NTNU
Norwegian University of
Science and Technology

FPGA based noise reduction in video cameras

Thor Arne Stangeland Brandsvoll

Master of Science in Electronics
Submission date: June 2011
Supervisor: Per Gunnar Kjeldsberg, IET
Co-supervisor: Lars Aurdal, Cisco Systems Norway

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

FPGA based noise reduction in video cameras

Thor Arne S. Brandsvoll

June 20, 2011

NTNU
Department of Electronics
and Telecommunications

Problem description

Student: Thor Arne S. Brandsvoll

Title: FPGA based noise reduction in video cameras

Problem: Video cameras pose particular challenges when it comes to image noise. The combination of high frame rates (imposing short exposure times) with physically small pixels on the image sensor makes noise a definite problem. Adding to this situation, any noise reduction in the camera must happen in real time at the rate imposed by the frame rate of the camera. All these factors combine to make noise reduction in video cameras both algorithmically and computationally challenging.

The work will be a continuation of an autumn project and the candidate will study the state of the art in video noise reduction algorithms. Based on existing literature, the candidate will select one or several algorithms, simulate their performance and implement them on an FPGA in an existing video camera product. As part of the assignment, a comparison with the performance of the current version of the noise reduction algorithms in the product should be performed.

Supervisor: Lars Aurdal, Cisco Systems Norway
Professor: Per Gunnar Kjeldsberg, NTNU

FPGA based noise reduction in video cameras

Master's thesis, circuits and systems design, TFE4915

Thor Arne S. Brandsvoll
Department of Electronics and Telecommunications,
Norwegian University of Science and Technology

Abstract

Video noise is an important issue even in modern camera sensors. The trend of higher resolutions and more FPS makes real time video processing in general a difficult task. In this study, the goal was to find a fast video denoising algorithm, which can be implemented on an FPGA without using an exaggerated amount of its available resources. A selection of algorithms were therefore reviewed, varying from some of the most basic to some of the most acknowledged. One of them, the Yaroslavsky filter, was selected because of its simple approach to the well recognized method of using only the most similar and close neighbor pixels in the average and noise removal process. Three modifications to the original Yaroslavsky was proposed, and implemented in Matlab for simulations. The first, and maybe most important modification, was to extend the algorithm from the spatio, to the spatio-temporal domain. This modification makes the algorithm something more than a image denoising algorithm applied on each independent frame in a video. The temporal extension utilizes the correlation between pixels in successive frames. The second modification was to introduce fuzzy thresholds, instead of the binary thresholds in the original Yaroslavsky. This makes the algorithm more adjustable, so it can mimic the more advanced Bilateral filter. The third modification proposed, was to make the Yaroslavsky capable of removing impulse noise. The original Yaroslavsky filter would in case of impulse noise, not detect any similar neighbor pixels, and thus leave it alone. The proposed modification was to introduce median filtering in such cases. The modified Yaroslavsky algorithm have been tested in Matlab, and compared with the original Yaroslavsky, as well as with the algorithm proposed in the preliminary project work. The simulation results showed that the proposed modified Yaroslavsky achieved the best results. VHDL was therefore used making an FPGA implementation of the algorithm. The proposed implementation consists of five components, and has four pipeline stages. The implementation was simulated in Modelsim to ensure correct manner of operation. It was then synthesized for an Altera Cyclone III FPGA, using both Quartus and Synplify. The highest clock frequency achieved was 87.7MHz, using 1044 logic elements, 345 registers, and 5 DSP blocks.

Preface

This master's thesis was written as the finishing part of my Master of Science degree in electronics, with specialization in digital electronics, at the Norwegian University of Science and Technology in Trondheim. The purpose of this thesis is to test and propose a video denoising method suitable for running in real time on an FPGA.

I would like to thank my supervisors Post Doc. Lars Aurdal at Cisco Systems Norway and Professor Per Gunnar Kjeldsberg at the Department of Electronics and Telecommunications, for guidance throughout the semester. I would also like to thank the rest of the camera group at Cisco Systems Norway for two fine weeks with close follow up at their offices in Lysaker. Last but not least, i would like to thank my fellow student Erik Strømme for numerous of valuable discussions.

Trondheim, June 20, 2011

Thor Arne S. Brandsvoll
brandsvo@stud.ntnu.no

Contents

1 Introduction

Video cameras have become very popular over the last years, even cell phones usually have video cameras integrated. With High-Definition (HD) video becoming popular, one might think that video noise does not longer exist. Quite the contrary, the hunger for higher resolution has caused the physical pixel size in camera sensors to shrink, making them more vulnerable for noise. Noise reduction has therefore been a topic of intensive research. The research has led to several good de-noising algorithms for both still pictures and video, which suffers from the same problem of pixel scaling. Not all of these denoising algorithms are suitable for Video Conferences. An important requirement in such systems is low latency, as a few hundred milliseconds in total delay makes the conversation flow jagged and unnatural. This real time constraint is particularly hard to meet for HD video, as more pixels must be processed, excluding the most complex algorithms. The algorithm must also be suitable for an FPGA with limited available resources, as it also contains other parts of the video processing pipeline. The main contributions in this work is modifications to the Yaroslavsky filter with belonging Matlab and VHDL implementations and simulations.

The remainder of this thesis is structured as follows. Fundamental principles of noise and video are given in Chapter 1. A review of denoising algorithms, from the most basic to the important new developed, is given in Chapter 2. A summary of the algorithm developed in the initial project work is given in Chapter 3. Modifications to a well known algorithm, the Yaroslavsky filter, is given in Chapter 4. The project work algorithm is compared with the original and the modified version of the Yaroslavsky filter with Matlab simulations in Chapter 5. A hardware implementation of the modified Yaroslavsky algorithm is proposed in Chapter 6. The work is discussed and concluded in Chapter 7 and 8. Future work is discussed in Chapter 9.

1.1 Noise

Noise is unwanted imperfections and errors in images produced by photo and video cameras. Poor image quality does not only degrade visual quality, but also affects sub-processing like compression. There are many different types of noise, and even more noise causes. Noisy images have always been a problem, and several de-nosing algorithms have therefore been proposed. The problem with many of these algorithms is that they are made to remove only one type of noise. In the real world, images are affected by several noise types at the same time. Another problem with many algorithms, is that they, in addition to removing noise, also blur the image, or can cause ghosting. This is especially a problem when the video contains fast movements. Besides, all the problems are even more severe in a real-time scenario like a video-conference. At a scenario like this, the

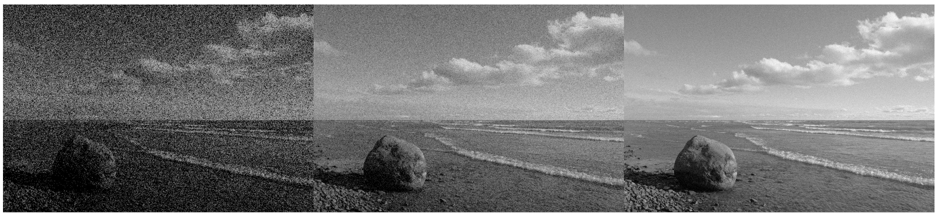


Figure 1.1: *Photon Shot Noise. From left to right:* λ *= 1, 10, 100 [3]*

algorithm must be run in real-time, a speed set by the number of frames per second.

1.1.1 Noise sources

Today's image sensors are mainly built using two different technologies, CMOS (Complimentary Metal-Oxide Semiconductor) and CCD (Charged-Coupled Device), with CMOS rapidly becoming the dominant technology. Some of the most important noise sources existing in both technologies [1, 2] are:

Noise sources

- **Photon shot:** Because light is quantified by nature, there exists an unavoidable uncertainty in the number of photons collected by each pixel. This is an increasing problem with the shrinkage of pixel size. The number of photons is known to follow the Poisson distribution, Equation 1.3, where λ is the expected number of photons for a given exposure time. For a large number of photons, the Poisson distribution comes close to the Gaussian distribution, Equation 1.1, so the photon shot noise can in good light conditions be modeled as White Gaussian Noise (WGN).

- **PRNU (Photo-Responsive Non-Uniformity):** The pixels in the image sensor have different sensitivity to light, simply because of process variations. Producing identical pixels becomes harder and harder with the technology scaling. PRNU is therefore also an increasing problem with the shrinkage of pixel size.

- **DCNU (Dark-Current Non-Uniformity):** Regardless if photons are collected by the pixel or not, a small current is produced due to random generation of free electrons. Different pixels produce a different amount of so called dark current, and the amount generated is highly temperature dependent.

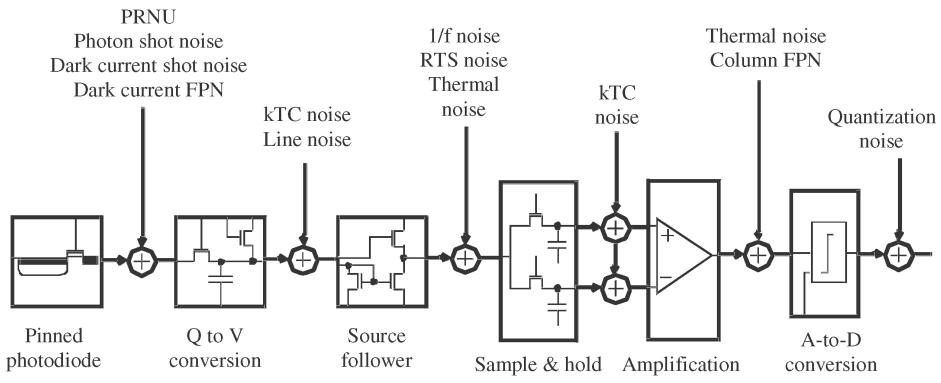


Figure 1.2: *Main sources of noise in a CMOS imaging pipeline (4T pixels) [2]. Most of them are not examined further in this work.*

- **Pixel defects:** Some pixels generate extremely high levels of dark-current. So called hot pixels will rapidly saturate, and end up as a white dot in the image. Some pixels will always read the minimum value, and are called "Dead pixels". They end up as a black dot in the image [4].

There are also a lot of variables affecting the amount of noise produced. Some of the most important factors are:

Noise affecting variables

- Light conditions
- Motive
- Exposure time
- Sensor temperature

1.1.2 Noise modeling

Real image noise models often consist of simple models added together. Some important noise models are given next.

Noise models

- **White Gaussian:** This is a widely used model. It is even used when it is only marginally applicable, because of its mathematical tractability in both the spatial and the frequency domain. The spectrum of a Gaussian random variable is a constant, containing all frequencies in equal proportions. The term white noise, is a carryover from the term white light, which contains nearly all frequencies of the visible light in equal proportions. The probability density function (PDF) of a Gaussian random variable, z, is given by Equation (1.1), and shown in Figure 1.3.

$$p(z) = \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{-(z-\mu)^2}{2\sigma^2}} \tag{1.1}$$

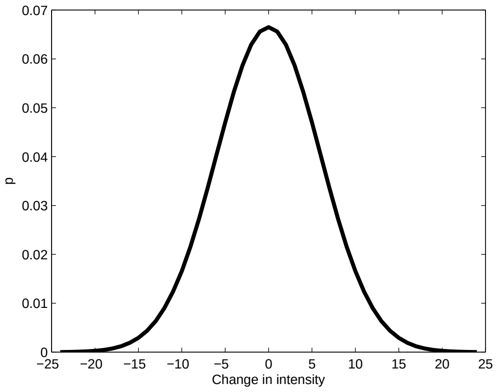


Figure 1.3: *Gaussian distribution with standard deviation, $\sigma_n = 6$, and mean, $\mu_n = 0$*

Approximately 70 % of the pixels will not differ more than one standard deviation, σ_n, from the correct pixel value.

- **Impulse:** Pixels affected by impulse noise have the same pixel value in every frame, independent of the true pixel value. This value is typically saturated, e.g. 255 or 0 in an 8-bit image caused by dark current or a pixel defect. The pixel can get different colors, depending on the pixel placement in the color filter array (see Chapter 1.3). The PDF is given by Equation (1.2) and shown in Figure 1.4.

$$p(z) = \begin{cases} P_a & \text{for } z = a \\ P_b & \text{for } z = b \\ 0 & \text{otherwise} \end{cases} \tag{1.2}$$

If P_a and P_b are approximately equal, and a is a high and b is a low value, the impulse noise is sometimes referred to as salt and pepper noise, because

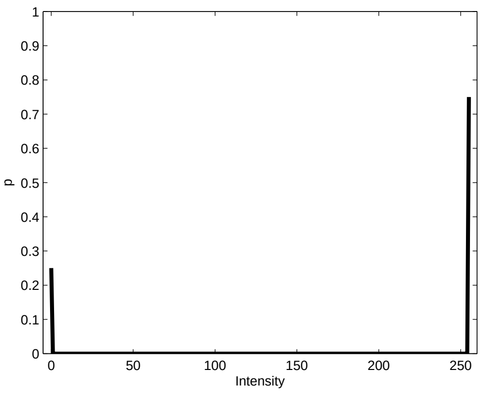


Figure 1.4: *Impulse distribution with $P_a = 0.25$ for $a = 0$, and $P_b = 0.75$ for $b = 255$*

of the white and black spikes it will cause in a gray scale image. If P_a or P_b is 0, the impulse noise is called unipolar. If none of them are zero, the impulse noise is called bipolar. Examples of Gaussian and impulse noise are shown in Figure 1.6.

- **Poisson:** Useful for modeling Photon Shot Noise. The Probability Mass Function of the Poisson distribution (PMF) is given by Equation (1.3) and shown in Figure 1.5.

$$p(k, \lambda) = \frac{\lambda^k e^{-\lambda}}{k!} \tag{1.3}$$

1.2 Denoising approaches

There are roughly two different methods of denoising. The first and most straightforward method is to approach the problem in the standard pixel domain. An early review of pixel domain methods are given in [6], where noise filters for dynamic image sequences (of which video sequences is a special case) are divided into four groups. The second method is to approach the problem in a transform domain. A popular transform domain is the wavelet coefficient domain. The focus in this work will be on pixel domain methods, as they are most used in practice.

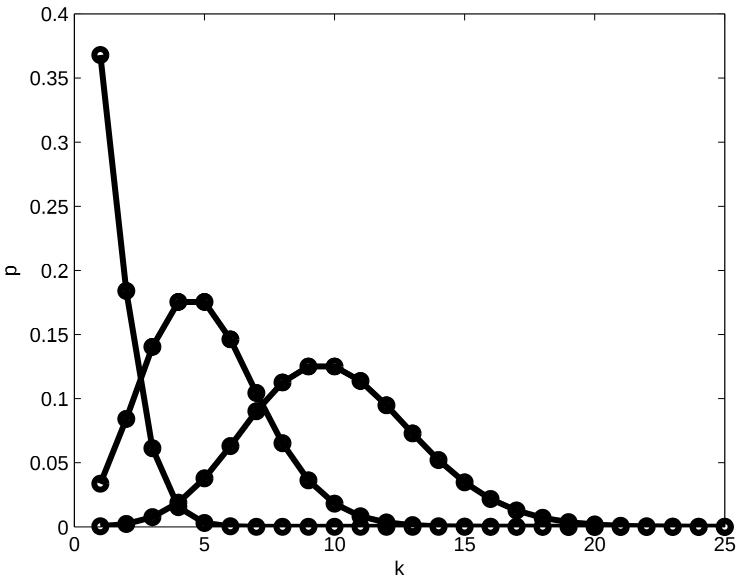


Figure 1.5: *Poisson distribution with λ = 1, 5, 10*

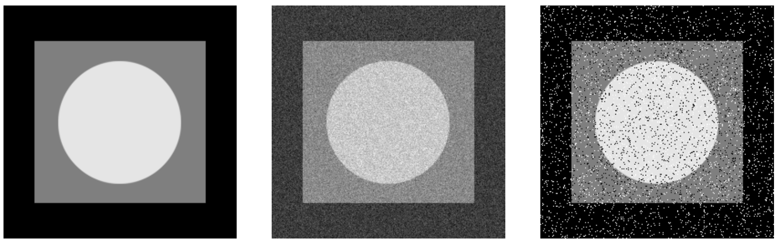

(a) Original pattern (b) Gaussian noise (c) Impulse noise

Figure 1.6: *Gaussian and impulse noise [5]*

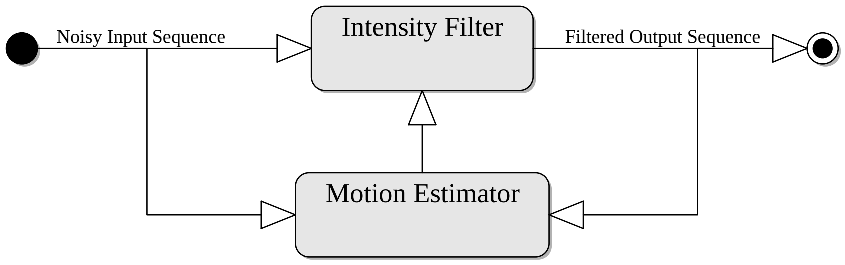


Figure 1.7: *Standard approach for motion estimation [6]. Both previous filtered images, and future images can be used for estimating motion. The filtering intensity is lowered if motion is detected*

1.2.1 Pixel domain

Types of pixel domain denoising filters

- Temporal (1D)

  Motion compensated

  Non-motion compensated

- Spatio-temporal (3D)

  Motion compensated

  Non-motion compensated

Spatial (2D, static) is also a third group in the pixel domain, but these filters are not exploiting correlation between successive images. Each image is therefore considered independent, similar to a photograph. Temporal and spatio-temporal noise filters therefore yield better results on video de-noising. Another name for spatial filtering is 2D filtering, because the image is considered a collection of pixels along two spatial axes. In temporal filtering, each pixel is considered as a 1D signal, that transverses along the temporal (time) axis, and only correlates with pixels on this axis, and not with neighbor pixels (2D) in the same frame. Not surprisingly, spatio-temporal filtering utilize both the two spatial axes, and the temporal axis (3D). 1D and 2D filtering can therefore be considered as special cases of 3D filtering.

Fast motions will often be blurred or ghosted when a noise filter is added. To avoid this, several denoising algorithms implement motion detection, so the intensity of the filter can be reduced in areas with heavy motion. The quality of the noise filter depends on the accuracy of the motion estimator, which again relies on the SNR (Signal-to-noise ratio) in the video. After testing several noise filters in the different categories, the conclusion in [6] is that no filter can be singled out as the solution for all applications.

1.2.2 Transform domain

Transform domain denoising methods first transforms the signal to another domain. De-noising is then applied in this domain, followed by an inverse transform bringing the signal back to the pixel domain. A popular transform domain is the wavelet coefficient domain, where the different coefficients are associated with frequencies and time. It is not a complete transform into the frequency domain as in the Fast Fourier Transform (FFT). The noise coefficients have small values compared to the signal coefficients, and all the coefficients lower than some threshold can therefore be cancelled out, and the video can be reconstructed in the inverse transform without the noise [7]. Other transform domains used for denoising includes the discrete cosine transform (DCT) [8], first and second generation bandelet [9], curvelet [10], contourlet [11], and ridgelet [12] transforms. Hybrids also exists, using methods both in a transform domain, and the pixel domain. There have been carried out a lot of research on transform domain algorithms lately. It is however clear that most of them are developed for image processing and that algorithms in the pixel domain is most used in practice. Because transform domain methods involves a transform and a reverse transform, a pixel domain method also seems to be the most cost effective choice for a real time video application.

