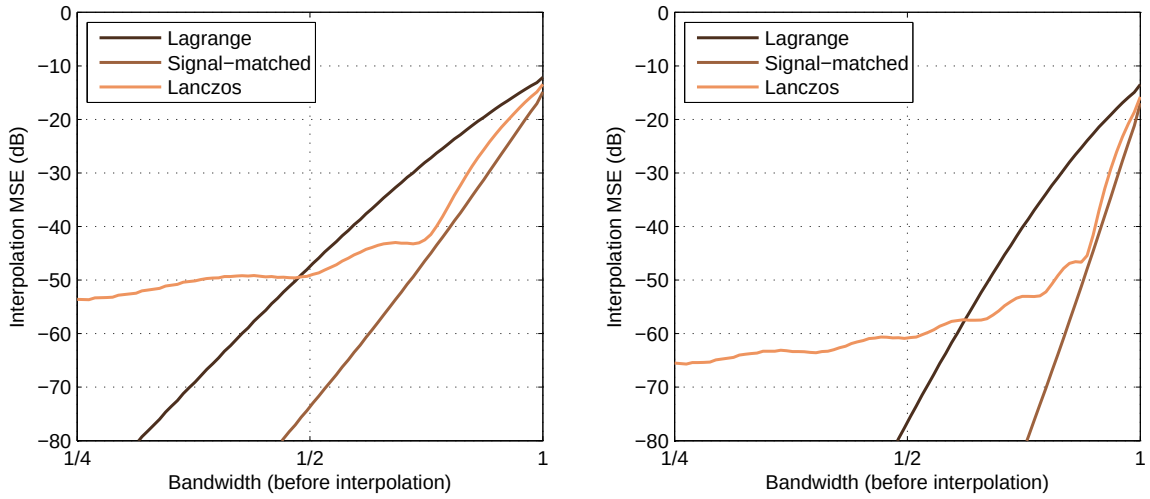
(a) Lagrange

(b) Signal-matched

(c) Lanczos

(d) Truncated sinc

Figure 4.38.: MSE vs. bandwidth for various interpolation filters



(a) 8 taps (b) 16 taps

Figure 4.39.: MSE vs. bandwidth for two interpolation filter sizes

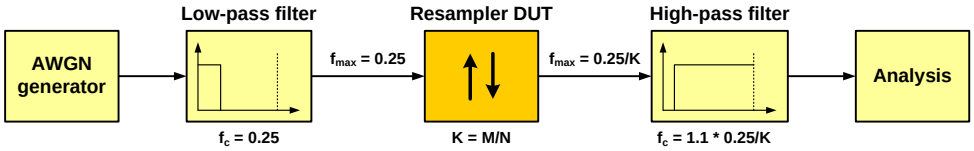


Figure 4.40.: Resampler spurious components measurement

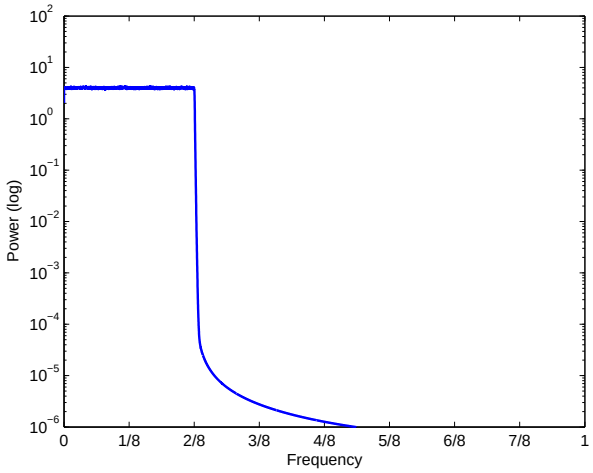
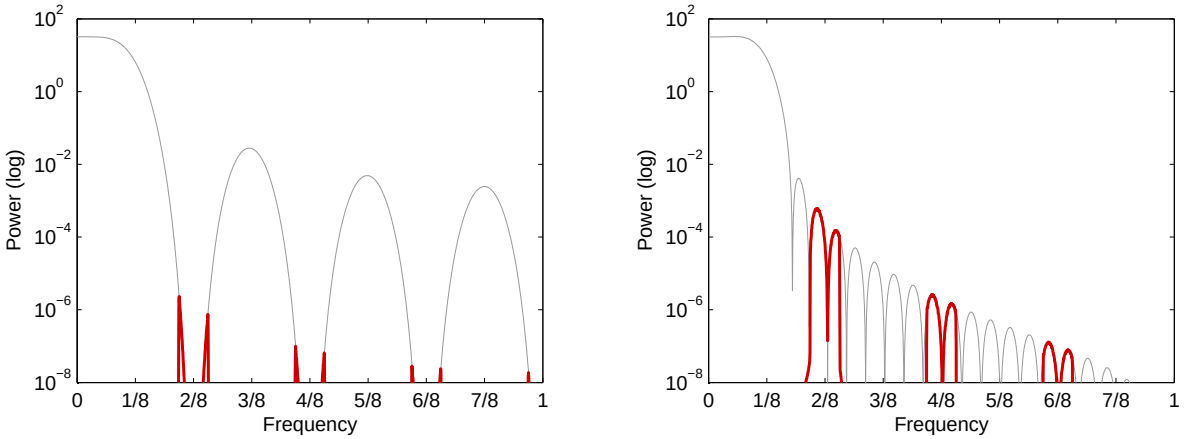


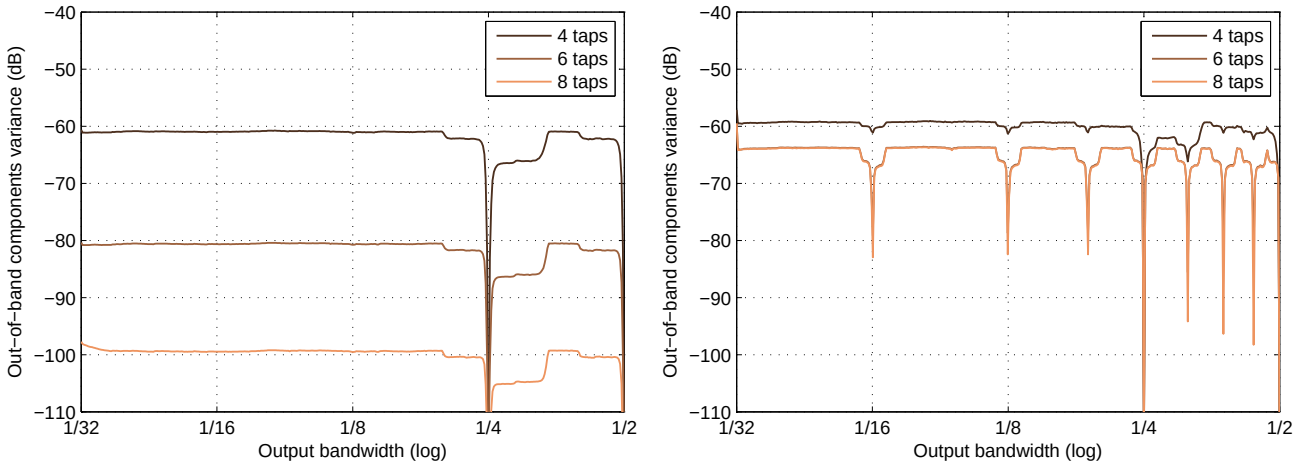
Figure 4.41.: PSD of the band-limited signal used for testing the resampler performance



(a) 6-tap Lagrange interpolation

(b) 6-tap Lanczos interpolation

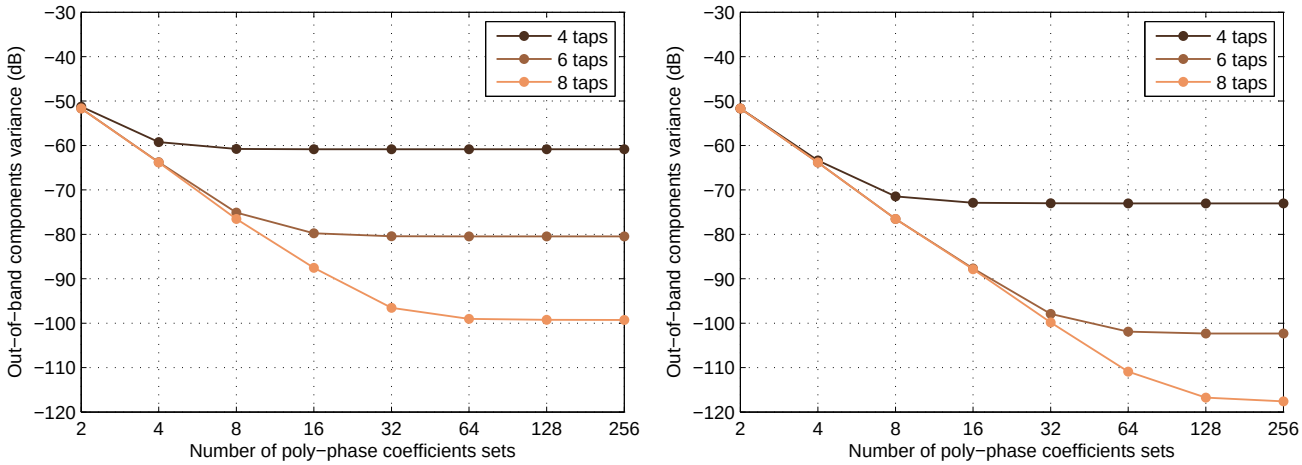
Figure 4.42.: Spectrum of the out-of-band interpolation artifacts



(a) exact coefficients

(b) 4 coefficients sets

Figure 4.43.: Resampling artifacts variance vs. output bandwidth



(a) Lagrange interpolation

(b) Signal-matched interpolation

Figure 4.44.: Resampling artifacts variance vs. number of coefficients sets