1.3 Obtaining colors

A common cost effective way of capturing color information is by laying a *Color Filter Array* (CFA) over the sensor. Pixels will then get different responses to different colors (spectral bands) in accordance with the CFA. The array is often composed of Red, Green, and Blue (RGB) color filters, although other choices of primary colors exist. A popular arrangement of an RGB colored CFA is the *Bayer pattern*. The Bayer pattern consists of 50% green and 25 % red and blue, as shown in Figure 1.8. More green is used to better collaborate with the response of the human eye, Figure 1.13. The color image is then reconstructed by estimating the pixels missing color components using the neighbor pixels in an operation called *demoisaicking*. This is important knowledge when working with noise reduction. An important question is whether to apply the noise reduction before or after demosaicking, as demosaicking will interfere with the original sensor noise. Some algorithms even propose to join demosaicking and noise removal [13], as they are both estimating problems. Most denoising algorithms however, assume the data to be demosaicked. A simple way of extending some of these algorithms to Bayer data, is to let them work only with pixels of the same color, although spatial

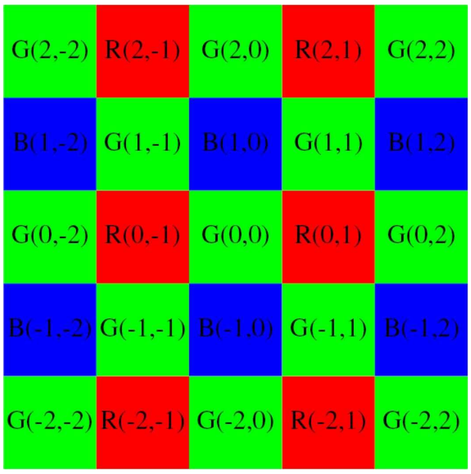


Figure 1.8: *Bayer pattern. 50 % green and 25 % red and blue*

redundancies between different colors exist and is therefore not taken advantage of. [13, 14]

1.4 Video quality measurement

To compare different denoising algorithms, a standard quality measurement is needed. When the result is to be viewed by humans, the best measurement is actually our subjective evaluation. Psychological experiments where a number of viewers watch a set of videos and rate their quality can be arranged, and a Mean Opinion Score (MOS) can be derived. This method however, is inconvenient and time consuming. In addition, the viewers can have a different Quality of Experience (QoE) [15]. QoE is affected by several conditions, including viewers focus of attention and video experience, display type and properties, viewing distance and exterior light and so on. A good mathematical quality measurement should predict the subjective human perception of the video. In addition to benchmarking, a quality measurement can be used to optimize denoising algorithms and its parameters. There are three different types of quality benchmarks.

Quality measurement types

- Full reference
- Reduced reference
- No reference

Most existing methods are full reference, meaning noise free reference data exists. These methods can do a frame by frame comparison with the denoised version of the nosy video, and the noise free video. Noise free artificial videos can easily be made, and noise can be added to make a noisy copy of the video. Adding a realistic amount of noise requires an advanced noise model, see Chapter 1.1.2. Noise free real world data however is hard to obtain (if it was not, this work would be superfluous). Often artificial noise is added to real world videos that are assumed to be noise free. A scenario where full reference data usually exists, is in video compression, where the compressed video can be compared with the uncompressed video.

Reduced reference is when the reference is only partially available, represented as a number of features. Examples are motion or spatial details.

No reference is when no reference data exists, for instance when the video comes straight from a video camera. These methods are flexible, but have a large drawback: The methods can have a hard time distinguishing between noise and video details. It turns out that designing a no reference metric is a very hard task [16]. The following methods are full reference measurement types, and some of the most commonly used quality measures in video denoising.

1.4.1 Mean squared error

The Mean Squared Error (MSE) between two images is the average of the squared differences in pixel intensity. The definiton is given in Equation 1.4.

$$MSE = \frac{1}{mn}\sum_{i=1}^{m}\sum_{j=1}^{n}[I_{\text{ref}}(i,j) - I(i,j)]^2 \tag{1.4}$$

Where I_{ref} is the intensity of the reference image. The Root Mean Squared Error (RMSE) is also used, which is the root of the MSE, Equation 1.5

$$RMSE = \sqrt{MSE} \tag{1.5}$$

1.4.2 Signal to noise ratio

A measure often used in signal processing is the Signal to Noise Ratio (SNR), which measures the power of a signal compared to the power of noise. SNR is also a useful metric in image processing. Assuming white noise, the SNR can be defined as the standard deviation of the image intensity (σ_i), divided by the standard deviation of the noise (σ_n).

$$SNR_{\text{white noise}} = \frac{\sigma_i}{\sigma_n} \tag{1.6}$$

The standard deviation of a gray scale image is defined in equation 1.7.

$$\sigma_i = \sqrt{\frac{1}{mn}\sum_{i=1}^{m}\sum_{j=1}^{n}[I(i,j)-\mu_i]^2} \tag{1.7}$$

Where μ_i is the average gray value, Equation 1.8.

$$\mu_i = \frac{1}{mn}\sum_{i=1}^{m}\sum_{j=1}^{n}I(i,j) \tag{1.8}$$

A more general definition can be made using the reference data. Expressed in decibels, this definition is given in Equation 1.9.

$$SNR_{dB} = 20\log_{10}\left(\frac{RMS}{RMSE}\right) = 10\log_{10}\left(\frac{RMS^2}{MSE}\right) \tag{1.9}$$

Where RMS is the Root Mean Square value, Equation 1.10.

$$RMS = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{n}[I(i,j)]^2} \tag{1.10}$$

The human brain is an amazingly good denoiser. We are able to see all the image details even when white noise degrades the image down to an SNR of 2. This is shown in Figure 1.9. The brain can de-noise video even better. The higher FPS, the more details are visible as the brain can be said to have a built in temporal denoising filter.

1.4.3 Peak signal to noise ratio

The most used quality metric used for video noise is perhaps the Peak Signal to Noise Ratio (PSNR). PNSR is mathematically just a logarithmic presentation of the MSE. The definition is given in Equation 1.11.

$$PSNR_{dB} = 20\log_{10}\left(\frac{I_{\max}}{RMSE}\right) = 10\log_{10}\left(\frac{I_{\max}^2}{MSE}\right) \tag{1.11}$$

Where I_{max} is the maximum theoretical value of the intensity, 255 in an 8 bit image. For a video sequence, the final PSNR value is calculated as the mean of the PSNR of all the frames. Even though PSNR is expressed in decibels, it is dimensionless as both the numerator and the denominator are pixel values. PSNR is popular for several reasons. It is fast to compute, and simple to understand. Researchers have also developed a familiarity with it over the years. PSNR has

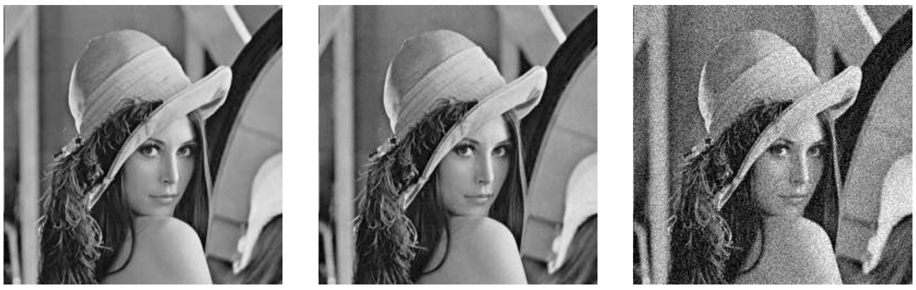

(a) Original (b) SNR = 18.3 (c) SNR = 2.2

Figure 1.9: *The original image has a standard deviation of $\sigma_i = 55$. White noise with $\sigma_n = 3$ is added to the middle image, giving 18.3 in SNR. White noise with $\sigma_n = 25$ is added to the right image, giving 2.2 in SNR. Note that all the details are still visible in the right image [17].*

also been criticized as a video quality metric, because it does not necessarily reflect the perceived quality. This is shown in Figure 1.10. PSNR does not take spatial relationship between pixels into account, and is just a byte by byte comparison. It is also meaningless to say that a PSNR value of e.g 30 is good. PSNR should be used only to compare different algorithms on the same data, as demonstrated in Figures 1.11 and 1.12.

An important reason why PSNR is still used as a quality metric is the lack of other good metrics. A better metric should take how the Human Visual System (HVS) works into account. [15, 16, 18–21]

1.4.4 Method noise

A method for comparing no reference videos, is to look at what the algorithm removed from the original noisy data. The removed data of two algorithms is called method noise, and can be analyzed and compared. Optimally, the method noise should not look like anything else than noise. This would mean that the algorithm did not remove important details from the image. The removed noise can therefore help to compare algorithms, in addition to comparing the result itself. [17]

1.5 The human visual system

The Human Visual System (HVS) is an advanced system, where a lot of research have been done. It is important to know how the HVS work, so denoising algorithms and quality metrics that better suit the human perception can be created.

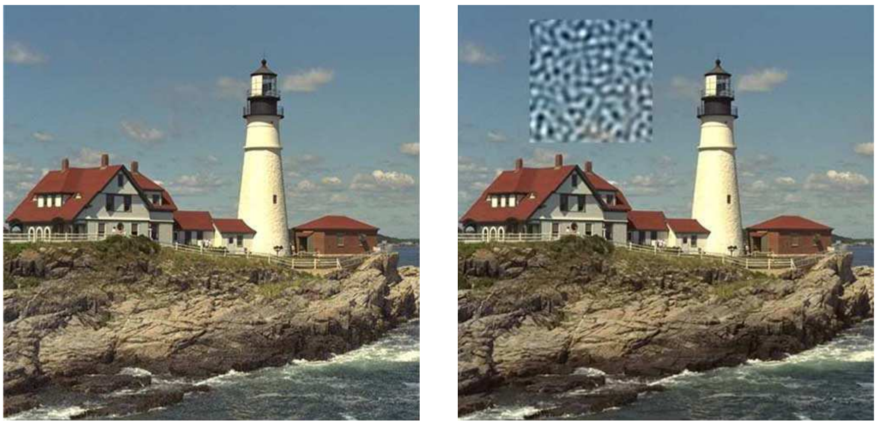

(a) High frequency noise (b) Low frequency noise

Figure 1.10: *Both image a and b have the same PSNR. The perceived quality is however very different. High frequency noise is inserted in the bottom of image a. This noise is camouflaged by all the details in the rocks and the sea. Low frequency noise is inserted in the top of image b. The distortions stands out from the smooth sky. [15]*

The human eyes are like self cleaning real time cameras with auto focus and adaptivity to light. They have two types of photo receptors, cones and rods. The rods are what makes us able to see in the dark. The cones are used in good light conditions, and makes us able to see colors and fine details. There are three different types of cones, L-cones, M-cones and S-cones with peak sensitivity at long (570nm), medium (540nm) and short(440nm) wavelengths, roughly corresponding to red, green, and blue. There are approximately 100 million rods and 5 million cones in the eye, where M-cones and L-cones accounts for the majority of them.

The rods and cones are not uniformly distributed. The highest density of cones is in the middle of the retina, causing a higher resolution and better capability to see details and color where the observer fixates. This spot is called the fovea and is about 0.5 mm in diameter. This could be bad news for denoising in video conferences where a typical scenario is a face in front of a static background. The observer will fixate on the face, and will therefore be more sensitive for noise in this area. Unfortunately, the face, or the mouth, will often be moving so that temporal filtering cannot be applied, causing more noise to be visible in the area where the observer is the most sensitive for noise. A contrast sensitivity function (CSF) is given in the Appendix, Figure A.1, and shows what contrast is needed to distinguish small spatial differences. The HSV spatial contrast sensitivity peaks

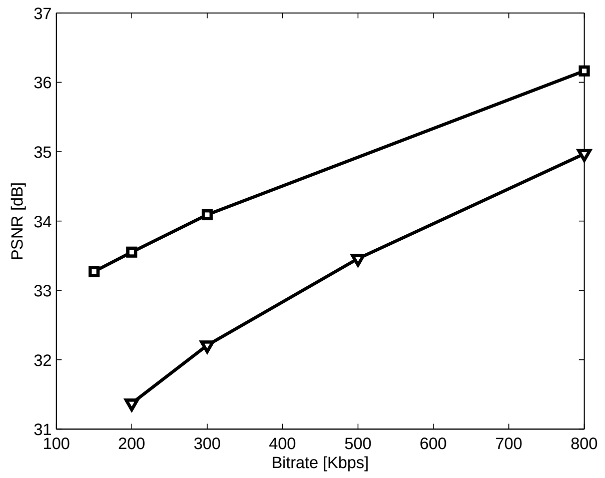


Figure 1.11: *PSNR and bitrates for a compression algorithm used on two different data sources. According to the PSNR values, the quality of the video with the squared markers is always better than the quality of the video with the triangle markers for all bitrates. [18]*

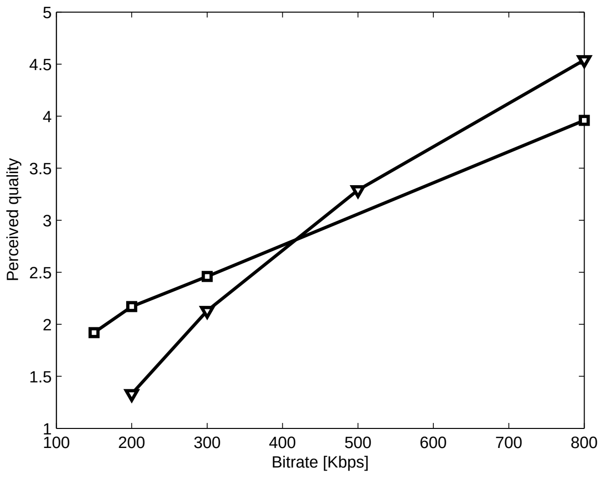


Figure 1.12: *Perceived quality and bitrates for the same compression algorithm used on the same data as in Figure 1.11. The perceived quality is better for the video with the squared markers on low bitrates, and better for the video with the triangle markers on higher bitrates. This is not what the PSNR suggested in in Figure 1.11. This shows that PSNR should not be used to compare algorithms over different video data. [18]*

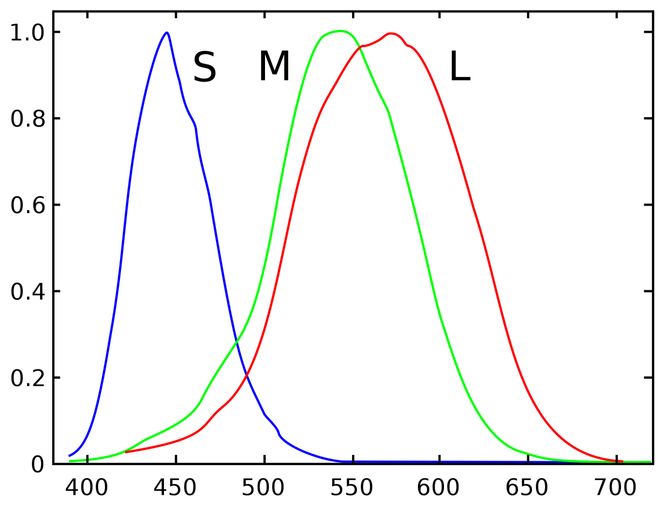


Figure 1.13: *Human cone response curves. The eye is very responsive to wavelengths corresponding to green light, which is partially covered by all three types of cones. This knowledge helped developing the Bayer pattern. [22]*

at 3 cycles per degree (CPD), and declines more rapidly at higher than lower spatial frequencies. Spatial frequencies higher than 40 CPD is undetectable for humans, even at the highest contrast. Similar graphs for chroma (color) components would have shown a lower sensitivity peak than for the luminance. If the color space used in the video is of this type, the most important plane to denoise is the luminance plane, e.g the Y-plane in YUV video. This is also the reason why the chroma components (U and V) are allowed reduced bandwidth, e.g. 4:2:2 chroma subsampling. [16]

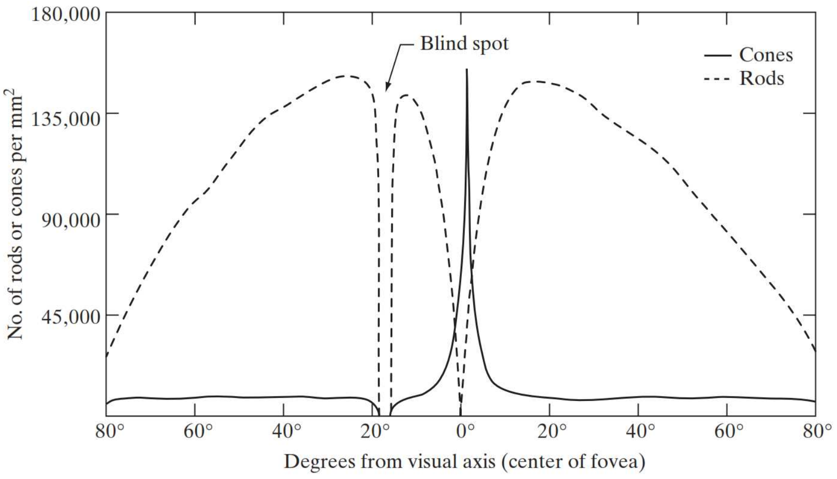


Figure 1.14: *The distribution of rods and cones. The fovea is in the middle of the retina and have the highest density of cones. Humans therefore have higher resolution where they fixate. [5, 23]*

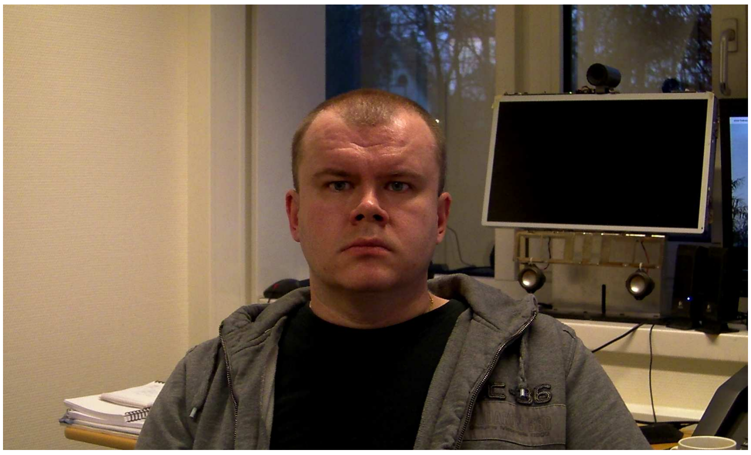


Figure 1.15: *A typical video conference scenario. Two people will fixate on each other's faces, and will perceive it with larger resolution. This is also where movement is most likely to occur (moving lips, eyes...), so temporal filtering cannot always be applied. Hence, there will be more noise present, where it is also easiest perceived. Hand gestures are also likely to be fixated on, and suffers from the same problem.*

2 Previous work

The algorithms presented are some of the most well known in the field of denoising. These are algorithms also used in image processing, and are presented this way for simplicity. They do operations in a sub image, with a size set by the pixel neighborhood. The sub image has several names like kernel, mask, filter or window. The kernel is moved over the image, so that the whole image is processed. When the kernel is linear, e.g., average or Gaussian filtering, this process is similar to a convolution, which can easily be computed in the Frequency domain. The kernel is therefore sometimes referred to as a convolution mask, even if it is not a linear kernel, e.g., median or bilateral filtering. The algorithms can generally easily be extended for image sequences, as shown in Chapter 2.9.

2.1 Average

The most basic noise removal technique is average filtering. Most noise removal algorithms involves some sort of averaging, and average filtering is the most pure averaging technique of them all. It replaces all pixels with an average of surrounding pixels. The number of pixels that are averaged can vary. A standard number is 9, where a 3x3 square of pixels is used. Another standard number is 25, where a 5x5 mask is used. Masks with an even number of pixels on the side, like 4x4, can hardly be used as they do not contain a middle pixel with surrounding neighbor pixels. An example of the weights in a 3x3 mask is given in Matrix 2.1. The shape does not need to be squared, and another popular shape is a diamond shape. A 5x5 diamond shaped is shown in Matrix 2.2. It is important that all the weights add up to 1 so the result is not biased. If the sum of the weights were higher than 1, it would result in a brighter image, and a darker image if the sum was lower than 1. In pure average filtering, all the weights of the closest pixels are the same, and the pixels in the rest of the image are 0. [5]

$$W = \frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix} = \begin{bmatrix}0.1111 & 0.1111 & 0.1111\\ 0.1111 & 0.1111 & 0.1111\\ 0.1111 & 0.1111 & 0.1111\end{bmatrix} \tag{2.1}$$

$$W = \frac{1}{13}\begin{bmatrix}0 & 0 & 1 & 0 & 0\\ 0 & 1 & 1 & 1 & 0\\ 1 & 1 & 1 & 1 & 1\\ 0 & 1 & 1 & 1 & 0\\ 0 & 0 & 1 & 0 & 0\end{bmatrix} \tag{2.2}$$

2.2 Median

Median filtering replaces the middle pixel with the median of its neighbors. Extreme values caused by impulse noise will effectively be removed, without having any effect on the filtered result. Median filtering is therefore better than averaging for removing impulses, because an extreme value could have a great effect on the average. It can be computationally heavy, as finding the median involves sorting of values. But optimizations can be done, e.g., only sort the lowest half of the values to find the middle value. Consider the example of intensity values in Matrix 2.3.

$$\begin{bmatrix} 73 & 73 & 124 \\ 74 & 120 & 122 \\ 79 & 119 & 255 \end{bmatrix} \tag{2.3}$$

The sorted result is 73, 73, 74, 79, $\underline{119}$, 120, 122, 124, 255 and the median is 119. 119 will then be the new median filtered value of the middle pixel (with the old value 120). The impulse, 255, does not have any effect on the result. The edge between the left 70- and the right 120- values is also kept sharp. [5]

2.3 Gaussian blur

Gaussian blur or Gaussian smoothing was proposed by Dennis Gabor in the sixties. He was the winner of the Nobel prize in physics in 1971 for the invention of optical holography, and he has been awarded the IEEE Medal of honor. A Gaussian filter is a filter whose impulse response, h, is a Gaussian function, Figure 1.3, where σ_s controls the amount of blur. Gaussian filtering in images, Gaussian blur, replaces a pixel with a weighted average of surrounding pixels, where the weights are based on the distance from this pixel. Because an image is a collection of discrete values, the weights are based on a discrete approximation to the Gaussian function. In theory, all the pixels in the image would contribute to this average, but the values are usually calculated in a small mask with 3x3 or 9x9 pixels. As σ_s becomes large, this mask approaches the average mask, where all pixels have equal weight. Matrix 2.4 shows an example of a Gaussian mask with size 3x3, and $\sigma_s = 0.5$. The matrix is symmetrical, and the weights decrease with the distance from the middle, and the values add up to 1. [17, 24]

$$W_s = \begin{bmatrix} 0.0113 & 0.0838 & 0.0113 \\ 0.0838 & 0.6193 & 0.0838 \\ 0.0113 & 0.0838 & 0.0113 \end{bmatrix} \tag{2.4}$$

2.4 Anisotropic

Gaussian blur filters equally in all directions, and is therefore isotropic. It will smooth out edges, as it will smooth out plane areas. Anisotropic (Gaussian) filtering tries to avoid this by smoothing only in the direction orthogonal to the gradient. The gradient represents where the image goes from low to high values (dark to bright), i.e., an edge. Anisotropic filtering thus uses samples from the edge for averaging the edge itself. A simple approximation of the image gradient can be found by the well known Sobel operator, using Matrices 2.5, to detect the changes in the x and y direction respectively.

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \tag{2.5}$$

Other gradient operators also exist, e.g., a recently proposed noise robust operator by Pavel Holoborodko, Matrices 2.6.

$$G_x = \begin{bmatrix} -1 & -2 & 0 & 2 & 1 \\ -2 & -4 & 0 & 4 & 2 \\ -1 & -2 & 0 & 2 & 1 \end{bmatrix} \quad G_y = \begin{bmatrix} -1 & -2 & 1 \\ -2 & -4 & -2 \\ 0 & 0 & 0 \\ 2 & 4 & 2 \\ 1 & 4 & 1 \end{bmatrix} \tag{2.6}$$

The gradient magnitude and direction can then be found by equation 2.7 and 2.8.

$$|G| = \sqrt{{G_x}^2 + {G_y}^2} \approx |G_x| + |G_y| \tag{2.7}$$

$$\theta = \arctan\left(\frac{G_y}{G_x}\right) \tag{2.8}$$

[5, 17, 25, 26]

2.5 Yaroslavsky

Average filtering smooths out noise by taking the average of neighbor pixels. Yaroslavsky filtering also calculates the average of neighbor pixels, but only if they have a similar intensity. It defines two neighborhoods, one with all pixels close in distance, the other one with pixels close in intensity. The average of the pixels existing in both neighborhoods is then calculated. The distance neighborhood is actually just a defined maximum border of the intensity neighborhood. Yaroslavsky proposed two different ways of finding the pixels with similar intensity, either by K-Nearest-Neighbors (KNN), where k usually is 5 or 6 in a 3x3 mask, or by all the neighbors with a difference not exceeding an upper and lower lower bound. In the first case, the neighborhood was called KNV, for K Nearest Values. In the latter case, it was called EV for Evaluated Values. The Yaroslavsky filter can therefore be summarized as the average of the KNV-neighborhood or the average of the EV-neighborhood. The KNV implementation is the most robust of the two implementations, but finding the KNV-neighborhood involves sorting, and is therefore a lot more computationally expensive than finding the EV-neighborhood. [17, 27]

2.6 Bilateral

Bilateral filtering is a developed version of the Yaroslavsky neighborhood filter, where weights are also assigned to the neighbors dependent on how close they are in both distance and intensity. The bilateral filter was first invented by Smith and Brady [28] with the name SUSAN, acronym for Smallest Univalue Segment Assimilating Nucleus. It was later presented by Tomasi and Manduchi [29], renamed as the bilateral filter, which is now the most common name. Bilateral, meaning two sided, comes from the fact that both distance and pixel values are utilized. By this definition, the Yaroslavsky filter can also be said to be a bilateral filter. The penalty function, or the weights, are calculated using Gaussian distribution, instead of a step function as in Yaroslavsky filtering. The bilateral filter is in a way similar to Gaussian blur, the same way Yaroslavsky filtering is similar to average filtering. The weight of the spatial distance, W_s, is given in Equation 2.9. The weight of the radiometric distance (intensity difference) is given in Equation 2.11.

$$W_s = e^{-\frac{d^2}{2\sigma_s^2}} \tag{2.9}$$

Where d is the distance from the middle pixel in the mask (0,0).

$$d = \sqrt{x^2 + y^2} \tag{2.10}$$

$$W_r = e^{-\frac{(I(0,0)-I(x,y))^2}{2\sigma_r^2}} \tag{2.11}$$

σ_s and σ_r control the decay of the spatial and radiometric weights accordingly, see Chapter 2.3. The final weight is then found by multiplying the two weights together. The final weight must be normalized, so all the weights in the mask add up to 1.

$$W = W_s * W_r = e^{-(\frac{d^2}{2\sigma_s^2}+\frac{(I(0,0)-I(x,y))^2}{2\sigma_r^2})} \tag{2.12}$$

A bilateral filter where the middle pixel is excluded from the averaging process has also been proposed, [28]. If the middle pixel is far from the other pixels in intensity (an impulse), the denominator in the normalization process will then be zero. When this happens, median filter is used, and the impulse is removed. [30]. The bilateral filter have become popular and received a lot of researchers attention. This have led to new variations and speed optimizations.

2.6.1 Separable bilateral filter

The separable bilateral filtering method presented in [31] can be executed in a fraction of the time of the traditional bilateral filter. The weights are determined by Gaussian functions. The Gaussian filter can be separated, i.e., first filtered in the x-direction, then in the y-direction. Two simple 1D filters are faster than one advanced 2D filter. This will not produce the exact same result, but it will be almost as good as the full 2D kernel implementation. The result has been reported to be satisfying on edges and uniform areas, but can introduce streak artifacts on textured regions. [32]

2.6.2 Local histograms

Another bilateral algorithm is based on the neighborhood histogram. Each bin in the histogram says how many pixels that have a given intensity value, and the result can be calculated from this information. The method is fast, because two close neighborhoods share almost the same histogram. As the kernel moves across the image, some pixel values are removed from the histogram, and others are added, so the histogram is updated for the next neighborhood. The result can introduce artifacts on sharp edges, but they can be removed by iterating the filter three times. The final result will not be exactly the same as the full kernel implementation of the bilateral filter, but it will not contain any artifacts. [33]

2.6.3 Piecewise linear / Bilateral grid

In the piecewise linear method, a 2D gray scale image is represented in a 3D way by something named a Bilateral grid. In this domain, bilateral filtering is obtained by Gaussian blur in a simple linear convolution. Because an $O(n^2)$ convolution like Gaussian blur in the pixel domain becomes an $O(n)$ multiplication in the frequency domain, and the transform and inverse transform have the complexity $O(n \log n)$ and order of one in complexity can be gained. Because the bilateral filtering is not a simple convolution, the set of possible intensity values is made into segments, and a linear filter is computed for each segment. Because the grid is down sampled, the algorithm deals with less data, and is therefore fast. This algorithm has also proven to be effective on GPUs because of the 3D grid. For color images however, a 5D grid is required. This method was also used in a single input multiple data (SIMD) implementation on a digital signal processor (DSP) for color video. [34–37]

2.7 Non-local means

The NL-means algorithm tries to take advantage of the high redundancy in natural images, i.e., every small window has similar windows in the same image. Similar windows are often next to each other, but can also occur far from each other. NL-means searches for similar small windows inside a larger search window, which can be as large as the whole image. In an implementation of the NL-means algorithm in [38], a search window of 21x21 pixels is used, and a work window of 7x7 pixels. All the pixels with a similar 7x7 neighborhood inside the 21x21 neighborhood are then used for a weighted average. The neighborhood similarity is computed using the Euclidean distance of the intensities between the work windows. The final pixel value is a weighted average of the pixels with the most similar neighborhoods, where the weights are calculated using Gaussian distribution with the Euclidean distance as variable, Equation 2.13.

$$W_{nl} = e^{-\frac{[(I(p_1)-I(q_1))+\ldots+(I(p_n)-I(q_n))]^2}{2\sigma_r^2}} \tag{2.13}$$

Where p_n and q_n are pixels in different image patches in the search window. The result needs to be normalized.

2.8 Algorithm complexity

An important aspect of the algorithms is their complexity. Several speed optimizations have been proposed to the bilateral filter. A recent paper even proposed methods on how to make the complexity in constant time O(1) [39]. A summary of the complexities is given in Table 2.1. The table shows how the computation grows as the kernel radius, r, increases. However, only small radii is needed in denoising. Larger radii can be used to create other effects like blurring. More important issues are latency and FPGA resource requirement.

2.9 Temporal extension

To remove noise more aggressively, the mask defining spatially close pixels can be extended in size to include more samples, e.g., use a 5x5 instead of a 3x3 mask. This will probably include more of both equal and different pixels. If the mask is instead extended in the temporal direction, so that the mask is transformed from a 2D to a 3D mask, there is a larger probability that more of the equal pixels will be added if the motive is the same in both frames (no motion). Therefore

Table 2.1: *Algorithm complexity*

	Complexity
Brute force bilateral/Yaroslavsky filter	$O(Sr^2)$
Separable bilateral filter	$O(Sr)$
Local histogram bilateral filter	$O(S\log r)$
Piecewise linear bilateral filter	$O(S+\frac{256S}{r^2\sigma_r})$
Non-local means	$O(Sr^2r_w^2)$

a motion detector could decide to turn off temporal filtering in case of motion, and on if there has not been any motion. This would however require additional logic. The denoising algorithms that also checks radiometric distance, i.e., the Yaroslavsky, Bilateral and NL-means, do not need an extra motion detector. If there have been motion, these algorithms will not use pixels from previous frames anyway, as the radiometric distance will be too large. These algorithms can therefore be said to have a built in motion detector. An important question is how far the mask should be extended in the temporal domain. Equal pixels is most likely to be found in the frames closest to the current frame. It is therefore not adequate to extend it several frames back in time. In addition, each additional frame requires extra memory and more pixels to be computed. The mask can theoretically also be extended ahead in time, as equal pixels are also likely to be found there, but this is not recommended in a real time system, as it increases the latency dramatically. [34]

2.10 Color handling

The radiometric distance makes most sense in gray scale data, where it can be measured as the difference of two pixel values. In color data, a pixel does not have just one value, it has three. There are different ways to represent color, as explained in Chapter 1. One way of measuring the radiometric distance on RGB-data is on the three color planes individually. A pixel will then get three distance values, and the noise filtering is applied on the three planes individually. Only pixels with the same color value are then averaged together, in correlation to the radiometric distance of the specific color. Another way to handle color data, is to measure the radiometric distance by calculating the Euclidean distance. All three RGB values will then contribute to a mutual radiometric distance. This will avoid color values from the same pixel ending up in different similarity groups. An experiment in [32] compares a bilateral filter implementation on individual RGB-planes, and on the RGB-planes jointly. The results are given in Figures 2.1, 2.2, and 2.3, and shows that processing the planes jointly gives the best result, but processing the planes individually is faster.

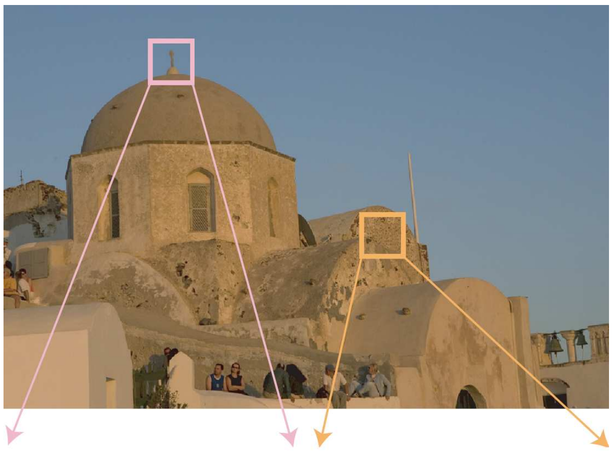


Figure 2.1: *The original image*

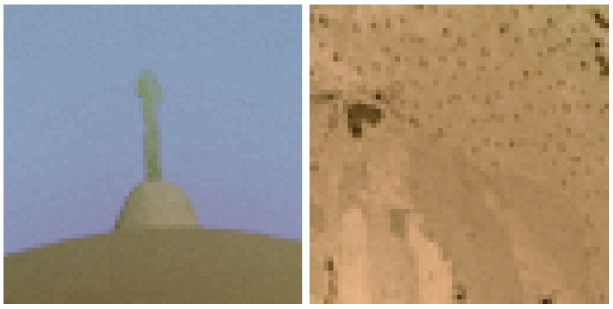


Figure 2.2: *This image is bilateral filtered on individual RGB-channels. The cross is more faded than in Figure 2.3, where the bilateral filtering is done on the color planes jointly. This type of fading is called color bleeding as the cross can be said to bleed color.*

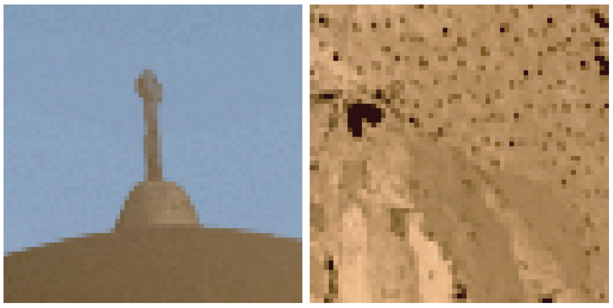


Figure 2.3: *This image is bilateral filtered on the RGB-channels jointly. The result is better than filtering on the RGB-channels individually, Figure 2.2, but it is also more computationally expensive.*

Another way of handling color is using a different color space than RGB. There are a numerous of different color spaces that can be used to represent color images. A color space where one of the channel represents a gray scale image, and the two other channels represents color information can be useful, as the noise filtering can be successfully applied on just the first channel. Noise filtering typically comes early in an imaging pipeline, and using another color space than RGB would mean doing a conversion in color space, which can be a computational heavy task. Other color spaces are therefore not further explored.

3 Project work algorithm

The project work led to a new algorithm inspired by a number of complex algorithms [40–53]. These algorithms often consisted of several filter options, and some logic choosing the best option. The project work algorithm was chosen so that the simplest and best methods in the reviewed algorithms could be united in an as simple as possible algorithm, capable of handling most scenarios well. The project work algorithm therefore consisted of several simple components, which together made an advanced algorithm. Its mode of operation is given in Chapter 3.1. The full project report is available in [54].

3.1 Mode of operation

The algorithm consists of three different detectors; a motion detector, an impulse detector, and an edge detector. These three detectors will choose between five different filtering techniques for the pixel being processed. The five filtering techniques are median filtering, average filtering, temporal filtering, average temporal filtering, and no filtering. The decision diagram is shown in figure 3.1.

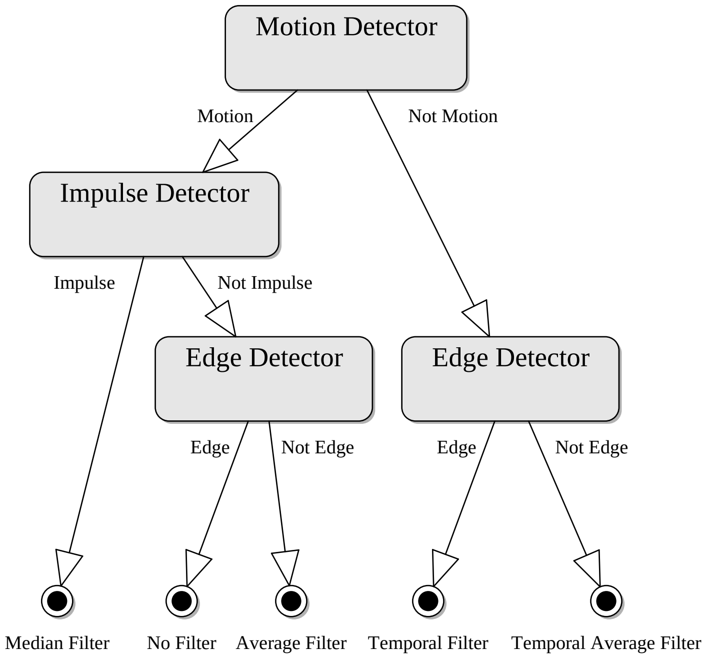


Figure 3.1: *The algorithm decision diagram*

3.1.1 Motion Detection

The motion detector is used to detect motion and avoid averaging in the temporal domain in areas with motion. Averaging two or more frames where an object has moved, causes the object to leave a trail in the filtered result, which is why temporal averaging must be avoided in such areas. The motion detector used, is loosely based on the motion detector in [40]. One of the differences is using the current and one previous frame, instead of two previous frames. This makes the detector less complex, and saves a lot of memory. Another difference is using a value calculated from neighbor pixels using Matrix 3.1, instead of a single pixel.

$$\frac{1}{36}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 24 & 2 \\ 1 & 2 & 1 \end{bmatrix} \tag{3.1}$$

If the absolute difference between two corresponding pixels in the previous filtered image and in the current weighted average pixel is greater than a fixed threshold, the pixel is marked as a moving pixel in a binary mask. The middle weighted average mask refuses heavy noise in a neighbor pixel to affect the pixel in process too much, but still eliminates some of the false motion positives caused by noise. If an equal weighted average mask would have been used instead, noise in one of the corners would have the weight 1/9 instead of 1/36, and could potentially cause all of its neighbor pixels incorrectly to be classified as motion. The same matrix is also used for average filtering (so the result does not need to be recomputed in this case), where the matrix moderates the effect of blurring. Anyhow, noise still causes a lot false positives, so the next step is to remove them from the binary mask. This is done with morphological opening.

3.1.2 Impulse Detection

A weakness with the motion detector, is actually one of its key features. Because all single motion pixels are removed, a sudden case of salt or pepper noise ends up being classified as non-motion, causing the pixel to be temporal filtered. This will tone down the effect of a sharp impulse, but will not remove it. Actually, if the impulse had been detected as motion, the result would have been even worse. The impulse detector therefore detects single pixels differing extremely from the previous frame, so they can be properly dealt with.

The impulse detector checks if the absolute difference between the previous filtered frame, and the current unfiltered frame is greater than a fixed threshold. This threshold is a lot higher than the threshold used in the motion detector. A binary mask of the result is created. Then if there are more than two 1's in a 3x3 window in the binary mask, it is not impulse noise, and the 1's are removed from the binary mask.

Note that this method also detects two neighbor pixels, a couplet, impaired by impulse noise. Couplets are difficult to remove, and a key driver to high volume image sensor yield [2]. It also removes dead pixels that have the same value in every frame, and not only a sudden impulse. The dead pixel will be replaced in the filtering process. The impulse detector will then in the next frame see a transition from a filtered pixel to a dead pixel, and filter the dead pixel again, and so on.

3.1.3 Edge Detection

The edge detector is used to detect edges, or more accurately, variability and details in the image. One of the problems is to distinguish between details and noise, so the noise can be removed from the edges. Another problem is to remove the noise without making the edges smooth and indistinct. In [41] the variance, σ^2, and a threshold is used to extract edges. In [42] the median absolute deviation and a threshold is used to extract edges. The median absolute deviation is a robust variance estimator, and works for highly corrupted images. The method in the presented algorithm uses standard deviation, σ, and a threshold to extract edges and make a binary mask of edge pixels. A better solution for an FPGA implementation in future work might be to use the method in [43], or variance instead of standard deviation, which is basically the same thing. For the Matlab simulation anyhow, local standard deviation is much easier to implement and produces good results.

3.1.4 Median filtering

All impulses detected by the impulse detector are median filtered. The 3x3 median filter replaces the pixel being processed with the median of the pixel and its eight neighbors. This will effectively remove impulses like salt and pepper noise. An advantage of filtering impulses with median filtering, especially compared to spatial or temporal average filtering, is that the impulse will be replaced by an actual pixel in the 3x3 working window, and not by a made up value where the impulse might have a huge influence. The median filter is therefore known to be edge preserving.

3.1.5 Average filtering

The average filter replaces the pixel with a weighted average generated by the Matrix (3.1). The matrix gives pixels closer to the middle pixel higher weight, while the middle pixel has a weight equivalent to 6/9 of the total weight. The four closest pixels have a weight equivalent to 2/9, and the four corner pixels have a weight equivalent to 1/9. The middle pixel has the highest weight to keep the blurring of details at a minimum, while still doing noise reduction. If more aggressive noise reduction is wanted, an average filter with equal weights can be used. Keep in mind that real world data will not be severely degraded by noise.

3.1.6 Temporal filtering

Temporal filtering is used when edges are detected, and motion is not detected. The temporal filter takes 1/2 of the current pixel, and 1/2 of the previous filtered pixel. This method is presented in [40], and is stated mathematically in Equation (3.2). It effectively removes noise without blurring edges.

$$f_{\text{filtered}}(x, y, t) = \frac{1}{2}[f_{\text{current}}(x, y, t) + f_{\text{filtered}}(x, y, t-1)] \tag{3.2}$$

3.1.7 Temporal average filtering

This filter is almost the same as the temporal filter in 3.2, but it has an important difference. It takes 1/2 of the output from the weighted average matrix (Matrix 3.1) and 1/2 of the previous filtered image. This way it removes more noise than the regular temporal filter, but will still keep some details. It is mathematically stated in Equation 3.3.

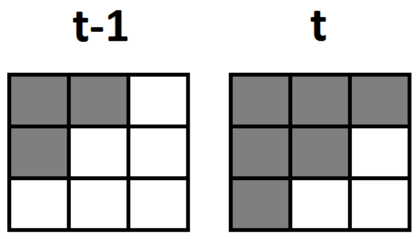
$$f_{\text{filtered}}(x, y, t) = \frac{1}{2}[f_{\text{averaged}}(x, y, t) + f_{\text{filtered}}(x, y, t-1)] \tag{3.3}$$

3.2 Disadvantages and improvements

3.2.1 Motion detector robustness

The static thresholds assumed a certain noise level, and the results were good for videos with this certain quality. The problem arise when the video quality varies. There can be both dark and bright areas in a video, with accordingly more noise in the dark areas. If more powerful noise occurred than what the motion detector was set for, noise would be detected as motion, and temporal filtering would be turned off. The power of temporal filtering would then not be utilized when it actually was the most needed. If the motion threshold was widened, so that powerful noise would not be detected as motion, the result would be worsened in video with little noise. This would happen because actual motion would not be detected. In other words, the motion detector was not robust enough.

In an attempt to fix this problem, fuzzy thresholds were introduced. The weight of the previous frame is then higher when the differences of the pixels are small (high probability of static pixels), and vice versa. The peak performance decreased in some situations, since fuzzy thresholds sometimes includes pixels that should have been cut, but with a low weight. A small improvement can also



(a) Previous frame (b) Current frame

Figure 3.2: *An edge in a 3x3 mask. The project algorithm would ignore similar neighbor pixels, averaging only the center pixel in time (t) and (t-1), or in this example use no averaging at all because of the motion.*

be made to Matrix 3.1, as dividing by 32 is more hardware friendly than dividing by 36. This can easily be achieved by lowering the weight of the middle pixel to 20 from 24.

3.2.2 Edge handling

Another problem is how the algorithm handled edges. If an edge was detected, the algorithm would either (temporal) filter the middle pixel with the time delayed version of the same pixel, or use no filtering at all in case of motion. This method worked well for avoiding blurring and ghosting, but it unfortunately also avoided efficiently noise removal in such cases. Consider the edge in Figure 3.2. The algorithm would not do any filtering at all because of the motion, when there are 9 similar pixels in the current and previous frame that could have been used.

4 Modified Yaroslavsky filtering

The original Yaroslavsky noise filter as described by Leonid P. Yaroslavsky, looks for similar pixels inside a given spatial neighborhood. The manner of operation is roughly the same as the bilateral filter, but it is also simpler since pixels can only be assigned the weight 0 or 1. It can be implemented on an FPGA in a straight forward and simple way, and have high potential of achieving both low latency and resource usage, which is the reason it is preferred over the bilateral filter. The optimization techniques available for the bilateral filter could be interesting, even though some of them was reported to introduce small artifacts. In addition, it was also not always clear how the optimizations can be applied to (color) video. The best place to start is therefore the easily understandable Yaroslavsky filter.

Three simple modifications to the Yaroslavsky filter is also proposed. First, it should be extended to the temporal domain by also checking for similar pixels in the previous frame. More frames could also be added, but there is a substantial memory overhead for each one. Second, instead of comparing with just one threshold, several thresholds should be used. Neighbor pixels are then categorized in similarity groups based on how equal the intensities are. The neighbor pixels are then not just averaged altogether as in the original Yaroslavsky filter. Weights are given to the pixels based on the similarity group they are in. A higher weight is assigned to pixels in the group defined by the lowest threshold. A weighted average is then calculated. Assigning different weights to pixels based on radiometric distance is also used in bilateral filtering. The weights in bilateral filtering is calculated for each pixel using Gaussian distribution. The modified Yaroslavsky filter uses predefined weights for each similarity group, and therefore avoids a calculation of the weight as in bilateral filtering, but still imitates the weighted average function. The third and last modification is to let the median filtering overrule the other filtering when most of the neighbor pixels exceed the largest threshold.

The proposed modifications to the Yaroslavsky algorithm are a result of the project work, and further reading and development. Comparing the modified Yaroslavsky to the project work algorithm shows that they actually have the same functions. Where the project algorithm had a separate motion detector, edge detector, and an impulse detector, the modified Yaroslavsky algorithm have these three modules incorporated in just one.

4.1 Neighborhood size

The original and the modified Yaroslavsky algorithm checks a number of pixels for similarity. This number is bounded by the neighborhood size. The most similar pixels is likely to be found closest to the middle pixel. At the same time, a larger neighborhood size implies more samples which could lead to a better average.

A larger neighborhood also implies more computation. As an example, consider Figure 4.2. The project work algorithm would detect both motion and an edge, and therefore do no filtering at all. The modified Yaroslavsky takes the similar pixels from both the current and the previous image, and ignores the different ones (from the other side of the edge). A full 3x3 window (N_8 neighborhood, [5]), and a diamond shape (N_4 neighborhood, [5]) is simulated in Chapter 5. The simulations show that the performance gain of the largest neighborhood is not overwhelming, and actually have worse results on the synthetic video in some cases, because of the test videos low resolution and fine detailed grid. The extra computation of a larger neighborhood is therefore not defended, and the diamond shaped N_4 neighborhood is proposed as shown in Figure 4.1 as it provides good results with minimal computation. Note that the N_4 neighborhood in the previous frame is actually of five pixels because of the previous value of middle pixel itself. The notation used for the whole neighborhood is therefore $N_{4,5}$. Thus, the highest number of pixels that can be averaged is ten. The lowest number of pixels that will get averaged is three, i.e., if more than 7 of the neighbors are not similar at all, the pixel is an impulse and is median filtered. However, if the trend of higher resolution continues, a larger neighborhood is probably needed.

4.2 Thresholds and weights

The threshold and weight values are (compared to the neighborhood size and the actual number of thresholds and weights) variables that can easily be changed on an FPGA, as long as the values are within certain limits. As shown in Chapter 6, a practical limit for the weights is 4 bit, or between 0 and 15. The weights and thresholds can be set to mimic the bilateral filter, with a (more) discrete Gaussian distribution. The optimal thresholds and weights depend on several variables, e.g., level of details and noise in the video. It should be possible to find good thresholds and weights for a specific camera if its noise model is known, but it might require some trial and error. Universal thresholds and weights, optimal for all scenarios do not exist.

4.3 Matlab implementation

The first operation in the Matlab implementation of the modified Yaroslavsky algorithm is to check the input for the resolution (rows and columns), number of planes (color or gray scale video), and number of frames. All the frames of a low resolution video can be handled at the same time. To keep the memory usage low, only one frame should be sent to the algorithm when the video is of high resolution. A small script reads one frame at a time, and writes the result to an output file, in this case.

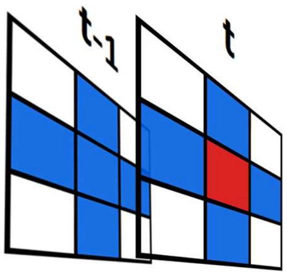


Figure 4.1: *The neighborhood size chosen. It is the four closest pixels in the current frame (N_4), and the five closest in the previous frame. The highest number of pixels possible in the average process is therefore ten.*

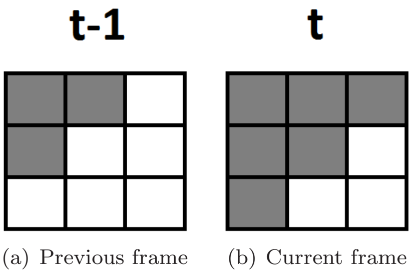


Figure 4.2: *An edge in a 3x3 mask.*

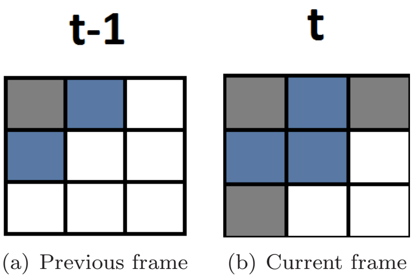


Figure 4.3: *The pixels used for averaging when the neighborhood size is $N_{4,5}$ as shown in Figure 4.1 and the image is Figure 3.2, are marked with blue. The algorithm will average the two blue pixels in the previous frame with the three blue pixels in the current frame. The weights will be equal because the neighbor pixels are exactly the same as the middle pixel. If a pixel was similar, but not equal, it would have gotten a lower weight. If it had been the full $N_{8,9}$ neighborhood, all 9 gray pixels in Figure 4.2 would have been averaged.*

The algorithm can now start processing the pixels. The intuitive way of doing this, is by making two nested for loops that process the pixels one by one. Because Matlab prefers working with matrices, and because a frame is nothing else than a large matrix, there is a faster way of processing the pixels. The first operation in the modified Yaroslavsky algorithm is to find the difference between a pixel and its neighbors. This operation is achieved by first padding the input symmetrically using the function *padarray* from the Matlab Image Processing Toolbox. To find the difference between the pixel and its (e.g.) left neighbor, a sub image of the padded image is then extracted. This sub image is shifted one position (to the left) and is as large as the original image. A matrix containing the difference between all the pixels in the original image and their left neighbors is then extracted by subtracting the sub image from the original image, and taking the absolute value of this result. This operation is done for every neighbor, so 4 matrices are obtained from the current frame, and 5 from the previews frame, if the neighborhood size is as proposed in Chapter 4.1. Matlab code for this operation is given in Listing 1.

Listing 1: *Matlab code for extracting a matrix containing the difference values between a pixel and its left neighbor.*

```
padded_image=padarray(image(:,:), [1,1], 'symmetric');
image_left_shifted=padded_image(2:rows+1, 1:columns);
difference_left=abs(image(:,:) - image_left_shifted(:,:));
```

All pixels in the difference image are then compared with four thresholds, and four binary masks for each neighbor are made. These binary masks determine the similarity group for the given neighbor pixel. They are mutual exclusive and collectively exhaustive, i.e., for a given pixel, only one of the four binary masks will have a 1, the others will have a 0, as a pixel must be in one similarity group. There is also a similarity group for the (dissimilar) pixels who exceed the highest threshold. Matlab code for this operation is given in Listing 2.

Listing 2: *Matlab code for finding the similarity group of the left neighbors. The lowest threshold is T1, and the highest is T4*

```
similarity_group1_left=(difference_left<=T1);
similarity_group2_left=(difference_left>T1)&(difference_left<=T2);
similarity_group3_left=(difference_left>T2)&(difference_left<=T3);
similarity_group4_left=(difference_left>T3);
```

When all the similarity groups for all the neighbor pixels have been found, a matrix containing the normalize constants can be made. The total number of neighbor pixels in the different similarity groups must then first be added, resulting in four new matrices, that added together will be the total number of the neighbors (9). Each of these four matrices now contains the number of pixels in each similarity group. The Matlab code for this operation is shown for similarity group 1 in Listing 3.

Listing 3: *Matlab code for adding together the total number of pixels in similarity group 1 for all neighbors.*

```
similarity_group1_total = (similarity_group1_left + ...
 + similarity_group1_right + similarity_group1_top + ...
 + similarity_group1_bottom + similarity_group1_(t-1)left + ...
 + similarity_group1_(t-1)right + similarity_group1_(t-1)top + ...
 + similarity_group1_(t-1)bottom + similarity_group1_middle(t-1));
```

The values in these four matrices are then multiplied with their respective weights and added together to a single matrix, containing the normalizing constant for each pixel. This operation is shown in Listing 4.

Listing 4: *Matlab code for making a matrix with each pixels normalize constant. The weight for similarity group 1 is W1.*

```
normalizing_constant = ones([rows colums]);
normalizing_constant = (normalizing_constant.*W_center + ...
+ W1.*similarity_group1_total + W2.*similarity_group2_total + ...
+ W3.*similarity_group3_total);
```

The weighted average can now be calculated and normalized by dividing each value with the corresponding value in the normalizing constant matrix. This is shown in Listing 5.

Listing 5: *Matlab code for computing the weighted average.*

```
result = (W1.*similarity_group1_left.*image_left_shifted + ...
+ W2.*similarity_group2_left.*image_left_shifted + ...
+ W3.*similarity_group3_left.*image_left_shifted + ...
+ W1.*similarity_group1_right.*image_right_shifted + ...
+ ... )./normalizing_constant;
```

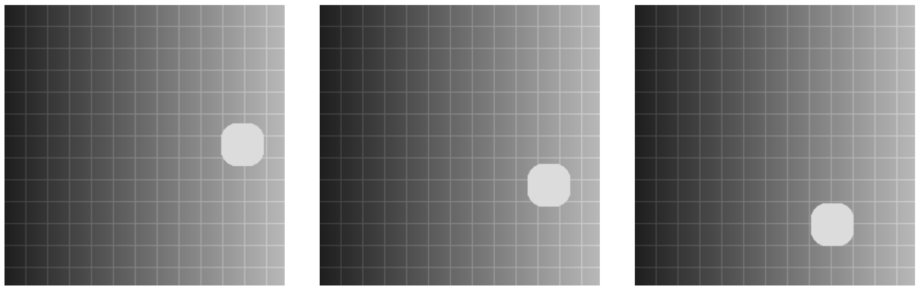
The final step is to overwrite the invalid values, where most of the neighbor pixels exceeded the highest threshold. This is done by first making a binary mask of these invalid pixels, and then performing median filtering on the corresponding pixels, overwrite the weighted average. The median filter is implemented using the *medfilt2*, which is another function in the Image Processing Toolbox. It takes the median of the N_8 neighborhood, and does not consider the pixels in the previous frame. The Matlab code for this operation is given in Listing 6.

Listing 6: *Matlab code for overwriting invalid pixels*

```
median_filter_mask = similarity_group4_total > 7;
```

```
result = result.~*median_filter_mask + ...
+ medfilt2(image(:,:),'symmetric').*median_filter_mask;
```

This final result is now saved, as the a previous image, so it is available for the next input image. Only two loops are used in the algorithm; the first loops through the number of planes, and the second loops through the number of frames. All the pixels are processed at the same time with matrix operations, which is preferred by Matlab.



(a) Frame 1 (b) Frame 5 (c) Frame 10

Figure 5.1: *The synthetic video used. The white disc moves around in circles over the grid*

5 Matlab simulations

The modified Yaroslavsky algorithm has been implemented in Matlab for simulation, as shown in Chapter 4.3. It can take a long time optimizing different algorithms for each test data. The algorithm variables are therefore not changed for the different test data and the robustness is therefore also tested. The emphasis in the presented simulation results, is on the synthetic data, as comparable numbers can easily be generated, since there exists a noise free full reference copy (Chapter 1.4. This might be a drawback, as the algorithms are ultimately to be used on real world data. The results from the synthetic data should however correlate with the results from real world test data to some extent. The algorithms have also been tested on real world data, but the results can only be compared subjectively. Also note that the figures have to be large in order to be compared, as the differences is on a pixel level.

5.1 Synthetic data (full reference)

The synthetic test video shows a white disc moving in circles over a background of variable brightness and a grid, Figure 5.1. The video can force unveiling of unwanted effects like blurring of the grid, and the white disc leaving traces. It has been tested with various amounts of motion, and various amounts of noise. The Matlab simulation time has also been noted. Figure 5.2 shows a still picture of the video with added noise. Salt and pepper noise degrades 1 out of 1000 pixels, and Gaussian white noise with mean 0, and standard deviation 6 (and a pixel range [0 255]) degrades all pixels. Since the noise was added synthetically, a full reference video exists, and the PSNR (Chapter 1.4.3) of the noisy video can be calculated. The added noise resulted in a PSNR of 32 dB. The gain can later be

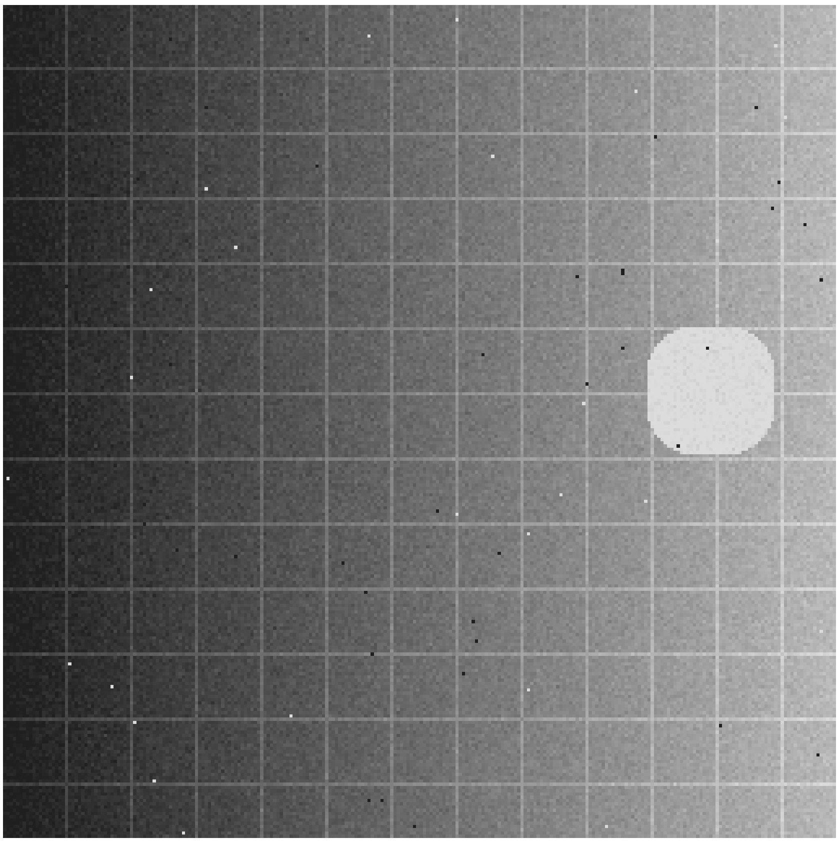


Figure 5.2: *Gaussian and impulse noise corrupted video with PSNR = 32dB. This is the same as Figure 5.3 (b), but in larger scale for comparison purposes.*

calculated for different algorithms.

The result from the original project work algorithm was very good on parts of this synthetic test. The PSNR gain on 7.2 dB is second best compared with the other algorithms. The problem is that it is not robust enough, which is shown in Table 5.1. The performance drops drastically in case of heavy noise. The original idea was that the thresholds should be modified for different scenarios with different noise types. Cameras do have different and known noise models which supported this idea. Still, various light levels means various noise level, so that several thresholds would be needed for the same camera, which is the reason why a version of the project algorithm with fuzzy thresholds was developed. The greatest problem with using only one (binary) threshold in the motion detector,

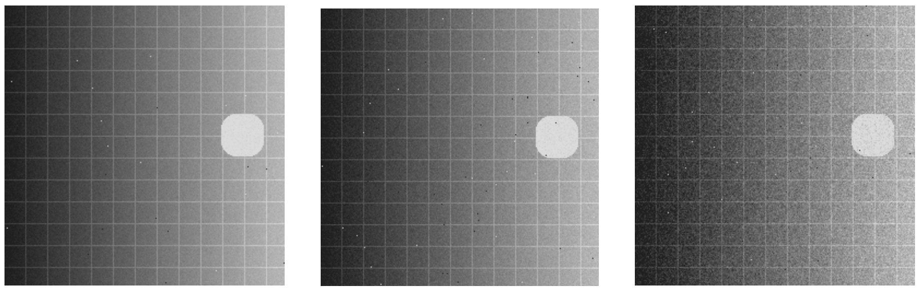

(a) 35 dB (b) 32 dB (c) 25 dB

Figure 5.3: *The synthetic videos used. Different noise intensities are added, giving a lower PSNR where most noise is added.*

is that temporal filtering would be completely turned off, when it in some cases was most needed. This would happen if heavy noise fooled the motion detector to believe there was motion, and then turn temporal filtering off.

When the project work algorithm was fitted with fuzzy thresholds in the motion detector, the results on the synthetic video are always under the performance of the original project algorithm. This is somewhat surprising, as the modified (with fuzzy thresholds) Yaroslavsky does better than the original Yaroslavksy. This can however in some sense be explained for the synthetic videos with the highest PSNR. The original binary thresholds are near to optimal in detecting motion for this data and will cut out most of the noise, while the fuzzy thresholds will blend in a bit of noise in the result. Perhaps should the fuzzy thresholded project algorithm have been more adjusted. Also note that the project algorithm both with and without the fuzzy thresholds leaves the black impulse in the white disc, Figures 5.4 and 5.5.

The original Yaroslavsky does a good performance especially on the high noise data. On the 25dB data, this algorithm achieves the best PSNR gain of all. This is because all pixels have equal weight, which is good when there is a lot of noise. The result on the test data with little noise is however the least good, as the equal weight causes a lot of unwanted blurring. The results of the original Yaroslavsky algorithm are actually computed using the modified Yaroslavsky algorithm, by setting the weights equal. This version of the original Yaroslavsky is therefore also a bit modified, as it is extended to the temporal domain, and has a median filtering options, but not the fuzzy thresholds. This means that the proposed modified Yaroslavsky algorithm is capable of the same PSNR gain as the original Yaroslavsky.

The proposed algorithm ($N_{4,5}$) achieves the best performance on the two

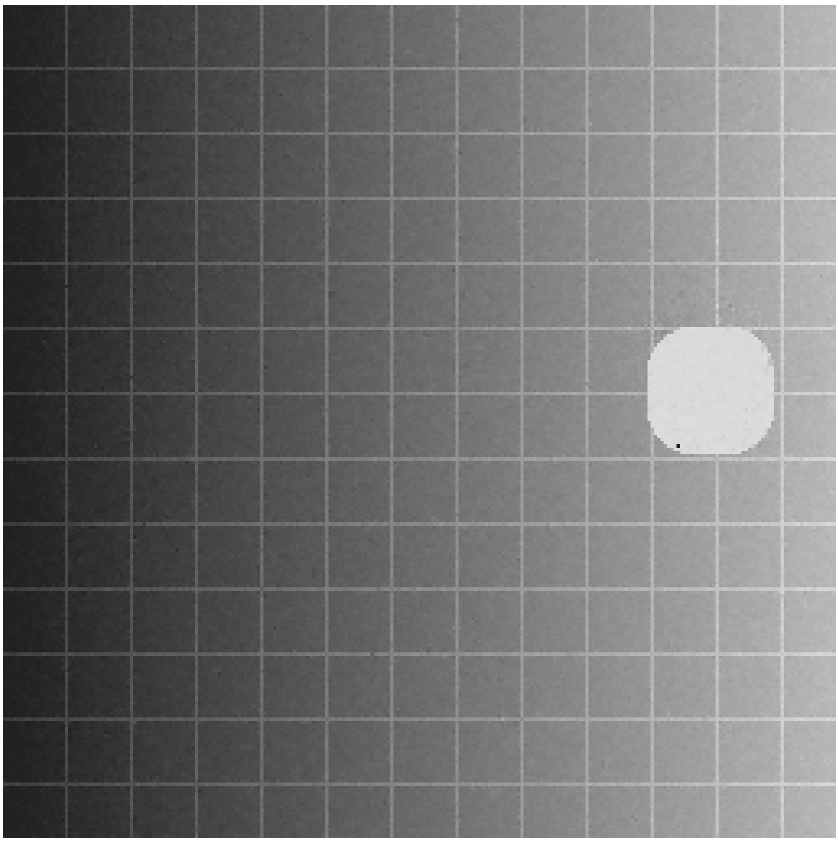


Figure 5.4: *The result after filtering 5.2 with the project work algorithm. The result is good, but it did not manage to remove the ipulse in the white disk. PSNR gain = 7.2dB*

videos with highest quality. The reason why it does better than the same algorithm, but with a larger neighborhood ($N_{8,9}$) is probably because of the grid in the video which is just one pixel thick. The diamond shaped neighborhood in the proposed algorithm matches this very well compared to the larger neighborhood. The algorithms should therefore have been tested on several different synthetic videos, as the $N_{8,9}$ version is likely to outperform the $N_{4,5}$ in other scenarios.

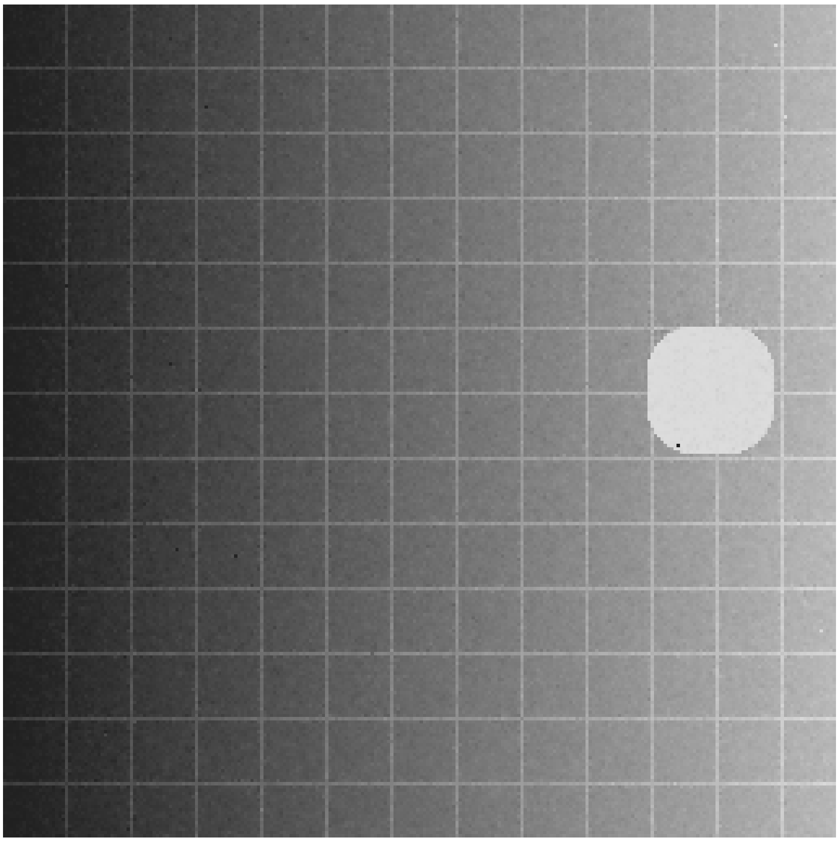


Figure 5.5: *The result after filtering 5.2 with the project work algorithm with fuzzy thresholds. The result is generally worse than the original project algorithm, but looks possibly smoother in motion areas. PSNR gain = 5.4dB*

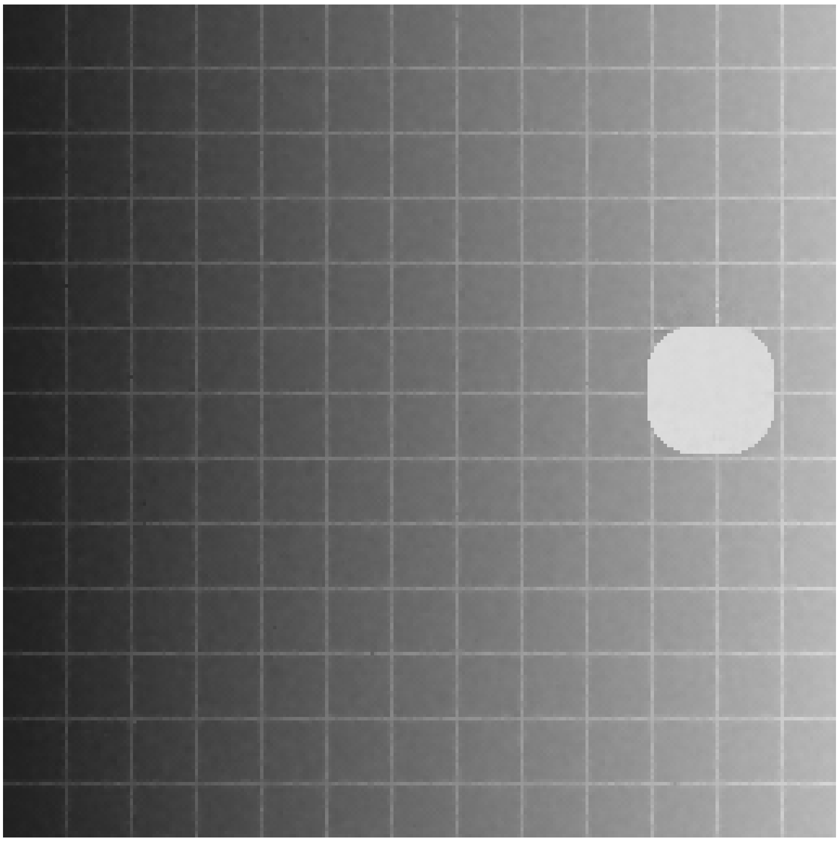


Figure 5.6: *The result after filtering 5.2 with the original Yaroslavsky $N_{4,5}$ filter. The grid has been heavily blurred, which affects the result. It have in contrast to the project algorithms removed all the impulses in this frame. PSNR gain = 6.1dB*

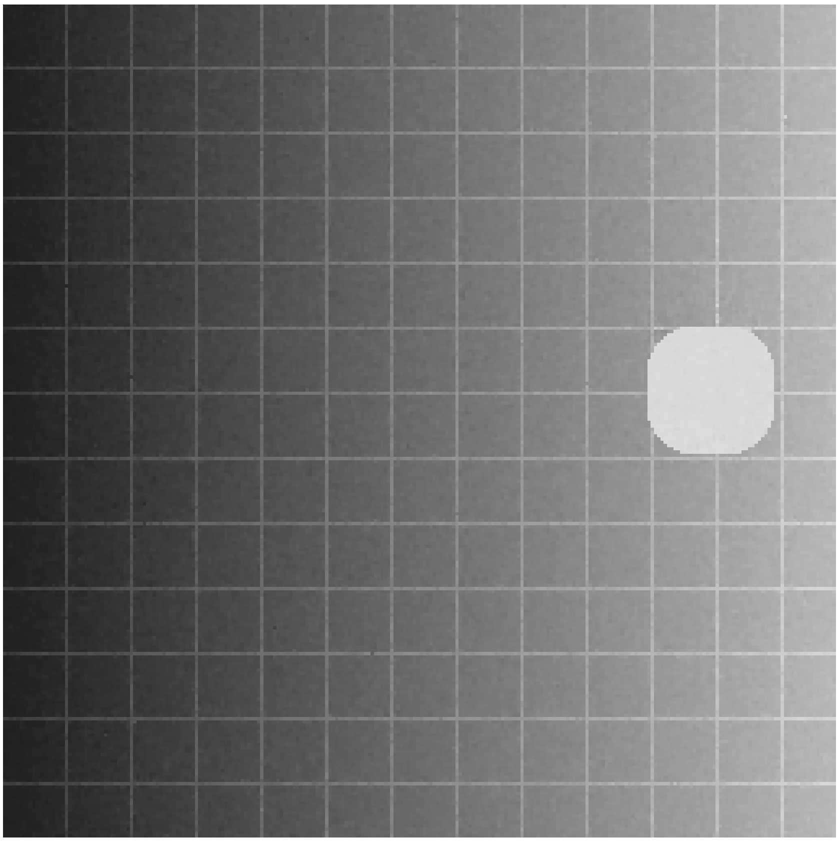


Figure 5.7: *The result after filtering 5.2 with the proposed algorithm, modified Yaroslavsky $N_{4,5}$. This is the best result achieved for this test video. All impulses are also removed in this frame. PSNR gain = 7.8dB*

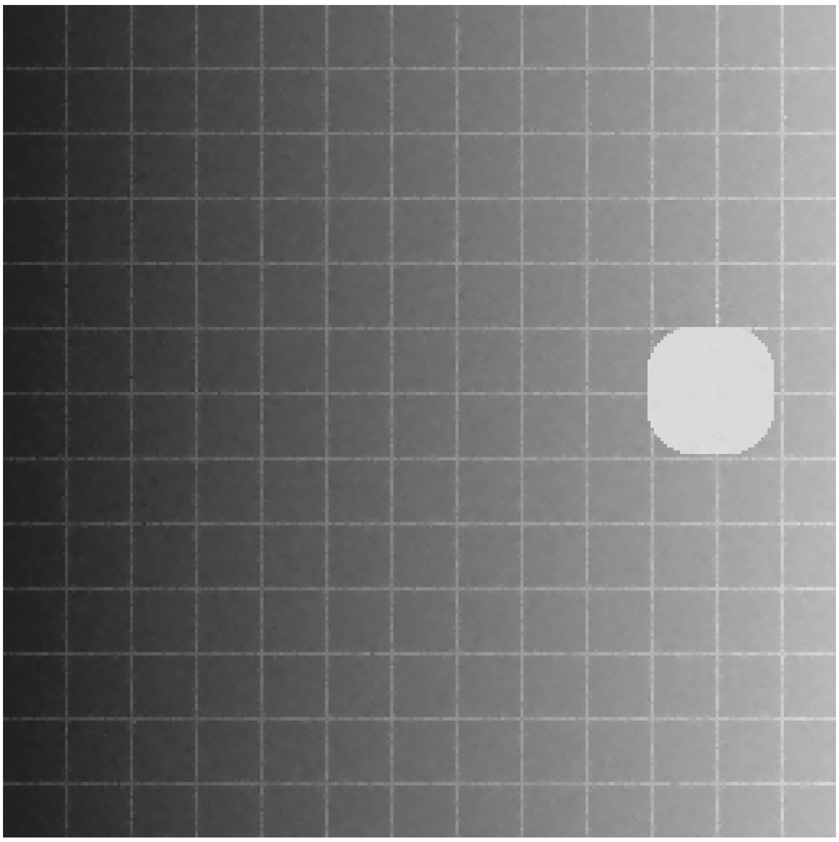


Figure 5.8: *The result after filtering Figure 5.2 with the proposed algorithm, modified Yaroslavsky with* $N_{8,9}$ *neighborhood. The grid is more blurred than with the smaller neighborhood. This might be improved by adjusting thresholds. All the impulses are also removed in this frame. PSNR gain = 6.1dB*

Table 5.1: *PSNR gain for videos with different noise levels (video quality). The algorithm thresholds are not changed. Keep in mind that the original Yaroslavsky algorithm is just the same as the modified Yaroslavsky, but with equal thresholds. The proposed algorithm is therefore able to achieve the best result on all the videos.*

Video quality	*25.4dB*	*31.5dB*	*34.5dB*
Project	2.6dB	7.2dB	7.5dB
Project fuzzy	2.0dB	5.4dB	6.0dB
Yaroslavsky $N_{4,5}$	**6.3dB**	6.1dB	4.5dB
Modified Yaroslavsky $N_{4,5}$	4.4dB	**7.8dB**	**8.2dB**
Modified Yaroslavsky $N_{8,9}$	4.5dB	6.1dB	5.5dB

Table 5.2: *PSNR gain for videos with different amounts of motion. The disk speed varies as number of frames to do a complete circle, i.e., lower numbers means more motion. Video quality is fixed to 31.5dB. All of the algorithms achieve better results when there is less motion, meaning that they all have a working motion detector. This result also correlates to Figure 1.7 which shows that the motion detector controls the filtering intensity. Also note that the proposed algorithm even achieves better results with high amounts of motion, than any of the other algorithms with low amounts of motion.*

Disk speed $[frames/360°]$	*30*	*60*	*120*
Project	7.0dB	7.2dB	7.4dB
Project fuzzy	5.2dB	5.4dB	5.5dB
Yaroslavsky $N_{4,5}$	6.0dB	6.1dB	6.2dB
Modified Yaroslavsky $N_{4,5}$	**7.6dB**	**7.8dB**	**7.9dB**
Modified Yaroslavsky $N_{8,9}$	5.9dB	6.1dB	6.2dB

Table 5.3: *Matlab simulation time for a 120 frames 256x256 gray scale video. Even though the simulation time on a Matlab implementation of an algorithm does not necessarily correlate well with the performance of an FPGA implementation, the results will give a hint. Because the original Yaroslavsky is simulated as the extended Yaroslavsky with equal thresholds, they have the same simulation time. In reality, the original Yaroslavsky would be faster. Also note that all the Yaroslavsky versions simulates faster than the project work algorithms.*

	Simulation time
Project	23.1s
Project fuzzy	32.2s
Yaroslavsky $N_{4,5}$	**10.3s**
Modified Yaroslavsky $N_{4,5}$	**10.3s**
Modified Yaroslavsky $N_{8,9}$	15.8s

5.2 Real world data (no reference)

The real world test data are 1920x1080 full HD video. Video quality cannot be fully determined by watching the video frame by frame, and the results are preferably to be viewed digitally. Because no noise free copy of the real world data exists (no reference), the PSNR gain cannot be calculated. The results must therefore be compared subjectively. Only a few examples of the test scenarios, and the results of the proposed algorithm is given as figures in this text. An example of the method noise is also given. Only a selection of the results are available digitally in the Appendix, as the amount of data is substantial. Just a second of an uncompressed video in 1920x1080p60 format takes 3*60*1920*1080 bytes, or 356MB. This gives a bitrate of 356MB/s or almost 3Gbit/s (the maximum uncoded transfer rate of the bus interface SATA revision 2.0 is 2.4 Gbit/s, taking 8b/10b encoding into account [55]), underlining the problem of real time denoising. Watching 1920x1080p60 videos without lag on a regular computer, might require the whole video to be pre-loaded in the computers memory (e.g., by using a RAM disk). It is also clear that testing different videos with different algorithms using different parameters will rapidly exceed the capacity of e.g., a 4.7GB DVD. More content could be fitted if the data was allowed to be compressed, but the compression would affect the result and make videos incomparable. An interesting benchmark would be to measure the bytes saved by compressing a noisy video and a denoised video, but this is beyond the scope of this master's thesis.

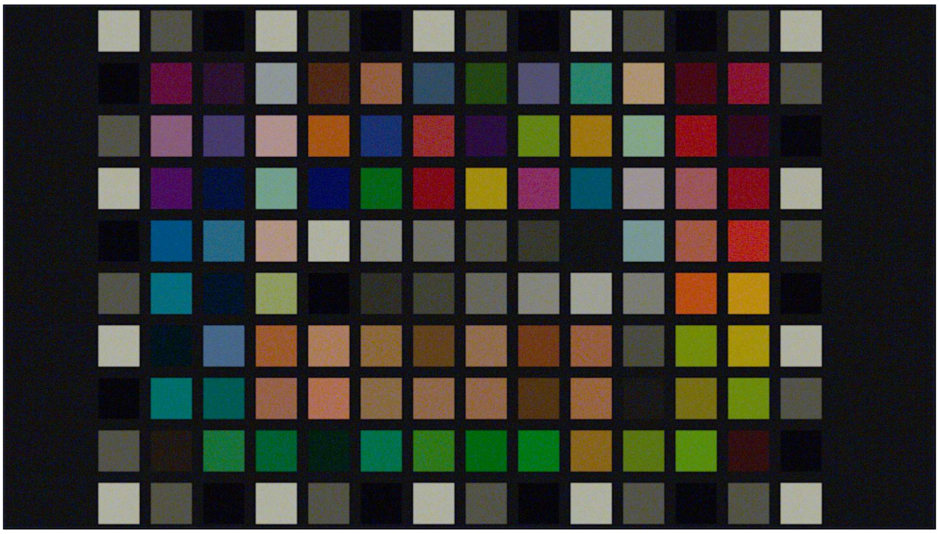


Figure 5.9: *Test scenario 1.*

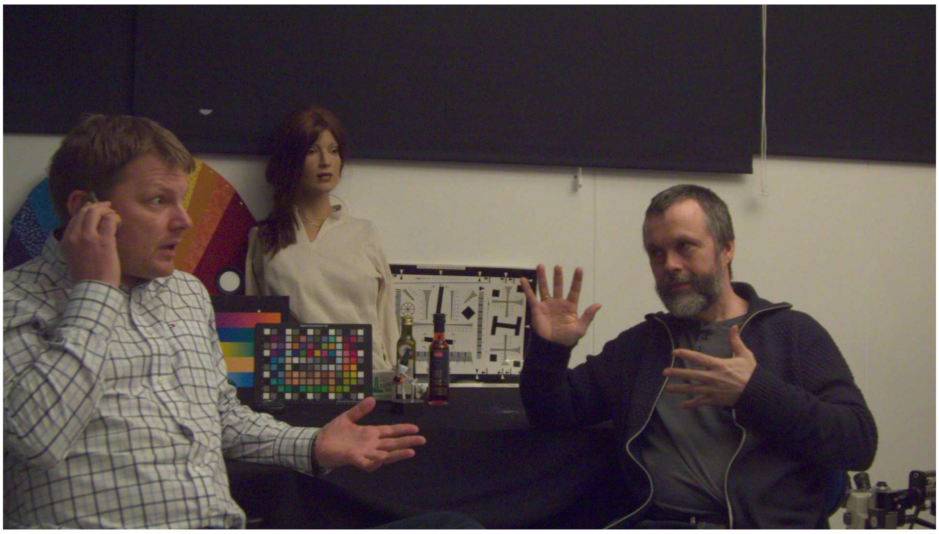


Figure 5.10: *Test scenario 2.*

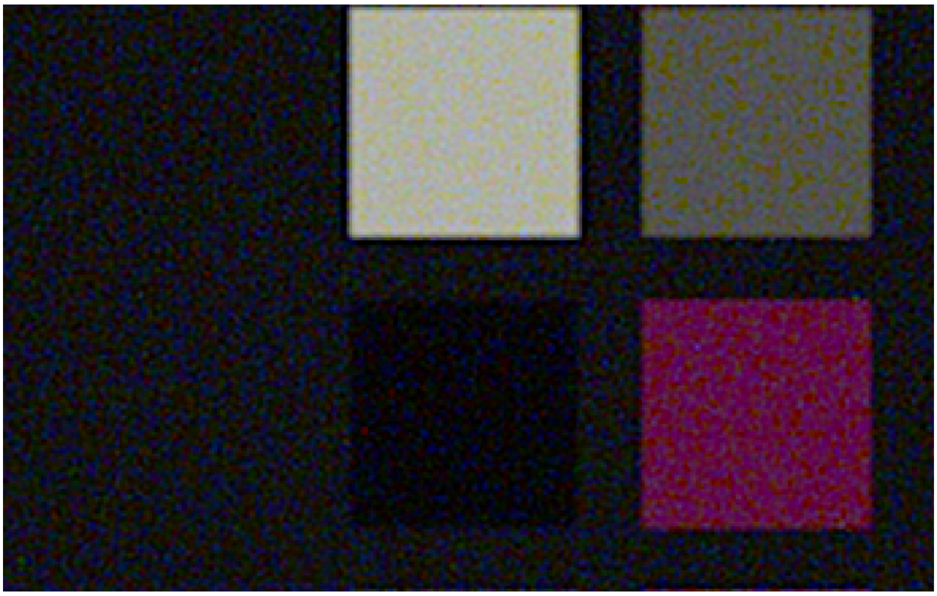


Figure 5.11: *Top left part of test scenario 1, Figure 5.9.*

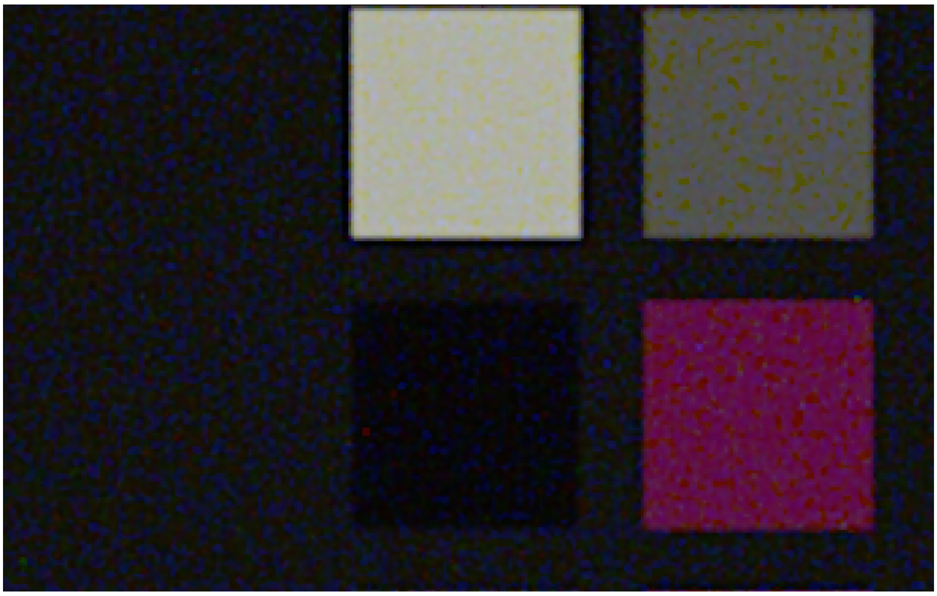


Figure 5.12: *Top left part of test scenario 1, when it has been filtered with the proposed algorithm, modified Yaroslavsky $N_{4,5}$. The noise have been blurred, but edges are kept sharp. Note that the red impulse in the black square has been emphasized, because the impulse originally spans over several pixels.*

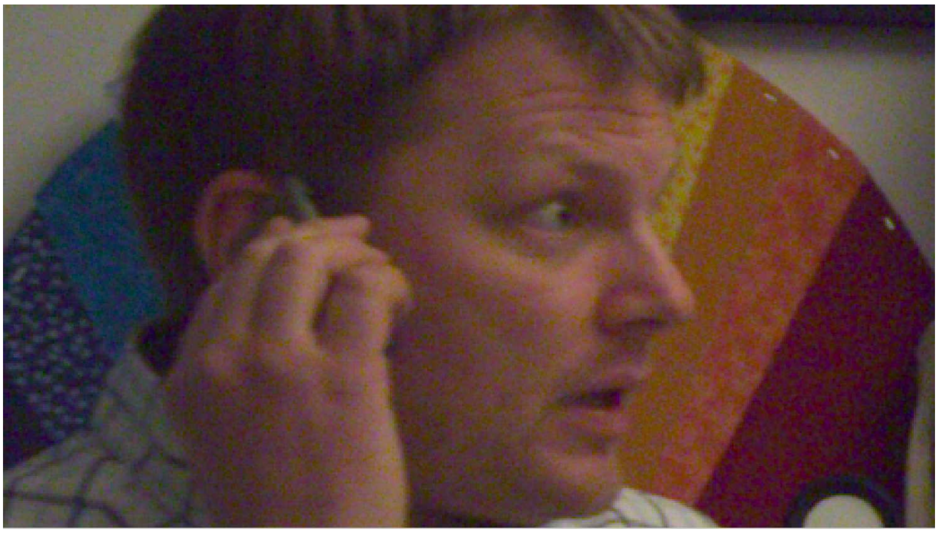


Figure 5.13: *A zoomed in version of test scenario 2.*

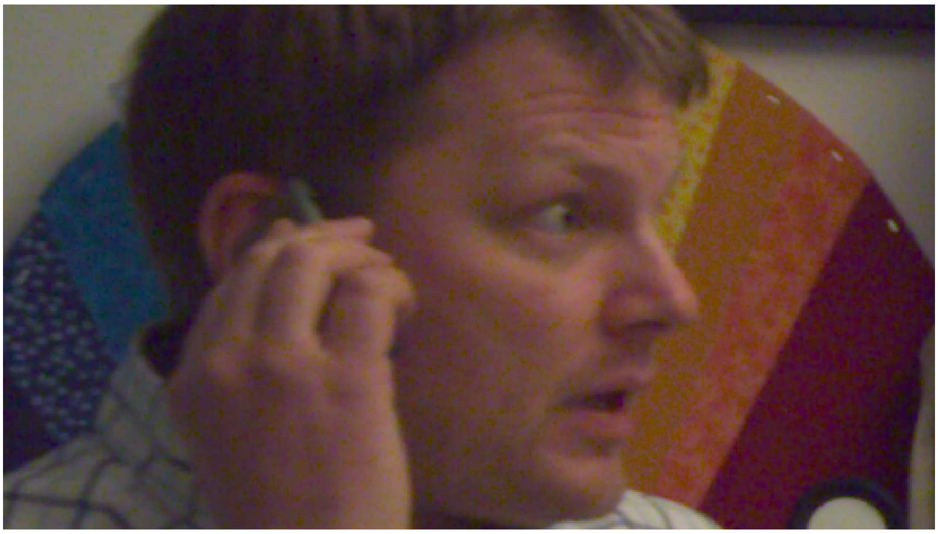


Figure 5.14: *A zoomed in version of test scenario 2, when it has been filtered with the proposed algorithm, modified Yaroslavsky $N_{4,5}$. The image is smoother with less noise, while edges and details are kept sharp.*

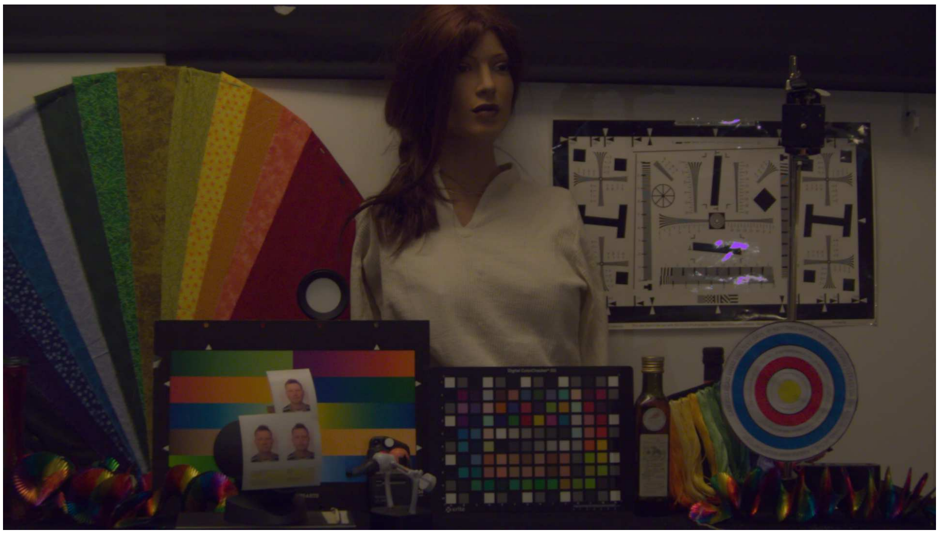


Figure 5.15: *Denoised version of test scenario 3.*

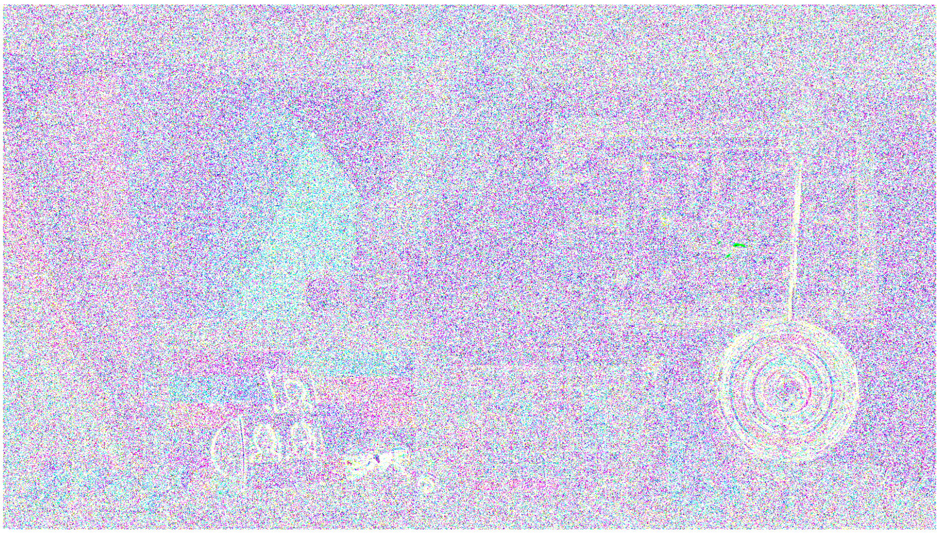


Figure 5.16: *The method noise of test scenario 3. This is what the algorithm has removed. It looks mostly just like noise, even though some details are visible, and thus have been removed from the image.*

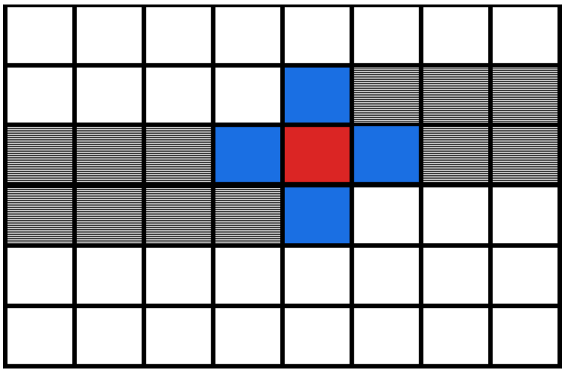


Figure 6.1: *Two lines of the incoming frame must be buffered. The bottom pixel in the 3x3 diamond shaped kernel is the incoming pixel. The algorithm thus introduces a minimum delay of two lines.*

6 Hardware implementation

When the Matlab simulations had proven the strength of the algorithm, it was written in VHDL for an FPGA implementation. Only the core functions of the algorithm is implemented, i.e., the computation of a weighted average from an input of ten 8 bit values. A flag is raised to signal for median filtering, but the median filter itself is not modeled (a median filter is available as an Altera Megafunction [56]). The logic controlling the pixel input and keeping track of lines and frames, as well as handling the frame border cases is also not modeled. One way of handling the border cases, is to pad the frame symmetrically i.e. to make a copy of the frame border, and put this copy outside the original border. A frame typically comes from the sensor serially, pixel by pixel. A whole frame of the previous filtered result must be stored in external memory. This is 1920*1080=2073600 values for each plane in full HD video, or about 6MB. In addition, two lines of the current unfiltered frame must be buffered. This is 2*1920=3840 values for full HD video, or 4kB. An explanation of why these two lines must be stored is given in Figure 6.1. The modeled logic is divided in five different modules. A reset function is included in all of them. Only the two IEEE packages std_logic_1164 and numeric_std are used. A signal flow chart is given in Figure 6.3.

6.1 Implementation details

Five pixels from the frame buffer, four pixels from the line buffer, and the incoming pixel are first sent to a module that finds the absolute value of the difference

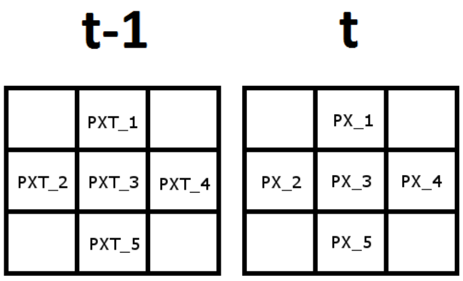


Figure 6.2: *Input pixel labels. The current frame middle pixel which is being filtered, is labeled px_3 and will be replaced by the output. This Figure shows how the input signals in Figures 6.4 and 6.5, and the VHDL code examples, should be interpreted.*

between the middle pixel and the others. This operation is done by first bit extending the 8 bit pixel value with a 0, and cast to a signed value. All the values corresponding to neighbor pixels are then independently subtracted from the value corresponding to the middle pixel, and the absolute value is found. The extra bit is then removed, and the difference values are sent to the next module (as 8 bit std_logic_vectors). The bit extension is necessary to handle negative results in the subtraction correctly. The operation is shown for PX_1 in Listing 7.

Listing 7: *VHDL sample code for PX_1 in the first module.*

```
TEMP_PX_DIFFERENCE_1 := std_logic_vector(abs(signed('0'&PX_1)
                         -signed('0'&PX_3)));
PX_DIFFERENCE_1 <= TEMP_PX_DIFFERENCE_1(7 downto 0);
```

The second module compares these nine difference values with the thresholds from the parameter register. It is also convenient to bind weights from the register to the pixels after the similarity group has been found, and count the number of group 4 pixels in case of an impulse. This is shown in Listing 8.

Listing 8: *VHDL sample code for PX_1 in the second module.*

```
if      (UNSIGNED_PX_DIFFERENCE_1 <= UNSIGNED_THRESHOLD_1) then
        PX_1_WEIGHT <= WEIGHT_1;
elsif  (UNSIGNED_PX_DIFFERENCE_1 > UNSIGNED_THRESHOLD_1) and
        (UNSIGNED_PX_DIFFERENCE_1 <= UNSIGNED_THRESHOLD_2) then
        PX_1_WEIGHT <= WEIGHT_2;
elsif  (UNSIGNED_PX_DIFFERENCE_1 > UNSIGNED_THRESHOLD_2) and
        (UNSIGNED_PX_DIFFERENCE_1 <= UNSIGNED_THRESHOLD_3) then
        PX_1_WEIGHT <= WEIGHT_3;
elsif  (UNSIGNED_PX_DIFFERENCE_1 > UNSIGNED_THRESHOLD_3) then
        UNSIGNED_NUMBER_OF_GROUP_4_PX := UNSIGNED_NUMBER_OF_GROUP_4_PX +1;
        PX_1_WEIGHT <= "0000";
```

```
end if;
```

Nine weights corresponding to the nine neighbor pixels and the weight of the center pixel are then sent to the third module, a multiply accumulator which multiplies the weight with the corresponding pixels, and adds the result of all pixels together. This intermediate result needs to be normalized before it is valid. If the weights are restricted to be a 4 bit value (0-15) and if the pixel values are 8 bit (0-255), this intermediate result will be a 16 bit value at maximum. $\log_2(10 * 15 * 255) = 15.2$. The weights are added together to find the normalize constant, in the fourth component, which also checks if more than seven (or another number from the parameter register) pixels are group 4 pixels, and raises a flag to signal for median filtering in such cases. The weights added together will at maximum be an 8 bit value, $\log_2(10 * 15) = 7.2$. The 16 bit intermediate result and the 8 bit normalization constant then matches a 16 by 8 bit divider nicely, in the fifth component. The output from the divider is the final result, unless median filtering must be applied. Every component (Figure 6.3), except the fourth, accounts for a pipeline stage. The result will therefore be ready on the fourth clock cycle. The circuit can be divided in even more pipeline stages to achieve a higher clock frequency. A good example is the multiply accumulate component which multiplies 10 values and adds the results all together in just one clock cycle. This operation can easily be divided into several pipeline stages. The best performance is generally achieved when all the pipeline stages are equally fast. This is something to keep in mind, so that no components are divided in unnecessarily many pipeline stages, which will only increase the latency.

6.2 Simulation results

The components have been simulated independently and together in Modelsim 6.5e. The clock is set to 50 MHz in the simulation, or a clock cycle of 20ns. The computation in every component happens synchronously on the rising clock edge. Scenarios with both plausible and extreme input values are tested. Two example waveforms of the algorithm's standard operation is given in Figures 6.4 and 6.5. Figure 6.4 shows that the correct result is computed on the given input, and how the do_median flag is raised when the output is invalid at 140ns when the middle pixel is 63, and all other pixels are 255. Figure 6.5 shows how changing the thresholds and adjusting the weights affects the output, even though the pixel input is the same. When the thresholds are widened, the high values 30 and 31 are included in the average and the output jumps from 15 to 19. The thresholds are then set back to default, and weights are changed instead, so that only the weight of the center pixel counts. This correctly causes the output to be the same as input px_3. When the asynchronous reset signal goes high at 150ns, the output is immediately set to 0.

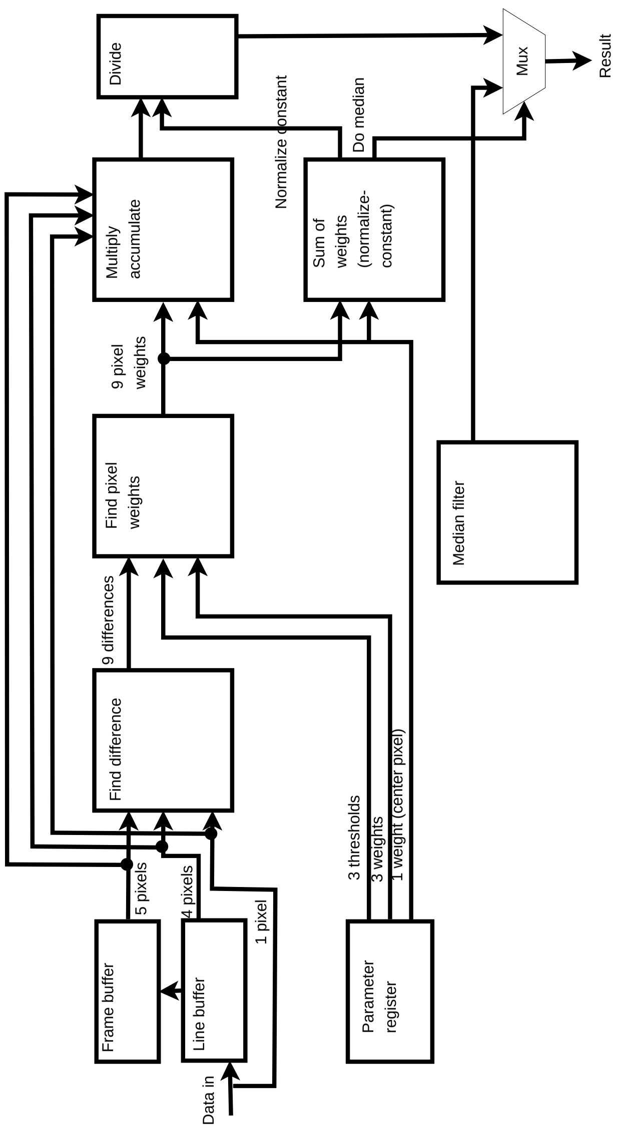


Figure 6.3: *Modified Yaroslavsky algorithm flow chart. Necessary control signals and registers are not showed.*

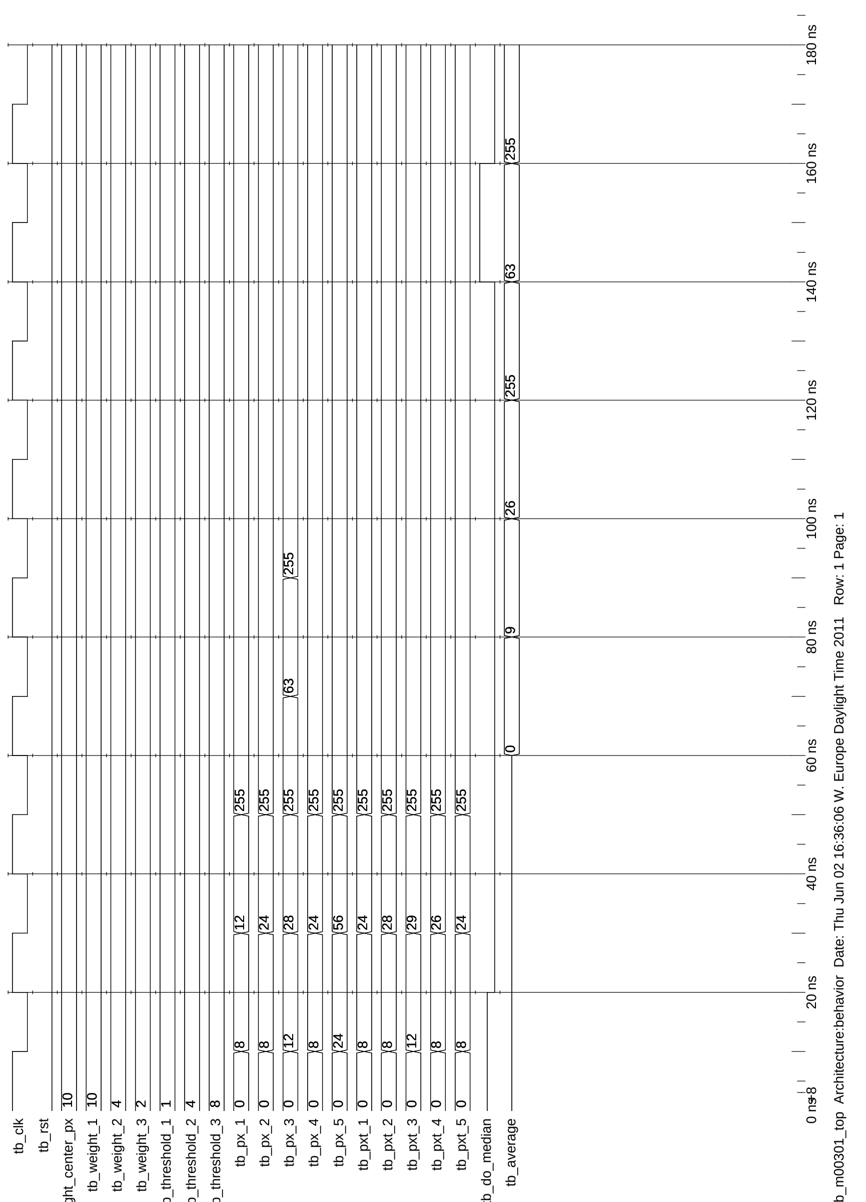


Figure 6.4: *This waveform shows that the circuit works as intended with the feasible input. Note that the do_median flag goes high the 140ns mark to signal that the output is invalid, i.e an impulse was correctly detected.*

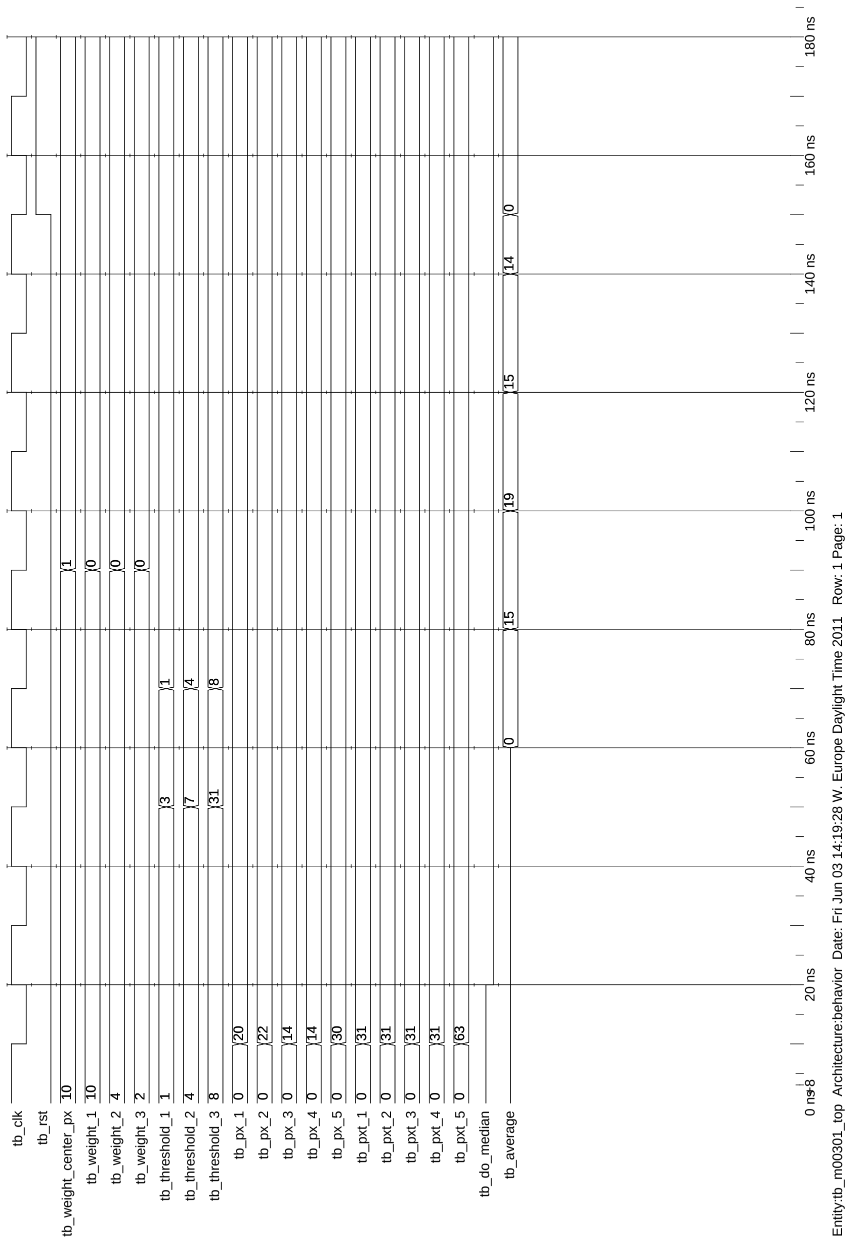


Figure 6.5: *This waveform shows how changing the input parameters (thresholds and weights) will change the output. These parameters makes the algorithm adaptable to different scenarios, e.g., cameras with different noise models.*

Table 6.1: *Synthesis summary. The algorithm should be divided in more pipeline stages to achieve a higher maximum clock frequency.*

	Logic elements	Registers	DSP blocks	f_{max}
Quartus	1587	303	**0**	35.0 Mhz
Synplify	1131	**296**	5	35.5 Mhz
Synplify w/ retiming	**1044**	345	5	**87.7 Mhz**

6.3 Synthesis results

The design was synthesized in both Quartus 11 and Synplify Pro C-2009 for the Altera Cyclone III (EP3C5F256C6) FPGA. The Cyclone III family is a low cost FPGA family from 2007, based on a 65nm process. It is suitable for both image and video processing applications. Synplify Pro has an option of doing retiming in the synthesis which can be used to optimize the clock period. This option actually made the clock frequency 2.5 times as high and freed up 87 logic elements, at the cost of 49 registers. The critical path (Figure A.2) through the multiply accumulator-component, which multiplies and adds ten numbers in just one clock cycle, and takes up 625 logic elements (Quartus Synthesis). This operation could easily be optimized for both area and performance by using the multiply-accumulate Megafunction, which is an Intellectual Property (IP) block from Altera [56]. Another component that could easily be optimized using a Megafunction, is the divider. The five DSP blocks used in the Synplify synthesis, are actually 18x18 bit multipliers used as ten 9x9 bit multipliers, which saves some logic elements compared to the Quartus synhtesis. The highest clock frequency achieved was 87.7Mhz. According to FPGA specialists at Cisco Systems, a clock frequency of at least 150Mhz should be aimed for. This ought to be achievable with some effort in optimizing the design. The synthesis results are summarized in Table 6.1.

7 Discussion

Video noise is by no means a solved problem. Modern high resolution camera sensors with high frame rates have only made the problem harder, as the data rates have become tremendous. The bitrate of uncompressed 1920x1080p60 video is almost 3Gbit/s. A lot of computation can therefore be saved by using a color space with a luminance channel like YUV. Because the human eye is more sensitive to luminance than chrominance, denoising can be limited to the luminance plane, thereby reducing the denoising data amount to about 1Gbit/s. Even though YUV is common in imaging pipelines, because the bandwidth can be reduced because of chroma subsampling, it is usually only available late in the imaging pipeline. Denoising should however be an early stage in the imaging pipeline, because other stages will be more effective on denoised data. And in the earliest stages of the imaging pipeline, only the raw image format from the image sensor is available. This usually means mosaicked RGB data. If the algorithm is to work with mosaicked data, it must select only neighbor pixels representing the same color, meaning that the neighborhood will be more spread in accordance with the CFA. If the algorithm is to work with already demosaicked RGB data, the simplest way is to process the planes individually. This can cause color bleeding as shown in Figures 2.2 and 2.3. A more sophisticated method is therefore to calculat the Euclidean distance where all three planes contribute to a mutual distance. The Matlab and VHDL implementation are however based on individual demosaicked planes.

It is obvious that blindly using of the simplest of the reviewed algorithms, e.g. Gaussian blur, will not provide good results. As shown in the project work [54], these are basic methods that must be combined with motion detectors or such, so the filtering can be reduced in certain areas of the frame. This is also true for the more advanced algorithms, such as the modified Yaroslavsky algorithm, but in this case the motion estimation is embedded as it will only use similar pixels in the averaging process anyways. If there has been motion in some of the pixels, they will not be similar, and the problem is solved. Because less pixels will be used for averaging in such cases, the total filtering intensity will be lowered. This is true for all the simulated algorithms, and is shown in Table 5.2. They all achieve a better PSNR when there is less motion, which is in accordance with Figure 1.7 (standard approach for motion estimation).

Comparing the modified Yaroslavsky with the project work algorithm, the project work algorithm used more resources to divide pixels into motion, edge, and impulse groups. The final filtering could then be carried out in a simpler way, as the normalization constant could be fixed to e.g., 2 or 32, which can be carried out by simple shift operations in the FPGA. The modified Yaroslavsky also has logic for handling edges, motion, and impulses, but these functions have been more incorporated so that the total resource usage is less than for the project work algorithm. It is also more flexible with respect to weights, and handles some cases better, as shown in Figures 3.2 and 4.3. As a matter of fact, all of the most advanced and well recognized algorithms reviewed, Yaroslavsky, Bilateral, and NL-means, have a variable normalizing constant, meaning that the division can not happen by shift operations.

Comparing the modified Yaroslavsky with the original Yaroslavsky algorithm, there have been done three modifications. The first modification is to extend the neighborhood to the spatio-temporal domain, which makes it more of a video denoising algorithm, rather than an image denoising algorithm applied on video. This is the most important extension, as pixels will be more averaged over time. A temporal function is necessary for all video denoising algorithms, and a pure spatial (2D) noise filter is actually not considered a video denoising algorithm at all, as described in Chapter 1.2 (Denoising approaches). The original Yaroslavsky is also spatio-temporal extended when compared with the modified Yaroslavsky in the simulations, as a pure spatial implementation of it would undoubtedly perform a lot worse.

The second modification was to introduce fuzzy thresholds, so different weights can be applied to pixels accordingly. This is actually the only modification differentiating the modified from the original Yaroslavsky in the simulation chapter. According to Table 5.1, this modification adds several dB gain for videos of higher quality. When the test video is of low quality, with a lot of noise, the original Yaroslavsky achieves better results. The reason for this is that the thresholds in the modifications have not been adjusted for the lower quality video. In all fairness, the original Yaroslavsky could have been adjusted to the higher quality videos as well, but the possibility of making adjustments is larger in the modified version. It is also a well known case that the Bilateral filter is more advanced and performs better than the Yaroslavsky filter. This second modification tries to mimic this superiority without being too costly in terms of computation.

The third modification is also an important one. The original Yaroslavsky has no detection of impulse noise, and an impulse will actually not be dealt with at all. This happens because no neighbors will be similar, so that only the middle pixel itself contributes to the filtered result. The detection of impulse noise can easily be implemented, but the following median filtering will require an not insignificantly amount of resources in an FPGA solution. The best way to implement the median filter, is probably by using the 2D median filter available as a MegaCore function in the Altera Video and Image Processing Suite [56].

Two different neighborhood sizes were simulated in Matlab. Surprisingly enough, the smaller $N_{4,5}$ scored better the larger $N_{8,9}$ neighborhood, despite its higher complexity and therefore also simulation time. However, the extra pixels available in the $N_{8,9}$ neighborhood is further away from the middle pixel, which can be a problem for fine details such as the grid in the synthetic video. Nevertheless, if the thresholds had been more strict, this would not matter. Another somewhat strange result, is that the project work algorithm did worse with fuzzy thresholds, whereas the Yaroslavsky with fuzzy thresholds did better. This might also be a problem with the thresholds. Setting the optimal thresholds is actually a general problem with most denoising algorithms, as it depends on the amount of noise in the video.

As shown in Chapter 1.4, another general problem, is how to compare video quality and denoising algorithms. As shown in Chapter 1.4, there are no algorithms that can predict the human perception of video qualities perfectly, especially not for no reference videos. Despite the drawbacks of PSNR, and the fact that more universal methods have been proposed, these have yet to gain popularity.

The FPGA implementation of the algorithm can make use of some optimizations. A reset signal is included in all the individual components. This is probably unnecessary as long as the denoising algorithm, seen as a single component in a larger system, has it. The four staged pipeline was a simple implementation from a designers point of view, and a good starting point for further optimizations. If the goal of the clock frequency is 150MHz, with a throughput of one pixel each clock cycle, the implementation should be divided in more pipeline stages. Putting into perspective, two lines must be buffered anyways, meaning 3840 clock cycles for full HD video. Four clock cycles is then just a drop in the bucket of the total latency. The gain of register retiming certainly shows great improvement potential.

8 Conclusion

The goal of this study was to find a fast video denoising algorithm, which could be implemented on an FPGA without using an exaggerated amount of its available resources. The proposed modified Yaroslavsky with the $N_{4,5}$ neighborhood size has proven to fulfill this goal. It is a spatio-temporal algorithm which handles motion, edges, and impulses well. It was implemented in Matlab using matrix operations and avoiding loops. Synthetic and real world Matlab simulations showed that the proposed algorithm performed overall better than the project work algorithm and the original Yaroslavsky algorithm, with high PSNR gains and a low computational complexity. There are no universal weights and thresholds, but a good starting point is to mimic the more advanced Bilateral filter.

The core functions of the algorithm was implemented in VHDL and simulated in Modelsim with belonging test benches. It was synthesized for the low cost Altera Cyclone III FPGA, using Quartus and Synplify. The algorithm required only a small part of the FPGA's available resources. The maximum clock frequency achieved was 87.7MHz, with a four staged pipeline. The pipeline can easily be divided in more stages to achieve a higher clock frequency. The total delay was four clock cycles in the pipeline, as well as the two lines of the frame that must be buffered if the frame is received and processed pixel by pixel.

9 Future work

A natural step further is to load the system on an actual FPGA, and connect it with a camera. This will require some further development so the specific video stream can be handled properly. The thresholds and weights will also need to be optimized for the specific camera. The VHDL code can also be more optimized, first of all by dividing the modules in more pipeline stages so a higher clock frequency is achieved. Second, by utilizing convenient Altera Megafunctions.

Another improvement to the algorithm, is to utilize the Euclidean distance as a function of all three planes. This would however introduce additional complexity. It could also be interesting to explore if the optimizations developed for the Bilateral filter could be translated to the modified Yaroslavsky.

Another interesting future work, would be to combine demosaicking into the algorithm, as these are similar operations, so that the total complexity of demosaicking and denoising can be lowered.

The computational power of FPGAs seems to increase in line with the demand of higher resolutions and frame rates. This means that the most powerful algorithms like the NL-means, not can be utilized before there is a break in this demand, so the FPGAs get the time to catch up.

References

[1] K. Irie, A. McKinnon, K. Unsworth, and I. Woodhead, “A technique for evaluation of ccd video-camera noise,” *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 18, no. 2, pp. 280 –284, 2008.

[2] R. Gow, D. Renshaw, K. Findlater, L. Grant, S. McLeod, J. Hart, and R. Nicol, “A comprehensive tool for modeling cmos image-sensor-noise performance,” *Electron Devices, IEEE Transactions on*, vol. 54, no. 6, pp. 1321 –1329, 2007.

[3] Wikipedia, “Photon Shot Noise,” http://en.wikipedia.org/wiki/File: Photon-noise.jpg, June 2010.

[4] E. B. Michael Langford, *Langford’s Advanced Photography.* Focal Press, 2008, ch. 6, Image Sensors, pp. 108–128.

[5] R. González and R. Woods, *Digital image processing.* Pearson/Prentice Hall, 2008, ch. 5.2, Noise Models, pp. 313–322.

[6] J. Brailean, R. Kleihorst, S. Efstratiadis, A. Katsaggelos, and R. Lagendijk, “Noise reduction filters for dynamic image sequences: a review,” *Proceedings of the IEEE*, vol. 83, no. 9, pp. 1272 –1292, Sep. 1995.

[7] D. Donoho, “De-noising by soft-thresholding,” *Information Theory, IEEE Transactions on*, vol. 41, no. 3, pp. 613 –627, may. 1995.

[8] A. Foi, V. Katkovnik, and K. Egiazarian, “Pointwise shape-adaptive DCT for high-quality denoising and deblocking of grayscale and color images,” *Image Processing, IEEE Transactions on*, vol. 16, no. 5, pp. 1395–1411, 2007.

[9] A. Maalouf, P. Carré, B. Augereau, and C. Fernandez-Maloigne, “Bandelet-Based Anisotropic Diffusion,” in *Image Processing, 2007. ICIP 2007. IEEE International Conference on*, vol. 1. IEEE, 2007, pp. I–289.

[10] B. Saevarsson, J. Sveinsson, and J. Benediktsson, “Combined wavelet and curvelet denoising of SAR images,” in *Geoscience and Remote Sensing Symposium, 2004. IGARSS’04. Proceedings. 2004 IEEE International*, vol. 6. IEEE, 2004, pp. 4235–4238.

[11] B. Song, L. Xu, and W. Sun, “Image denoising using hybrid contourlet and bandelet transforms,” in *Proceedings of the Fourth International Conference on Image and Graphics.* IEEE Computer Society, 2007, pp. 71–74.

[12] G. Chen and B. Kégl, “Image denoising with complex ridgelets,” *Pattern recognition*, vol. 40, no. 2, pp. 578–585, 2007.

[13] K. Hirakawa and T. Parks, "Joint demosaicing and denoising," *Image Processing, IEEE Transactions on*, vol. 15, no. 8, pp. 2146–2157, 2006.

[14] R. Ramanath and W. Snyder, "Adaptive demosaicking," *Journal of Electronic Imaging*, vol. 12, p. 633, 2003.

[15] S. Winkler and P. Mohandas, "The evolution of video quality measurement: From PSNR to hybrid metrics," *Broadcasting, IEEE Transactions on*, vol. 54, no. 3, pp. 660–668, 2008.

[16] M. Farias, "Video Quality Metrics," *Digital Video, Book edited by: Floriano De Rango, ISBN*, pp. 978–953.

[17] A. Buades, B. Coll, and J. M. Morel, "On image denoising methods," Technical Note, CMLA (Centre de Mathematiques et de Leurs Applications, Tech. Rep., 2004.

[18] Q. Huynh-Thu and M. Ghanbari, "Scope of validity of PSNR in image/video quality assessment," *Electronics letters*, vol. 44, no. 13, pp. 800–801, 2008.

[19] D. Salomon, G. Motta, and D. Bryant, *Handbook of Data Compression.* Springer, 2009.

[20] Z. Wang and A. Bovik, "A universal image quality index," *Signal Processing Letters, IEEE*, vol. 9, no. 3, pp. 81–84, 2002.

[21] Z. Wang, A. Bovik, H. Sheikh, and E. Simoncelli, "Image quality assessment: From error visibility to structural similarity," *Image Processing, IEEE Transactions on*, vol. 13, no. 4, pp. 600–612, 2004.

[22] Wikipedia, "Human cone response curves," http://en.wikipedia.org/wiki/File:Cones_SMJ2_E.svg, October 2008.

[23] ——, "Schematic diagram of the human eye," http://en.wikipedia.org/wiki/File:Schematic_diagram_of_the_human_eye_en.svg, January 2007.

[24] M. Lindenbaum, M. Fischer, and A. Bruckstein, "On Gabor's contribution to image enhancement," *Pattern Recognition*, vol. 27, no. 1, pp. 1–8, 1994.

[25] Pavel Holoborodko, "Noise Robust Gradient Operators," http://www.holoborodko.com/pavel/image-processing/edge-detection/, 2009.

[26] P. Perona and J. Malik, "Scale-space and edge detection using anisotropic diffusion," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 12, no. 7, pp. 629–639, 1990.

[27] L. Yaroslavsky and M. Eden, *Fundamentals of digital optics: digital signal processing in optics and holography.* Birkhauser, 1996.

[28] S. Smith and J. Brady, "SUSAN - A new approach to low level image processing," *International journal of computer vision*, vol. 23, no. 1, pp. 45–78, 1997.

[29] C. Tomasi and R. Manduchi, "Bilateral filtering for gray and color images," in *Proceedings of the Sixth International Conference on Computer Vision*, ser. ICCV '98. IEEE Computer Society, 1998, pp. 839–.

[30] M. Elad, "On the origin of the bilateral filter and ways to improve it," *Image Processing, IEEE Transactions on*, vol. 11, no. 10, pp. 1141–1151, 2002.

[31] T. Pham and L. van Vliet, "Separable bilateral filtering for fast video preprocessing," in *Multimedia and Expo, 2005. ICME 2005. IEEE International Conference on*, 2005, p. 4 pp.

[32] S. Paris, P. Kornprobst, J. Tumblin, and F. Durand, "A gentle introduction to bilateral filtering and its applications," in *ACM SIGGRAPH 2007 courses.* ACM, 2007, pp. 1–es.

[33] B. Weiss, "Fast median and bilateral filtering," *ACM Transactions on Graphics (TOG)*, vol. 25, no. 3, pp. 519–526, 2006.

[34] M. M. Bronstein, "SIMD implementation of the bilateral filter," Department of Computer Science, Israel Institute of Technology, Tech. Rep., September 2009.

[35] J. Chen, S. Paris, and F. Durand, "Real-time edge-aware image processing with the bilateral grid," in *ACM SIGGRAPH 2007 papers.* ACM, 2007, pp. 103–es.

[36] S. Paris and F. Durand, "A fast approximation of the bilateral filter using a signal processing approach," *Computer Vision–ECCV 2006*, pp. 568–580, 2006.

[37] F. Durand and J. Dorsey, "Fast bilateral filtering for the display of high-dynamic-range images," in *ACM Transactions on Graphics (TOG)*, vol. 21, no. 3. ACM, 2002, pp. 257–266.

[38] A. Buades, B. Coll, and J. Morel, "Image and movie denoising by nonlocal means," *IJCV*, 2006.

[39] F. Porikli, "Constant time o (1) bilateral filtering," in *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on.* IEEE, 2008, pp. 1–8.

[40] T.-H. Chen, Z.-H. Lin, C.-H. Chen, and C.-L. Kao, "A fast video noise reduction method by using object-based temporal filtering," in *Intelligent Information Hiding and Multimedia Signal Processing, 2007. IIHMSP 2007. Third International Conference on*, vol. 2, 2007, pp. 515 –518.

[41] T.-H. Chen, C.-Y. Chen, S.-F. Huang, and C.-H. Chen, "A Real-Time Video Noise Reduction Algorithm in the Dusk Environment," in *Third International Conference on International Information Hiding and Multimedia Signal Processing*, November 2007, pp. 531 – 534.

[42] S.-W. Lee, V. Maik, J. Jang, J. Shin, and J. Paik, "Noise-adaptive spatio-temporal filter for real-time noise removal in low light level images," *Consumer Electronics, IEEE Transactions on*, vol. 51, no. 2, pp. 648 – 653, may. 2005.

[43] J. Wu, X. Du, Y. fang Zhu, and G. Wei-kang, "Adaptive fuzzy filter algorithm for real-time video denoising," oct. 2008, pp. 1287 –1291.

[44] Y. Nie and K. Barner, "The fuzzy transformation and its applications in image processing," *Image Processing, IEEE Transactions on*, vol. 15, no. 4, pp. 910 –927, apr. 2006.

[45] C.-B. Wu, B.-D. Liu, and J.-F. Yang, "A fuzzy-based impulse noise detection and cancellation for real-time processing in video receivers," *Instrumentation and Measurement, IEEE Transactions on*, vol. 52, no. 3, pp. 780 – 784, jun. 2003.

[46] M. Ghazal, A. Amer, and A. Ghrayeb, "Homogeneity-based directional sigma filtering of video noise," in *Image Processing, 2005. ICIP 2005. IEEE International Conference on*, vol. 1, 2005, pp. I – 97–100.

[47] A. Amer and E. Dubois, "Fast and reliable structure-oriented video noise estimation," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 15, no. 1, pp. 113 – 118, 2005.

[48] K. Hirakawa and T. Parks, "Image denoising using total least squares," *Image Processing, IEEE Transactions on*, vol. 15, no. 9, pp. 2730 –2742, 2006.

[49] N.-X. Lian, V. Zagorodnov, and Y.-P. Tan, "Edge-preserving image denoising via optimal color space projection," *Image Processing, IEEE Transactions on*, vol. 15, no. 9, pp. 2575 –2587, 2006.

[50] S. Bhagavathy and J. Llach, "Adaptive spatio-temporal video noise filtering for high quality applications," in *Acoustics, Speech and Signal Processing, 2007. ICASSP 2007. IEEE International Conference on*, vol. 1, 2007, pp. I–761 –I–764.

[51] Y. Matsuo, Y. Nishida, S. Gohshi, and S.-I. Sakaida, "Reducing noise in high-resolution video sequences by using wavelet shrinkage in the temporal-spatial domain," may. 2009, pp. 1 –4.

[52] S. Chang, B. Yu, and M. Vetterli, "Adaptive wavelet thresholding for image denoising and compression," *Image Processing, IEEE Transactions on*, vol. 9, no. 9, pp. 1532 –1546, sep. 2000.

[53] R. Mahmoud, M. Faheem, and A. Sarhan, "Intelligent denoising technique for spatial video denoising for real-time applications," nov. 2008, pp. 407 –412.

[54] T. A. S. Brandsvoll, "FPGA based noise reduction in video cameras," project work, NTNU, 2010.

[55] Wikipedia, "Serial ATA," http://en.wikipedia.org/wiki/Serial_ATA, June 2011.

[56] Altera, "Altera Megafunctions," http://www.altera.com/products/ip/altera/mega.html, June 2011.

Appendix A Figures

A digital appendix is also attached, and includes Matlab and VHDL code for the modified Yaroslavsky algorithm, together with example results.

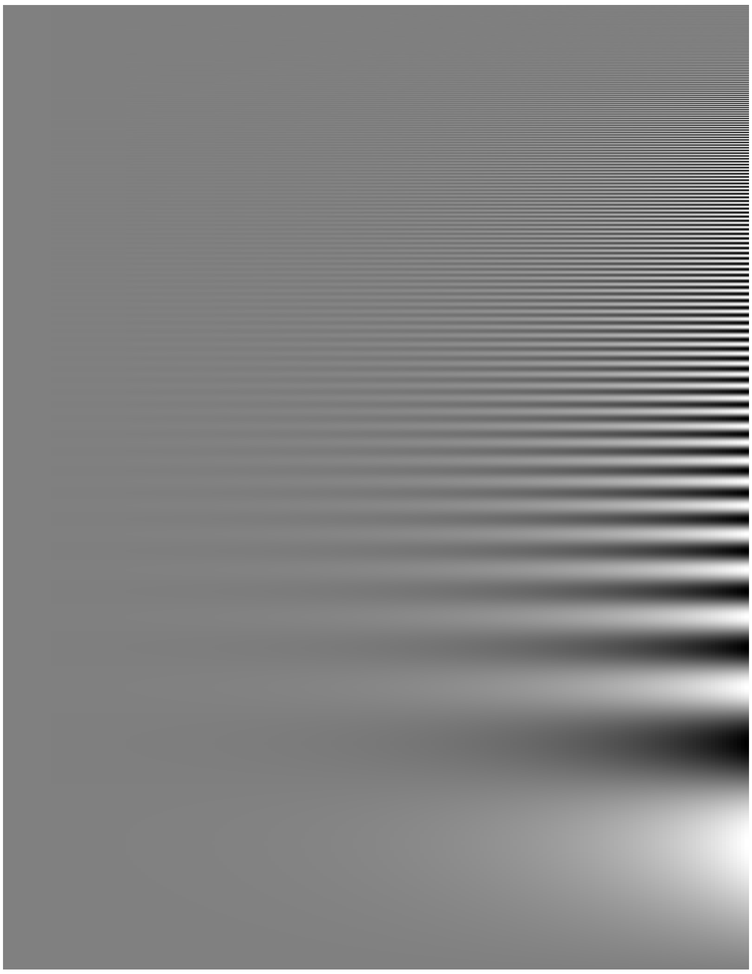


Figure A.1: *A Contrast sensitivity function. The contrast increases from left to right, and the spatial frequency increases from the bottom to the top. If the human perception of contrast was only dependent on the actual contrast in the image, the alternating black and white bars would appear to have equal length. As the perception of contrast is also dependent on the spatial frequency (the width of the bars) the bars seems longest somewhere in the middle of the image, dependent on the view distance. Because this area changes with view distance, it shows a property of the HSV, and not a property of the figure itself. Similar graphs for chroma components shows that humans are less sensitive to chroma, meaning the luminance has first priority in denoising.*

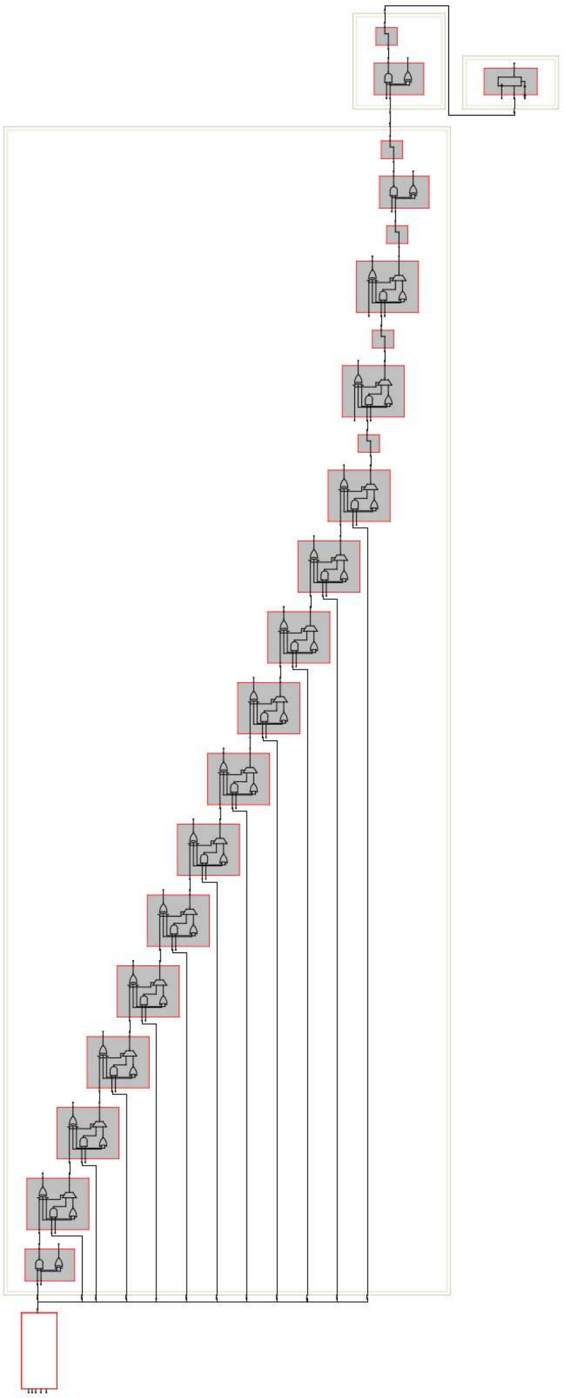


Figure A.2: *The critical path. It appears in the multiply-accumulate component.*